The Effectiveness of *t*-Way Test Data Generation

Thesis

The Effectiveness of *t*-way Test Data Generation

Michael Ellims, BSc, MSc

Submitted for the degree of Doctor of Philosophy in Computer Science

Submitted: March 2009

Revised: October 2009

Acknowledgements

I would like to give special thanks to my wife Theresa for first, allowing me to start on this project and to provide support and encouragement along the journey. Also to my son Maxwell, for being such a joy despite being such a distraction.

I would also like to thank my supervisors Darrel Ince and Marian Petre for putting up with my sometime spectacular lack of progress and for providing help and guidance when it was most needed. Thanks are also due to P.A.V. Hall and A.S.Meehan for independently tracking my progress and providing some interesting conversations on my work as it progressed. Finally to Prof. P.H.Garthwaite for advice on statistical analysis.

I would also like to thank the following persons who either made difficult to obtain work available or answered the authors' questions and otherwise provided information. Jamie Andrews, Kathy Bassin, T.Y. Chen, Rance Cleaveland, Myra Cohen, Eugenia Diaz Fernandez, Walter Gutjahr, Richard (Dick) Hamlet, W.B. Langdon, Chris Meudec, Jeff Offutt, Stuart Reid, Marc Roper, George B. Sherwood, J. Startdom, Harry Sneed, Brett Stevens and Alison Watkins.

Related Publications

The following publications are derived directly from the work undertaken as part of the PhD study program.

Ellims, M., Ince, D., and Petre, M. 2008. The Effectiveness of T-Way Test Data Generation. In Proceedings of the 27th international Conference on Computer Safety, Reliability, and Security (Newcastle upon Tyne, UK, September 22 - 25, 2008). M. D. Harrison and M. Sujan, Eds. Springer-Verlag, Berlin, Heidelberg, 16-29

Ellims, M., Ince, D., and Petre, M. 2007. The Csaw C Mutation Tool: Initial Results. In Proceedings of the Testing: Academic and industrial Conference Practice and Research Techniques - MUTATION (September 10 - 14, 2007). IEEE Computer Society, Washington, DC, 185-192.

Ellims, M. "The Csaw Mutation Tool Users Manual". Open University Technical Report. 2007/09.

Ellims, M. D. Ince, D.C. and M. Petre, M., "AETG vs. Man: an Assessment of the Effectiveness of Combinatorial Test Data Generation", Open University Technical Report. 2007/08.

The following two publications were developed in conjunctions with work undertaken as part of the PhD study program but are not directly related to the thesis presented.

Ellims, M., Bridges, J., and Ince, D. C. 2006. The Economics of Unit Testing. Empirical Softw. Eng. 11, 1 (Mar. 2006), 5-31

Ellims, M., Bridges, J., and Ince, D. C. 2004. Unit Testing in Practice. In Proceedings of the 15th international Symposium on Software Reliability Engineering (November 02 - 05, 2004). IEEE Computer Society, Washington, DC, 3-13

Other Publications

The following publications were published during the period in which the PhD. study was undertaken but are not directly connected with the work undertaken as part of that program of study.

Ellims, M. "Thoughts on the relative importance of experience and techniques". Safety-critical Systems Symposium, 2008. (Invited Paper).

Ellims, M. "Is Security Necessary for Safety?", Embedded Security in Cars (ESCAR'07) Berlin, 2006. (Invited Paper).

Bridges, J. Wartnaby, C.E. Stannard, D. Styles, J. Ellims, M. "Frameworks for Power and Systems Management in Hybrid Vehicles: Challenges and Prospects", SAE Technical Paper Series 2006-21-0005.

Ellims, M. 2004. On wheels, nuts and software. In Proceedings of the 9th Australian Workshop on Safety Critical Systems and Software - Volume 47, Brisbane, Australia. T. Cant, Ed. Australian Computer Society, Darlinghurst, Australia, 67-76. (Keynote Speaker Address).

Ellims, M., Parker, S., and Zurlo, J. 2002. Design and Analysis of a Robust Real-Time Engine Control Network. IEEE Micro 22, 4 (Jul. 2002), 20-27.

Abstract

Modern society is increasingly dependent on the correct functioning of software and increasingly so in areas that are considered safety related or safety critical. Therefore, there is an increasing need to be able to verify and validate that the software is in fact correct and will perform its intended function. Many approaches to this problem have been proposed; however, none seems likely to supplant the role of testing in the near future.

If we accept that there is, and will be, a continuing need to be able to test software then the question becomes one of how can this be done effectively, both in terms of ability to detect errors and in terms of cost. One avenue of research that offers prospects of improving both of these aspects is the automatic generation of test data.

There has recently been a large amount of work conducted in this area. One particularly promising direction has been the application of ideas from the field of experimental design and in particular, the field of *t*-way adequate factorial designs.

The area however, is not without issues; there is evidence that the technique is capable of detecting errors but that evidence is not unequivocal. Moreover, as with almost all work in the area of automatic test generation, there has been very little comparative work comparing the technique with other test data generation techniques. Worse, there has been effectively no work done that compares any automatic test data generation technique with the effectiveness of tests generated by humans. Another major issue with the technique is the number of tests that applying the technique can result in. This implies that there is a need for an automated oracle if the technique is to be successfully applied. The flaw with this is of course that in most situations the oracle is the human that is conducting the tests, a point often ignored in testing research.

The work presented here addresses both of these points. To do this I have used a code base taken from an industrial engine control system that has an existing set of high quality unit tests developed by hand. To complement this, several other techniques for automatically generating test data have been applied, namely random testing, random experimental designs and a technique for generating single factor experiments. To address the issue of being able to compare the error detection ability of all of the sets of test vectors, rather than the usual effectiveness surrogates of code coverage I have used

mutation analysis on the code base to directly measure the ability of each set of test vectors to discover common coding errors. The results presented here show that test data generation techniques based on *t*-way factorial designs are at least as effective as hand-generated tests and superior to random testing and the factor experimental technique.

The oracle problem associated with the factorial design techniques was addressed using a test set minimisation approach. The mutation tool monitored which vectors could "kill" which code mutants. After a subset of the test vectors had been run, the most effective vectors were retained and the rest discarded. Likewise, mutants that were killed were removed from further consideration and the process repeated. Experimental results show that this minimisation procedure is effective at reducing computational overhead and is capable of producing final sets of test vectors that are comparable in size with the sets of hand-generated tests and so amenable to final hand checking.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Background

Software in modern industrial society is ubiquitous. We use it in our everyday lives to wash our clothes, transport us to and from our place of work, keep our homes comfortable and even monitor our health. Given the volume of software that we depend on and given that a significant proportion can be considered safety related if not safety critical then there is a large and growing need to be able to verify and validate the correctness of that software.

To consider how many software controlled devices we rely on we need only consider the modern automobile, which can have in excess of 60 individual programmable electronic control units [190]. These entertain us, make us comfortable and translate the driver's commands to accelerate and decelerate the vehicle into control actions on the engine and coordinate these with torque demands from other systems such as anti-lock braking systems (ABS) and the transmission.

Kopetz [198] considers many of these systems to be safety critical. For example, if an adaptive cruise control (ACC) system (Valentine [307]) allows a vehicle under its control to approach too closely to another vehicle then there may be insufficient room to slow the vehicle safely. With some systems being considered today the situation is even more extreme. Consider steering systems; these can range from what Ackermann [3] describes as "*disturbance attenuation*" (pg.23) systems acting through electric power assisted steering through to full steer-by-wire systems that remove the mechanical connection between the driver and the vehicle which require both fault tolerant hardware and software (Isermann, *et al.* [179]). With such systems there may be no possibility of human intervention if a failure occurs[1].

Failures in systems containing software are attributable not only to faults within the software but also incorrect specifications, faults in the compiler and even the micro processor that the software executes on. That there are multiple insertion points for errors, from conception through implementation and into deployment is well documented by

[1] For example at highway speeds in a tunnel the collision time is less than the typical human reaction time.

Abdellatif-Kaddour, Thvenod-Fosse and Waeselynck [1], Fetzer [119], and Ellims [110]. That these all can, and do have visible effects on deployed systems is witnessed by the numerous cases documented by Neumann [241], Leveson [213] and Peterson [264]. Thus there is still a need for improved methods of verifying and validating software based systems.

1.2 Approaches to Verification and Validation

Multiple approaches have been used for verifying and validating software, ranging from informal techniques such as code walkthroughs through testing and on to formal verification. However, no technique of itself appears to be completely satisfactory.

1.2.1 Formal Methods

For example if we consider formal methods which at least on the surface appear to offer a solution, we find that even systems that have been formally proved to be correct still suffer from some hard problems.

Fetzer [119], Ellims [110] have pointed out virtually all elements in the execution chain such as unverified compilers, microprocessor hardware may contain faults which can affect the reliability and safety of the final system. In addition the environment itself can effect the final system behaviour and it is usually necessary to included in any analysis sensors and actuators of varying quality.

Within the formal method community Kneuper [194] highlights some of the same issues as [119], [110] and goes on to highlight that formal methods only account for part of the problem and are perhaps not ideally suited to some of the "softer" elements of software design such as usability of human machine interfaces.

Then there is the problem of whether or not the requirements from which the formal specifications are derived are either correct or even complete. Kneuper [194] notes that it is possible to "*create an incorrect specification which, even when implemented correctly, still results in a faulty program*" (pg. 392).

The classic example here is of course Naur's text formatting program [240] which while not formally proved correct, was intended to demonstrate how such a process can be applied. Goodenough and Gerhart [134] report that there are at least seven errors in the

algorithm as published. Moreover they also point out that London [217] corrects four of the seven errors in the original code and goes on to "prove" the resultant program correct. Goodenough and Gerhart also note that had the program been run with the example text around half of the errors would have been located.

A less well known example is the application of simulated annealing to generate test scenarios by Abdellatif-Kaddour *et al.* [1] to the control of a steam boiler. The system is of particular interest here as it has been extensively studied as an application of formal methods to a realistic system by Abrial *et al.* [2] amongst others. Abdellatif-Kaddour *et al.* claim to have found a number of new scenarios that can lead to a boiler explosion that are not explicitly exposed by any of the formal systems analysis documented in [2]. However they note the problem in general is with the requirements *not* with the analysis; a point also noted by Cuéllar and Wildgruber [85].

These two examples suggest that formal methods are probably not a complete solution, if for no other reason than human fallibility. This suggests that testing still has a role to play when developing software and perhaps especially when developing complex, safety critical systems.

1.2.2 Dynamic Testing

The term testing is usually understood to mean dynamic testing which the *Oxford Dictionary of Computing* defines as "*any activity that checks by means of actual execution whether a system or component behaves in the desired manner*". This brings out the first of the issues that need to be considered: how is desired behaviour defined and can we effectively distinguish desired from undesired behaviour? The definition continues to state that "*the system (under test) is supplied with input data, known under these circumstances as test data*". Given the first point above, one would ideally wish to select test data that will reveal undesirable behaviour.

That test data generation is an issue has been stated many times; for example Ould [257] states that this is the most significant issue associated with testing. While it is certainly a significant issue, it possibly overstates the case. As noted in the initial paragraph of this section and by DeMillo and Offutt [100] the major issue is possibly "the oracle problem": knowing when a "failure" has occurred, i.e., effectively distinguishing desired from

undesired behaviour which is generally known as the oracle problem. As highlighted by Weyuker [323] this can sometimes be an acute problem, for example when the correct results are not known or possibly even knowable before the program has been run. This remains primarily a human centred activity requiring the tester to calculate expected results by some other means or to be able to test results in a manner that do not require knowledge of the expected results, for example using metamorphic testing as explored[2] by Chen, Cheung and Yiu [54].

There are other problems associated with the test process, for example in investigating how testing is performed Teasley, Leventhal and Rohlman [293] discovered what they termed a positive test basis, that is input test data is skewed to demonstrating that software works rather than that it does not. Stacy [289] attempted to set this within a wider context of general cognitive bias and Watkins [315] offered this as a justification for automating the process of test data generation. Ellims, Bridges and Ince [112] offer an alternative reason for automating the process, at least for unit testing, in that it can be disliked intensely by those applying it.

1.2.2.1 Good Tests

Given that it is desirable for a number of reasons to automate or at least partly automate the test data generation the issue then becomes one of how to generate "good" tests, that is those that will differentiate desirable from undesirable behaviour. However, one of the longstanding major issues in test data generation is providing an exact or at least workable definition of what the term "good" actually means.

For example, Myers [238] states that the purpose of testing is to find bugs. While this may be a valid statement about a single aim of the testing process, it is of no practical use in defining a procedure for finding either bugs or for generating tests that will reveal their presence, especially if no "bugs" exist to be detected.

Myers [238] does however suggest an alternative: a graded set of code coverage criteria which require an increasingly rigorous set of requirements for testing the logic, or more correctly, the control flow of a program.

---

[2] Developed from suggestions in Weyuker's paper on testing un-testable programs [323].

The set of criteria given is as follows;

- statement coverage: every statement is executed at least once.
- decision (branch) coverage: each decision is evaluated to true and to false.
- condition coverage: executing each condition in a decision takes on all outcomes at least once.
- decision/condition coverage: each condition takes on all possible outcomes at least once and each decision takes on all outcomes at least once.
- multiple condition coverage: all possible combinations of condition outcomes are tested.

A more precise statement of the problem is given by Edvardsson [108] in terms of program paths i.e. that "*given a program P and a (unspecific) path u, generate input x ε S, so that x traverses u*" (pg. 3), where S is the set of all possible inputs.

Historically this approach has resulted in the generation of a number of path-based test adequacy measures such as statement, branch, path coverage and so on. This in turn has led to a large amount of work on generating test data that meet or model those adequacy criteria.

There is however a fundamental flaw in this approach in that it is clearly possible to execute all paths in a program without finding flaws that are present simply because they are not associated with which path is taken (i.e. domain faults) but rather with what computations are performed on a path. Hamlet has noted this issue on a number of occasions [149], [150].

Another approach is to view the problem as a data selection task. That is, rather than attempting to select data that executes specific paths, we select data based on properties that are intrinsic to the data and the problem. There are a number of such techniques including random testing, boundary value testing and combinatorial techniques. However, again it is a trivial exercise to show that on its own data selection is not a completely adequate approach, because it is possible that a branch or path will not be taken.

1.2.2.2 Limits of Testing

An obvious question to ask is: what types of errors will a test generation method discover? In their seminal investigation of what constitutes a "good" test, Goodenough and Gerhart [134] have observed, that to be useful, a test generation technique has to be

reliable. Defined informally, a reliable test is one that is "*designed not so much to exercise program paths as to exercise paths under circumstances such that an error is detectable if one exists*" (pg. 164).

Howden [169] defines the reliability requirement more formally as follows, if P is a program to implement function F on domain D, then a test set $T \subset D$ is reliable if:

$$\forall t \in T, P(t) = F(t) \Rightarrow \forall t \in D, P(t) = F(t)$$

However in the same paper he proves that there is no computable procedure that can be used to generate a nonempty finite set that can show that $P(t) = F(t)$ and draws the implication that "*the best that can be hoped for* are test *strategies which work for particular classes of programs*" (pg. 208).

1.2.2.3 Testing in the Real World

Howden's result above could be depressing if it were not that in practice testing does seem to be effective at producing reliable code, at least in some safety related applications. This has even been commented on by Hoare [162].

Kopetz [198] estimates that, in automotive applications, from the data available from the ADAC[3] [4] that the number of safety related failures in cars is in the order of 10^{-9} per hour of operation. McDermid and Kelly [222] have used data compiled by Ellims [110] from vehicle recall data and accident statistics to calculate potential failure rates for safety critical automotive systems, both giving a figure of 10^{-7} per hour per system.

Shooman [285] performed a similar study using Federal Aviation Authority (FAA) Airworthiness Directives and information on aircraft utilisation, arriving at a similar value of around 10^{-7} failures per hour.

1.3 Automated Testing

In this chapter, it has been argued that there is a significant need to be able to verify and validate safety related and safety critical software and that not even formal methods are a complete solution to the problem.

---

[3] The German equivalent of the Automobile Association (AA).

The focus is then moved to what is probably the method most used in practice for verification and validation: dynamic software testing. Several issues associated with testing are examined. These include the need to be able to build "good" test sets, and what exactly good could mean in practice and the problem of determining whether a test passes or fails. The final parts of the discussion examine two aspects of testing, first: what degree of reliability we can place on it from a theoretical view and, second, how well it seems to actually perform in practice.

The last section of this discussion suggests that even though our definitions of what a good test is or might be are weak and theoretically most techniques cannot be considered to be reliable, in practice testing, used along with other verification techniques does seem to be reasonably effective.

The incentives to automate testing as much as possible are strong: it is reasonably effective and unlikely to be displaced by formal methods completely and will therefore continue to be performed, given the fact that human generated tests are subject to cognitive bias and that in general testing is disliked provides a strong incentive to automate it as much as possible.

Thus the focus of the research carried out here to find an effective technique for automatically generating tests with the information that is readily available on most development programs. To do this two further criteria can be given. First, the technique has to perform at least as well as human testers. Second, the oracle problem must remain tractable for humans to deal with.

1.4 Key Contributions

This thesis makes the following contributions to research in this area.

First, it provides a direct comparison with a number of methods for automatically generating test sets including random, base choice and high factor *t*-way adequate test sets, and uniquely against human generated test sets developed as part of normal software production process. Importantly mutation analysis has been used to increase the discrimination ability of the tests which allows differences between the performance of techniques to be more easily shown. The research presented here shows that:

- while randomly generated test sets can be effective they are not reliable;
- that contrary to results previously published, the base choice technique performs very poorly;
- that high factor *t*-way adequate test sets are in general competitive with hand-generated test sets.

Second, it provides a practical method for incorporating high factor *t*-way testing and mutation analysis into a development process, which can avoid much of the computational overhead that may otherwise be encountered.

Third, it advances the art of mutation by showing that not only are variable declarations a legitimate target for mutation but that such mutations are detectable.

1.5 Structure of the Thesis

The thesis is divided into seven major chapters and an appendix as follows;

- Chapter 2 provides an overview of techniques that have been used to automatically generate tests data.
- Chapter 3 is a detailed review of work published in the area of using combinatorial techniques to automatically generate test data.
- Chapter 4 takes the results of the analysis of the technique performed in chapter three and highlights the weakness with the current state of the art and puts forward a program of work to be undertaken to address those weaknesses.
- Chapter 5 provides the detail of a number of experiments undertaken to investigate the hypothesis's developed in Chapter 4.
- Chapter 6 presents the discussions and final conclusions.
- Chapter 7 provides an analysis of how the research present here can be carried forward.
- Appendix A, examines in detail the Csaw mutation tool and compare its capabilities with other existing work.

2. Automatic Test Data Generation: an Overview

2.1 Introduction

A large number of techniques have been investigated to automatically generate test data for software. This section examines those techniques that are directly applicable to the types of information that are currently available, i.e., the code itself, and information about the data that is being operated on and is produced. Information available from requirements can of course be implicitly included in this but techniques such as formal methods have been excluded. Although such methods are used in a limited number of high integrity situations, their use in industry currently is not that wide spread.

The techniques discussed in this chapter have been classified very broadly into four main areas:

- Section 2.2 on random testing;
- section 2.3 on mathematically inspired techniques;
- section 2.4 on adaptive testing;
- section 2.5 on symbolic execution.

Each of these sections in turn includes a brief introduction to the technique being examined, a review of the work conducted, an analysis of the literature and a summary.

2.2 Random Testing

2.2.1 Random Testing: Introduction

The traditional view of random testing is summed up by Myers [238] who stated that "*probably the poorest methodology of all is random testing*" (pg. 36). Arguments against random testing are based for the most part on the concept that test data selected randomly will have a low probability of detecting an error. Myers used this argument as did Beizer [31] who stated that, relative to the boundary value analysis criteria of verifying boundaries and testing at points where it is known that bugs reside; "*what is the probability that a set of randomly chosen test points will meet the above criteria? End of argument against random testing*" (pg 200).

However, this view was questioned by Duran and Ntafos [107]. Their study consisted of two main parts: firstly, a simulation of the *expected* effectiveness of random versus partition testing and, secondly, an examination of the fault detecting ability of random testing on a small set of programs.

Work on random testing falls broadly into three main categories: simulation studies such as those performed by Duran and Ntafos [107]; analytical studies such as those performed by Weyuker and Jeng [325]; and a large body of work that, as suggested by Ince [175], uses random testing as base method for comparison with other more complex techniques in experimental studies.

2.2.2 Random Testing: the Technique

Ince [176] has defined random testing as a process that involves the random selection of data values from the input domain of the software unit under test. It is usually assumed that the input domain is the set of integers from m to n and in such a domain the mechanical process of generating the data is well understood. However, it should be noted that different programs require different types of random data to be generated. For example, in the experiments conducted by Frankl and Weiss [125] [121], the authors noted that each program used in their experiment required its own method of selecting "random" test sets.

The observation that random test data generation is conceptually simple only when numbers are considered is noted by Hamlet [150] who used the example of test data generation for a compiler to illustrate this issue. Random test generation has been used for compiler testing by Bird and Munoz [36] but because of their need for programs that both compiled and ran, the construction scheme adopted by them appears relatively complex. Therefore, it would appear that there is a continuum of programs that range from those in which it is trivial to generate random test data, to those for which it is probably impossible and/or meaningless, for example, software that simulate physical systems such as the state of the atmosphere or structures such as oilrigs.

There is also the issue of how random data should be extracted from the input domain. For example, it is usually assumed that random data is generated from a uniform distribution with replacement, i.e., that all inputs have equal probability of being selected. However, it may be advantageous to select from a different distribution. In particular, if

statistical inferences are to be derived from random testing, then the distribution of test cases should perhaps mimic the operational profile of the program in use. Hamlet [150] noted, however, this approach is not without its own problems. For example, the operational profile may be unknown or uninteresting.

2.2.3 Random Testing: Simulation Studies

The work reported in Duran and Ntafos [107] is based on the idea of partitioning a domain D into k subdomains, $D_1 \ldots D_k$ of size $d_1..d_k$ where each subdomain has associated with it a failure rate of θ_i. Thus for a single test vector selected randomly from D, there is a probability p_i that it will execute in subdomain D_i. It is assumed that the number of test vectors n is equal to k for both random and partition testing and that for partition testing one test vector is selected from each subdomain[4].

Duran and Ntafos performed a number of simulations with varying values for k, θ_i and p_i and reached the conclusion that random testing is an effective test technique if it is assumed that it is less expensive[5] to generate random test vectors than to use another technique, for example hand-generated partition testing.

Hamlet and Taylor [152] performed a similar set of experiments, as it was felt that the results presented by Duran and Ntafos [107] were "*counterintuitive*". The experiments reported used the same model as Duran and Ntafos [107] and arrived at essentially the same conclusion: that is, a small increase in the number of randomly generated test vectors would be sufficient to overcome any advantage that partition testing may have. An investigation into the effects of homogeneity[6] also produced similar results. However, it is significant that for small failure-prone partitions, they found that partition testing performed better than random testing.

---

[4] Weyuker [325] gives a clear and detailed exposition of the background theory and assumption used in Duran and Ntafos [107] and Hamlet and Taylor [152].

[5] An exact definition is not supplied by the authors who use the term "cost effective", it is assumed that they are considering total monetary cost.

[6] A homogeneous subdomain is one where any input will reveal a failure if one is present, i.e., probability of failure in the subdomain is 1 or 0.

The simulations performed by Tsoukalas *et al.* [304] also showed similar results thereby adding more weight to the argument that random testing was more "cost effective" than partition testing. Moreover, under the assumptions built into their model they gave an experimentally verifiable prediction that $2n$ test vectors generated at random were required to achieve the same or better fault detection probability as n test vectors generated using partition testing techniques.

The simulations conducted by Ntafos [244] seem to demonstrate that for lower numbers of tests, partition testing outperforms random testing when tests are allocated proportionally to the size of the partition as proposed in by Chen and Yu [56]. However as in previous simulation studies, there is a region where proportional partition testing performs better than random testing overall. Overall, the conclusion that Ntafos comes to is that if the cost of performing random testing is lower than the cost of performing partition testing then random testing still has the advantage.

This body of work seems to show that partition testing may be of little value if the cost of testing using randomly generated test vectors is lower than of partition testing. Nevertheless, whether this is in fact the case this remains an open question and is dependent on a number of factors.

How representative of real programs are the simulations? For instance, we must consider that some partitions may be much more difficult to hit than other partitions. Experimental work in section 2.2.5 seems to indicate that is indeed the case as demonstrated empirically by Michael *et al.* [232]. Random testing may not do as well here as partition testing.

How well do partitions model actual partition sizes? It could be that real programs contain many more small partitions than larger ones, for example, it is not unknown for code to fail on single values.

What is the true cost of performing random testing? The assumption that is generally made that it is lower. However it is not clear that this is the case in practice, for example the generator described in Bird and Munoz [36] is highly complex and Frankl and Weiss [125] commented on the need to build problem specific generator functions as does Hamlet [150]. Claessen and Hughes [61] also noted that fine-grained control of the generation process is necessary for testing to be effective.

Finally, there is the need for an effective oracle. The cost of testing has two major components: one of which is the generation of test data; the other is being able to recognise that a test has failed. It may be that this second component is more expensive for random generators than partition testing simply because that the data generated follows no pattern.

2.2.4 Random Testing: Analytical Studies

Weyuker and Jeng [325] performed an analysis of random *versus* partition testing using the models proposed by Duran and Ntafos [107] and Hamlet and Taylor [152] as the starting point. They concluded that, as suggested by Hamlet and Taylor, the effectiveness of partition testing depends on how inputs that result in failures are concentrated within partitions. Therefore, partition testing can be better, the same as, or worse than random testing. Optimally, partitions should be selected to concentrate faults within particular partitions. However in general there is often no such effective partitioning strategy.

Work by Chen and Yu [56], [57] showed that under certain assumptions partition testing will perform better than random testing, namely when the number of test cases is equal, when test cases are selected from a uniform distribution with replacement, when almost all subdomains are homogenous and that for some domains the number of failures is small. In [58] Chen and Yu proposed a technique for allocating tests across partitions. However, the practical utility of their approach may be low because while the first two conditions are trivial to meet, in practice the second two may not hold. In particular, the assumption on subdomans being homogeneous may be incorrect given that boundary testing appears to be such an effective technique (see section 2.2.5).

Nair *et al.* [239] examined the issue of partition *versus* random testing from the view that partition testing could be considered to be stratified random sampling. The authors stated that "*it is well-known in the statistical literature that stratified sampling enjoys many advantages over simple random sampling*" (pg. 168). Taking this approach they reached a rather more unequivocal view on the effectiveness of the two techniques than the majority of other work. Namely that in general partition testing is superior if the partitions are not selected at random and the test cases are selected independently. It is interesting that this result was obtained by mathematicians and appeared in that literature rather than the traditional computer science literature.

The approach taken in the analytical work by Frankl *et al.* [123] examined delivered reliability. That is the probability that the software will fail when operating under its operational profile, rather than the probability of finding a defect as used in the Duran and Ntafos [107] study and this is an interesting variation on the usual effectiveness measures. However, again the results are not unambiguous. Once more we have an analysis that indicated that the relative advantage the two techniques have over each other is dependent on the nature of the problem.

Gutjahr [148] examined the same problem, but extended the deterministic model used with a probabilistic one. The main conclusion of this work was that in certain circumstances partition testing could be up to k times more effective (upper bound) than random testing, where k is the number of partitions. This situation arises, the authors show, when there are many small sub-domains and a few large sub-domains and when the sub-domains are homogeneous.

The results given in Boland, Singh and Cukic [38] are significant as they confirm the results presented in Weyuker and Jeng [325] and Gutjahr [148] using vector ordering techniques. Like Nair *et al.* [239], Boland, Singh and Cukic [38] stated under what conditions partition testing can be expected to perform better than random testing. However, unlike Nair *et al.* [239], these conditions are based on the relative values of the failure rates and as such are of only limited utility in practice.

As with the models used in the simulation experiments, the analytical studies are simplifications of an actual partition testing situation and, as before, the validity of the simplifying assumptions actually are not clear.

2.2.5 Random Testing: Empirical Results

Given the results above, we need to examine the literature to see what supporting evidence, if any, is available to support the view of the simulation studies that random testing is more effective than partition testing. There are several specific questions that need to be addressed:

- What effect does the size of the partitions have?
- How representative of real programs are the simulations?

- Do experimental results support the idea that random testing requires only a limited number of more test cases?

The most studied program in testing literature is the triangle program introduced by Myers [238]. Gutjahr [148] examined a version of the triangle program and found that the size of the partitions for equilateral, isosceles and scalene triangles were $O(n)$, $O(n^2)$ and $O(n^3)$ respectively[7] where n is the size set of integers. Thus the partition containing equilateral triangles is extremely small[8] compared with the other two partitions and thus we should expect the program to be difficult to test using randomly generated data.

This is, in fact what was observed, by both Deason *et al.* [97] and Michael *et al.* [232] when performing random tests on versions of the triangle program. In [97] it was found that there was no improvement in the number of condition outcomes covered when the number of test vectors was increased from 45 to 504. In Michael *et al.* [232] 10,000 test vectors were tried but random testing never exceeded 49% condition/decision coverage.

Additional evidence comes from Jones *et al.* [183] who compared randomly generated test sets against data generated by adaptive methods (see section 5.2) for four small programs. In all cases, the number of random test cases required to achieve branch coverage agreed closely with what would be expected from the partition sizes.

It should be expected that this type of situation would meet the requirements that several authors deem necessary for partition testing to be more effective than random testing. This type of small partition does seem to occur in practice. For example, in an experiment that tested *N*-version programming, Vouk, McAllister and Tai [312] discovered a set of faults with probabilities of detection by random testing of 10^{-6}, 10^{-12}, 10^{-24} and 10^{-36}.

Similarly, Michael *et al.* [232] performed a number of tests on synthetically generated programs with varying levels of nesting and condition complexity. Their results for the percentage coverage of condition/decisions are summarised in Table 1, where the nesting levels used were 0, 3 and 5 as shown in the top row of the table, and the condition complexity levels used were 1, 2 and 2 as shown in the first column.

---

[7] The values given in the paper are incorrect due to a printing error, personal communication 2002.

[8] The ratio is for n/n^2 is 1.5×10^{-5} and n/n^3 is 2.3×10^{-10} for 16 bit integers.

Table 1. Summary of the percentage condition/decision coverage achieved for synthetic programs generated as part of a study by Michael *et al.* [232].

	0	3	5
1	95	78	73
2		61	48
3		47	27

These results indicate that there is *an* effect related to nesting and conditional complexity that has not been taken into account with the model of partitions used in simulation and analytical studies.

In light of the above discussion it is instructive to look at the small programs examined in a number of studies - namely Duran and Ntafos [107], Frankl and Weiss [125], [121] and Michael, McGraw and Schatz [232], on which random testing was successfully performed.

When this is done, one feature is readily apparent. The majority of the programs have a single input presented either in the form of an array or a matrix. Therefore, the data space may not form partitions of the type exhibited by the triangle program or programs such as those simulated in Michael *et al.* [232]. Even the programs that take two inputs place few constraints on what can be valid combinations. For example, any two numbers have a greatest common denominator (GCD) that can be found using Euclid's algorithm. Thus, they do not provide strong supporting evidence for the effectiveness of random testing.

The results given by Frankl, Weiss and Hu [121] for the expected number of tests cases that are required to make random testing as effective as the data flow and mutation techniques are interesting. Of the 18 variations of the nine programs examined, for six the random test set was projected to require the same number of test vectors. For three programs the expected test sizes were two or three times the "optimal" size; one program each at factors of four, six, seven, ten and twelve times the "optimal" size. Finally, for four of the subjects the authors projected that an *infinite* number of random tests would be required.

Empirical work performed by Thevenod-Fosse *et al.* [294], [296] provide comparative results relative to mutation adequacy for three techniques; random testing with uniform distributions; statistical random testing; and structural test sets constructed to meet coverage adequacy requirements on the four programs. Their major results were that;

- Uniform random testing performed poorly: killing only 76% of mutants, leaving 685 alive. Structural testing killed 85-99% leaving between 312 and 405 alive[9]. Statistical testing killed 99% leaving only 6 alive.
- Random testing performed well on two programs but very poorly on the other two.
- There was no relationship seen between the stringency of the structural criteria and its performance.

Furthermore, significant numbers of random test cases were required, with 850 being used for one program, which would be a significant overhead if results needed to be examined manually. However, this work does provide some empirical support for Chen's [56] assertion that it is more effective to allocate tests proportionally.

This work by Thevenod-Fosse *et al.* [296] also supports the observations by Frankl, Weiss and Hu [121] that in some cases random testing performs very badly. Here, random testing performed well for only 2 of 4 test programs and for one performed very badly killing fewer than 60% of the mutants. Results from latter work by Thevenod-Fosse and Waeselynck [295] were no more encouraging with only five of twelve faults detected.

Other experiments have provided mixed results. Hutchins *et al.* [174] examined a set of seven programs of 141 to 512 lines of code to investigate the effectiveness of dataflow and control flow adequacy criteria for discovering seeded faults. Of the 106 faults investigated in detail, dataflow adequate test sets out performed edge adequate and random test sets in 31 cases; and edge adequate sets out performed dataflow and random in 25 cases. For a further 32 cases there was no difference in the performance of dataflow and edge adequate sets but both out performed random test sets. In nine cases, the random test sets outperformed dataflow and edge adequate test sets. Finally, nine could not be classified. In addition, it was found that only test sets in the 99-100% dataflow and edge coverage area were really effective and that a predicted 160% increase in the size of the random test sets would give the same test effectiveness.

Reid [269], [270] conducted a set of experiments on 20,000 lines of production avionics software written in Ada. In [269] Reid examined the effectiveness of equivalence partitioning, boundary value analysis and random testing; this work was extended in [270]

[9] This varied according to which test adequacy criteria test sets were designed to meet.

to encompass modified condition/decision coverage (MCDC) and branch condition combinations testing. Reid concluded that random testing seemed to have the advantage over equivalence partitioning that required similar numbers of test cases. It was noted that equivalence partitioning was better at revealing sub-domains but that these occurred in only three of the 17 modules studied. Significantly, random testing was nowhere near as effective as boundary value analysis - approximately 35,000-50,000 test cases were required for random testing to show the same fault detection rate.

Frankl and Iakounenko [124] investigated the fault detection ability of branch adequate, all-uses adequate and random testing for fixed test set sizes drawn from a universe of 10,000 random test cases generated using a generator developed by Pasquini *et al.* [262] for an antenna configuration program developed for the European Space Agency. The major findings of this study were that for 7 of the 8 faults present there was large increase in effectiveness as coverage increased and both techniques were more effective than random tests set of the same size.

Frankl and Deng [122] used the same experimental technique to investigate the results from [124] using a larger sample (100,000 cases) of randomly generated vectors to create fixed size test sets to meet branch and dataflow adequate coverage criteria. They then compared the effectiveness of those sets to results obtained from operational testing. The results from this empirical study were similar to [124] in that test sets with high coverage had higher values of reliability improvement than random test sets of the same size (50 vectors). Furthermore, as the coverage increased probability of achieving higher reliability also increased. However, the authors noted the extra cost associated with applying these techniques.

The experiments detailed above probably comprise the best empirical evidence available that compare partitioning and random testing. In summary:

- All the studies found that for the majority of cases partition testing performed better than random testing for the same number of test cases.
- When random testing appeared as effective as partition testing, the number of extra test cases were close to the number predicted by simulation studies most notably the factor of two predicted in Tsoukalas *et al.* [304].

- When random testing did not do as well as a coverage based adequacy criteria it fails spectacularly, most notably in examples provided by Frankl and Weiss [121], Reid [269], [270] Deason [97] and Michael *et al.* [232] where the number of additional test cases required for random testing to be as effective was orders of magnitude larger.

One further experiment is of interest, Godefroid *et al.* [131] used symbolic execution targeted at finding data to locate execution failures (exceptions) and assertion violations. Empirical evaluation was performed on a small air-conditioning control example and on an implementation of the Needham-Schroeder public key authentication protocol. In both cases, assertion violations were located in reasonable time periods (1 second and 22 minutes); in neither case did random testing find solutions after several hours of searching.

The conclusion that can be drawn from this is that although randomly generated tests can theoretically be expected to perform well in some, or possibly even most cases, in general the technique does not appear to be reliable in extremis.

In light of the above, it still seems necessary to try to establish a better understanding of what the limitations of random testing are using an empirical approach. This may be possible using a simulation method similar to that used by Michael *et al.* [232] and by examining a larger set of programs drawn from industrial applications. It is worrying that essentially the same observation on needing a better understanding of random testing was made by Ince [175] over two decades ago and that the question appears to be no closer to being finally resolved.

An interesting question that arises out of the work surveyed is whether we are examining the correct question. The majority of the investigations have compared random testing with partition testing where a very generous definition of what constitutes a partition has been allowed, most notably based on code coverage or data flow criteria. Chen *et al.* [55] have noted that both simulation studies and analytical work make the assumption that any data point in a partition is as good as any other. However there are indications that stronger partitioning criteria such as those imposed by boundary value analysis are much more effective than randomly generated tests (for example, Reid [269], [270]). Nair *et al.* [239] make this point particularly strongly by stating that "*for any given partitioning of the input domain, gains in efficiency can be achieved by judiciously choosing the test allocation*

scheme. The importance of doing this does not seem to be fully appreciated in the software testing literature" (pg. 168).

2.2.6 Random Testing: Summary

The critical paper in the literature on random testing is the seminal work by Duran and Ntafos [107], which challenged the traditional view that the technique is not being very useful as expressed by authors such as Myers [238] and Beizer [31]. This critical paper in turn has lead to an avalanche of work that has both simulated the situation such as Hamlet and Taylor [152], Tsoukalas *et al.* [304] and Ntafos [244] in an attempt to clarify the situation. Likewise a large amount of rather inconclusive analytical work first by Weyuker and Jeng [325] and latter led by Chen [56], [57], [58][10] has been performed in an attempt to refute the view. The only clear argument against the effectiveness of random testing comes from Nair *et al.* [239].

Finally, the strongest argument against the usefulness of random testing comes from examining the empirical work performed by researchers such as Frankl and Weiss [121], Reid [269], [270] Deason [97] and Michael *et al.* [232]. This work demonstrates that the number of additional test cases required for random testing compared with partition testing can be orders of magnitude larger than those required for techniques based on partition testing.

Figure 1 shows the chronological progression on work cited in this section as divided into the three major themes examined, that is simulation studies, analytical studies and empirical work. In addition, papers that have directly used the triangle program have been highlighted.

[10] More papers than this have been produced but have not been cited in the interests of brevity.

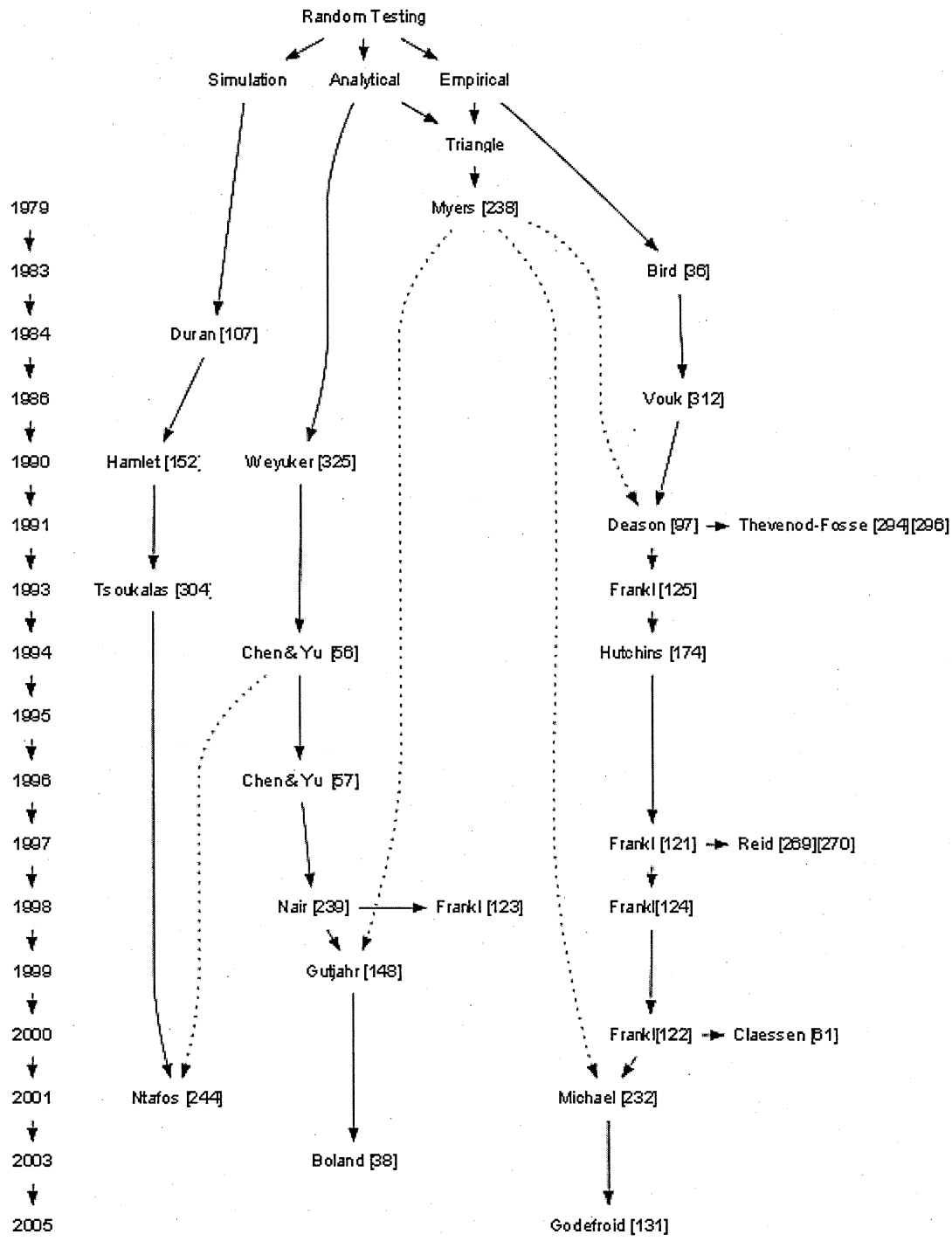

Fig. 1. Chronological order of major work examined under random testing, organized by area of activity.

2.3 Mathematically Inspired Techniques

2.3.1 Mathematically Inspired Techniques: Introduction

This section covers a number of techniques for generating test data based on mathematical relationships between data items. The concept of using data coverage as a testing criterion was first proposed by Sneed [288]. However, while Sneed's work bears some similarity with the techniques discussed here in that it uses defined domains, it differs in that it does not propose a technique for generating the required data, but rather concentrates on the dynamic measurement of data coverage.

Grindal, Offutt and Andler [140] have classified the majority of the methods discussed in this section as combinatorial techniques. These authors provide taxonomy for combinatorial techniques based on the construction methods and degree of randomness involved in their construction. However, this classification sometimes separates techniques that result in test sets with similar properties and sometimes groups together techniques that produce test sets with differing properties. Consequently, an alternative classification is used here that closely matches the major points of the subsumes relation developed by Grindal *et al.* [140], but distinguishes between the techniques based on the authors intentions.

The three classifications used here are as follows:

- Geometric techniques.
- Boundary following techniques.
- Combinatorial techniques.

Although all these techniques can be considered to involve systematically finding combinations of data values (as opposed to a possibly non-systematic data set constructed by hand), there are several features that differentiate between the techniques. For example, the original motivation of the techniques originators appears to be different. Thus, the primary aim of Malaiya [219] was to remove a perceived weakness in random testing by ensuring points are maximally separate and even suggests that his technique could be combined with combinatorial techniques. In Cohen *et al.* [67] the aim was to cover all pairwise interactions between variables.

Similarly, the selection of data points is different in each technique. For example, anti-random testing [219] attempts to gain maximal coverage of the input data space. Hoffman *et al.* [164] applied domain boundary value testing, whereas combinatorial work is focused on input space partitioning (Ammann and Offutt [8]) using techniques such as equivalence partitioning and Ostrand and Balcer's [256] category-partitioning technique.

The original authors of the techniques presented here do not cite each other's work, i.e., Mandle [220] is not cited by Malaiya [219] whereas Cohen *et al.* [67] does cite this work but does not cite Malaiya [219].

There are a number of techniques examined that do not fit into the combinatorial framework but nevertheless are based on mathematical concepts.

Support for the view that these techniques are best treated separately comes indirectly from the subsumes relationship given in Grindal *et al.* [140] for *t*-way coverage for combinatorial techniques and explicitly excludes the anti-random [219] and boundary following techniques developed in Hoffamn *et al.* [164].

Despite the differences highlighted above it is believed that it would be more appropriate to keep the topics grouped together to emphasize the connections between them rather than separate them out into distinct parts. As a result, this section is somewhat longer than other sections.

2.3.2 Mathematically Inspired Techniques: Geometric Methods

Malaiya [219] proposed a scheme where distance is defined either in terms of Cartesian coordinates (Euclidian distance) or Hamming distance. Perhaps, somewhat surprisingly, the technique was not applied directly but was utilised via checkpoint encoding in which selected points were encoded as binary fields within a single bit string. One advantage of this scheme is, of course, that each vector in a test set can be represented as a single bit string, which reduces the computational overheads of applying the technique. Table 2 shows a simple case of a checkpoint encoding of a single integer variable.

Another of the claimed advantages of this encoding scheme is that it allows objects such as data structures to be assigned distances. The exact encoding of the variables is dependent on an analysis that uses techniques such as domain, partition and boundary analysis, all of which could be partly automated.

Table 2. Assignment of checkpoint codes to values of a variable.

min - 1	min	internal	internal	max	max + 1
000	001	010	011	100	101

Malaiya [219] also suggested that this technique could be combined with the combinatorial strategies discussed in section 2.3.4 and chapter 3 but did not elaborate on this concept.

Yin *et al.* [343] expanded on the work developed by Malaiya [219] by presenting an extended example of testing the triangle program. The main focus of their work was to obtain effective code coverage. To achieve this, several attempts at generating adequate specifications for the data relationships were required. In addition, the data specification was quite complex, which reduced the practical utility of the technique.

Wu *et al.* [337] and Yin [342] explored the applicability and effectiveness of anti-random *versus* random testing for hardware fault coverage for simple types of fault, i.e., stuck-at and bridging faults. Although [337] reported that for small numbers of vectors (50-60), anti-random data performed significantly better than randomly generated vectors, the results in Yin [342] did not replicate this. Interestingly Wu *et al.* [337] investigated the effectiveness of selecting inputs to ensure an anti-random pattern for output data. Their analysis showed no clear advantage (nor disadvantage) over anti-random testing applied to input data and the authors speculate this is because the anti-random properties of the output data were not preserved internally.

A variation of the distance function was proposed by Mayrhauser and Bai [311] to improve the computational efficiency of the geometric technique. In particular the original method required enumeration of the input space and distance computations for all vectors and the vector representations had to be binary, hence the introduction of checkpoint encoding. It was proposed by these authors that this approach might lead to the types of issues with fault detection effectiveness that were suggested by work reported in Hamlet and Taylor [152], which dealt with partition *versus* random testing.

The last of these points is interesting in the light of results by Nair [239] and Guntjahar [148] on random versus partition testing. Those results suggest that the overlaying of partitions via check-point encoding could potentially be the more effective approach.

The solution proposed by Mayrhauser *et al.* [311] to the issue of checkpoint encoding is threefold:

- average all *existing* test vectors into a single vector, the centroid;
- locate the set of vectors that are orthogonal to the centroid;
- determine which of the orthogonal vectors is maximally distant.

Mayrhauser *et al.* expressed their solution in terms of binary vectors where each bit represented a dimension of the problem space. However, in principle it could be extended to deal with integer and floating point values.

Kobayashi *et al.* [197] conducted experiments with anti-random testing and found that results were better than random testing. This work is examined further in chapter 3.

Finally, in Chan *et al.* [53] it was observed that failure causing regions fall into one of three types: point failures, "strips" though a domain and "blocks" within the domain. To improve the probability of hitting one of these failure-causing regions - in particular the strip and block type regions they proposed a technique closely allied to anti-random testing - Adaptive Random Testing (ART). This technique uses essentially the same measure of distance as Malaiya [219] but without the checkpoint encoding and also uses a randomly generated candidate test set in a similar manner to Cohen *et al.* [67] to select the member furthest away for all data points currently selected.

Chan has continued to build on this work in a regular series of papers that investigate variations on the ART technique. While all of these are interesting, none demonstrated a significant improvement on the basic scheme. Furthermore, while the concept of anti-random testing is of itself interesting, new work in the field seems to have declined drastically. This has possibly come about because of a number of factors, including:

- issues highlighted by Yin *et al.* [343] that concern the difficulty in selecting effective encoding of the data;
- results from Mayrhauser and Bai [311] and Yin [342] that suggest that the technique may not always be competitive with random testing for small test sets.

2.3.3 Mathematically Inspired Techniques: Boundary Following

Boundary value testing as defined by Myers [238] is an extension of equivalence partitioning. The primary assumption with equivalence partitioning is that any value within

a partition can be used for a test and that any value is good as any other. Boundary value testing however requires that values be selected at the edges or boundaries of a partition rather than anywhere within the partition.

Both Myers [238] and Howden [172] provide a similar set of "rules" for selecting values including:

- variables that have a range of values, select the values for the end of the ranges, invalid cases and the interior of the range,
- variables that have a number of values (e.g. arrays), test the values at the ends and one in from the end.

In addition, both authors recommend that the rules be applied to out-of-range values in the same way. Howden extends this to include testing of all values for variables with discrete values (e.g. enumerations in C) and to include the value zero for ranges which include it.

Hoffman has been involved in a series of papers that used these principles for automated test case generation. The basic framework was laid out by Hoffman and Breakley [163] in a paper that introduced some of the key concepts. In particular the technique of representing the test cases as a set of n-tuples is presented as a core idea along with the concept of a generator (iterator) for constructing valid n-tuples.

Hoffman and Strooper [165] extended this work by Hoffman and Breakley [163] and investigated in more depth how automatic generation of data can be reconciled with the oracle problem by giving examples of functional testing, trace invariants and large scale random testing.

Hoffman, Strooper and White [164] have attempted to codify these rules as explicit definitions and have provide *formal* definitions of what constitutes a boundary. These definitions comprise two major variants as follows: k-bdy(D), the k^{th} boundary of domain D and k-per(D) which is the k^{th} perimeter of the domain D. In addition, two variants of these are defined, k*-bdy(D) and k*-per(D) which informally[11] comprise the union of the boundaries or perimeters for $i = 1..k$.

[11] In the interests of brevity complete details are omitted, see [164] for the mathematical definitions.

For example given two variables x and y with domains defined as $D(x) = [0, 5]$ and $D(y) = [0,6]$ then for the single domains the boundary sets are shown in Table 3.

k	k-bdy(x)	k-bdy(y)	k*-bdy(x)	k*-bdy(y)
1	{0, 5}	{0, 6}	{0, 5}	{0, 6}
2	{1, 4}	{1, 5}	{0, 1, 4, 5}	{0, 1, 5, 6}
3	{2, 3}	{2, 4}	{0 ,1, 2, 3, 4, 5}	{0, 1, 2, 4, 5, 6}
4	{2, 3}	{3}	{0 ,1, 2, 3, 4, 5}	{0, 1, 2, 3, 4, 5, 6}

Table 3. An example of boundary sets over the domains D(x) = [0, 5] and D(y) = [0,6].

These concepts can be extended to a boundary set for multiple domains as stated above by taking the Cartesian product of k-bdy(D) for all domains and for k*-bdy(D) by forming the union of the sets.

Boundary types can be represented in a matrix (after Hoffman *et al.* [164]) as shown in Figure 2. Figure 2(a) shows the points selected for 1-bdy(x, y) and Figure 2(b) shows the arrangement for 2-bdy(x, y). In Figure 2(c) the union of these sets, 2*-bdy(x, y) is given.

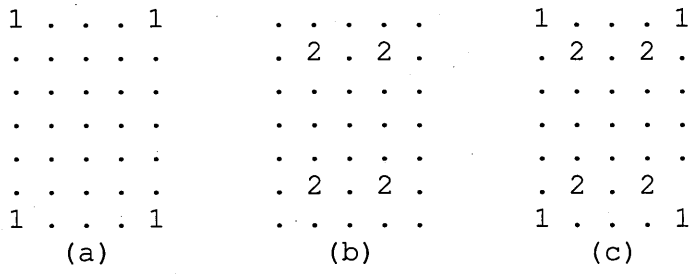

Fig. 2. Boundary values as defined by Hoffman *et al.* [164] in diagrams (a) to (c).

The perimeter relation k-per(D) can be displayed in the same manner and 1-per(x,y) and 2-per(x,y) are shown in Figure 3(a) and Figure 3(b) respectively.

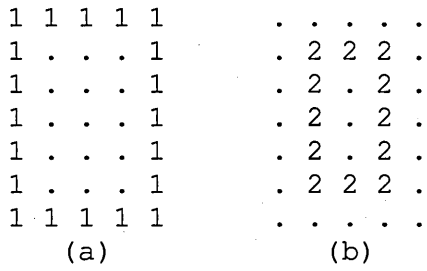

Fig. 3. Perimeter sets as defined by Hoffman *et al.* [164] for the domains defined above.

It can be seen that the lower order ($k \leq 2$) definitions for boundaries do not match the traditional definitions of boundary value testing, whereas the definition of the perimeter exceeds the normal requirements by including all values on the edges.

An indication of what effect this had on coverage can be obtained from examining the major empirical results presented in Hoffman, Strooper and White [164] for a sort function. These results indicate that 1-bdy and 1*-bdy do not provide adequate statement coverage whereas 2*-bdy and the lower order k-per versions did. In both of these cases, however, there was a better approximation to the traditional definitions of boundary value testing. The better match provided by both the perimeter variants also appeared to translate into a better fault detection ability, with the majority of the lower-order perimeter variants detecting all the seeded faults.

Daley, Hoffman and Strooper [94] extended the work presented in Hoffman, Strooper and White [164] in a number of ways. First they applied it to testing Java classes. Second, they extended the test generation framework to allow more automation. Third and most significantly, they investigated the concept of chaining domain dependencies. Dependent domains are quite common in practice. For example given an array, the valid values of the index are dependant of the size of the array. The example used in [94] is a windowing manager for a spreadsheet-like application. The dependent data here was the window position on the screen, the position being dependent on the shape of the table and the cursor position. Unfortunately, the process of defining and generating data based on dependent domains is not fully automated and requires code to be generated manually.

Given the results from Reid [269], [270] and the conventional wisdom as stated by Beizer [31] that errors "*hide in the corners*" (pg.198), the domain following technique seems quite promising. However more empirical work is required to confirm this and this does not seem to be forthcoming. There is however one weakness in the technique: an

effective oracle is required to allow running the number of test cases a technique such as 2*-per would create.

2.3.4 Mathematically Inspired Techniques: Combinatorial

This section is a summary; chapter 3 contains a detailed review of the same subject matter.

2.3.4.1 Combinatorial Techniques

In at least one sense most techniques that are used to generate test data can be considered combinatorial techniques because combinatorics involves selecting or arranging a set of "objects" from some finite set of objects such as the set of valid integers and Cameron [52] defines it as "*the art of arranging objects according to specified rules*" (pg. 2).

The two most widely known combinatorial structures are combinations and permutations. That is, an unordered set of r objects from a set of n objects and the number of ways of ordering n different objects.

Two examples illustrate the limits of what constitute combinatorial. The simplest case of a combinatorial technique is one where, given a set of inputs a single value is selected for each input. This produces exactly one test case and, is in general, of little utility although for straight-line code it would be statement and branch adequate. The most complex example of a combinatorial technique is one in which all values for all variables are used.

While the first case is of little practical value because of its potential weakness in detecting faults or obtaining coverage, the second suffers from the fact that there are too many test cases and the amount of time required to both run and examine the results is excessive, if even possible.

In between these two extremes, there are several techniques for generating sets of test vectors that are potentially more useful. These techniques provide a means of selecting data from a set that maximises the probability that interactions between variables will be tested and that results in a set of vectors, that although large, is capable of being executed in a reasonable time period. A large amount of this work examines the construction of test sets in which all pairs, triples, etc. of values for all variables taken n at a time are generated, as well as the effectiveness of those test sets.

The motivation for using test sets constructed in using these techniques is derived from the way that statistical experiments in various fields are conducted to maximise the amount of information obtained for the minimum amount of effort. The general topic is called design of experiments and the specific example used in research is termed factorial experiments and is covered widely in areas such as industrial quality control and engineering (Diamond [101]), psychology (Keppel and Saufley [189]) and biology (Fowler and Cohen [120]) even at an undergraduate level.

The combinatorial structure that forms the usual starting point for discussion in factorial experiments is the Latin square. This is an N by N matrix that has the property that the values from 1 .. N inclusive appear in each row or column exactly once. An example is shown in Figure 4, which represents the interaction of *three* variables each taking four values. The values of the first variable being the row indices, the values of the second variable the column indices and the values of the third variable are the values in the set {A, B, C, D}.

variable 1 \ variable 2				
	A	B	C	D
	B	C	D	A
	C	D	A	B
	D	A	B	C

Fig. 4. An example of a Latin square.

Thus, we have a set of sixteen test vectors v_1 .. v_{16} that can be read from the matrix as follows:

$$v_1 = \{1, 1, A\} \quad v_2 = \{1, 2, B\} \quad v_3 = \{1, 3, C\} \ \ldots \ v_{16} = \{4, 4, C\}$$

Note that in the mathematical literature, the variables under consideration are normally referred to as factors and the different values that each factor can take on are referred to as levels rather than values. However, in this work, the more usual terms used in programming, i.e., variables and values will be retained.

To deal with more values we need to create a larger Latin square. However, it should be noted that there is no Latin square for $N = 6$ [12] and it is not uncommon for a Latin square for a particular value not to exist. To deal with more variables we need to form a Greco-Latin square[13] using two orthogonal Latin Squares[14] as shown in Figure 5.

A	B	C	D
B	C	D	A
C	D	A	B
D	A	B	C

Square 1

4	3	2	1
3	2	1	4
2	1	4	3
1	4	3	2

Square 2

A 4	B, 3	C, 2	D, 1
B, 3	C, 2	D, 1	A, 4
C, 2	D, 1	A, 4	B, 3
D, 1	A, 4	B, 3	C, 2

Resultant

Fig. 5. Combining Latin squares to cover a fourth variable.

Clearly, this process can become more difficult as the number of variables or the values increases. For example, consider the situation where the variables have an uneven number of values. If one variable has nine values we deem "of interest," then even if all the other variables have only three or four values of interest we are forced to use a large array to deal with just one variable.

A solution to this sort of problem is to remove the requirement for using a *balanced* design. A Latin square is balanced as all the values are used the same number of times. This leads to the concept of a covering array (CA). Informally, a covering array is a set of vectors where the set as a whole is guaranteed to meet some covering property, often that all pair-wise (2-way) interactions between values of all variables are present.

A pair-wise (2-way) adequate test set is one where all 2-way interactions between n input variables v_1 to v_n will be covered. In the test set there will be a vector such that for every value that the variable v_i is allowed to take it will be paired with each value the variable v_j is allowed to take for all i and j, where $i \neq j$.

---

[12] The problem for N= 6 is originally proposed by Euler.

[13] Cameron [52] explains the terminology derives from using Latin characters for the first square and Greek for the second orthogonal square.

[14] Two Latin squares $A = (a_{ij})$ and $B = (b_{ij})$ are orthogonal if there exists unique values i and j such that $a_{ij} \neq b_{ij}$ $\forall i,j$. For further details see Cameron [52].

An important consideration is which values each variable will be allowed to take on. In general the tester will select data points for each input variable that are of "interest" based on criteria such as data input ranges, domain partitioning and other heuristic rules. Selection of all values is impossible except where only a small number of values are allowed such as for enumerations.

To make this more concrete consider a function with three input variables, v_1, v_2 and v_3 that take on the values a_1, a_2, a_3 and b_1, b_2 and c_1, c_2 respectively. Then a 2-way adequate test set that ensures that a vector exits that contains all values of v_1 paired with all values of v_2 and all values of v_3 and all pairs of v_2 and v_3. A set of seven test vectors for this example is shown in Figure 6.

1:	a_1	b_2	c_1
2:	a_2	b_1	c_2
3:	a_3	b_1	c_1
4:	a_2	b_2	c_1
5:	a_1	b_2	c_2
6:	a_3	b_2	c_2
7:	a_1	b_1	c_1

Fig. 6. An example seven vector, 2-way adequate test set for 3 variables.

Larger values of t can be used, for example $t = 3$ would involve matching all sets of three variables and $t = 4$, four variables in the same way.

It should also be noted that this terminology in this area is not yet fixed. Some work such as Cohen *et al.* [67] refers to covering arrays that meet the pairwise criteria using that term, the term t-way is also used for example by Lei *et al.* [211], [210] and as the number of factors increase above $t = 2$ (pairwise) that terminology is becoming more common. Thus a pairwise covering array can also be said to be a 2-way covering array. For higher order covering arrays the terms or n-way or t-way or t-wise can be used as can the term t-covering, as in 3-way or 3-covering. In addition, some authors also refer to the strength of a covering array. For example a CA of strength $t = 3$ is a 3-way or a 3-wise CA.

The term "design" is also used when talking about covering arrays. For example, an orthogonal array (e.g. a Latin square) can be described as a balanced experimental design and an unbalanced design is also referred to as an incomplete design.

Formally, (after Cohen and Colbourn [76]) an orthogonal array $OA_\lambda(N; t, k, v)$ is a N x k array on v symbols such that every sub-array contains ordered subsets of size t from v symbols exactly λ times. Here, N is the number of rows, t is the "strength" of the array (e.g. $t = 2$ is pairwise), k is the number of parameters and v is the number of values of each parameter. Normally, λ is taken to be one and the subscript is dropped. When $N = v^t$ the OA is optimal.

A covering array $CA(N; t, k, v)$ is a N x k array on v symbols such that every N x t sub-array contains all ordered subsets from v symbols of at least size t.

The other object that needs to be defined here is the mixed level covering array because in the context of testing this is the most interesting case. A mixed level array has a variable number of values for each parameter and is denoted as $MCA(N; t, k, (v_1, v_2 \ldots v_k))$ where $v = \sum^k v_i$ which can also be written as $MCA(N; t, (w_1^{r1}, w_2^{r2} \ldots w_s^{rs}))$. For example $MCA(N; 2, (5^1, 3^8, 2^2))$ or more usually as $MCA(N; 2, 5^1, 3^8, 2^2)$, has a strength of 2 (2-way) and has one parameter (r1) with five values, eight parameters (r2) with three values and two parameters (r3) with two values.

2.3.4.2 t-way Test Set Generation

The original work with 2-way adequate test sets by Mandl [220] was derived directly from the design of experiments field (e.g. Diamond [101]) and used the same techniques to construct the test sets based on Latin and Greco-Latin squares. Williams and Probert [331] used Galois Fields to construct larger arrays that potentially address some shortcomings of Greco-Latin squares. Stevens and Mendelsohn [291] investigated the use of an existing set of covering arrays as a basis for constructing larger arrays in which gaps were filled using simulated annealing; a similar approach was taken by Williams [330] for MCA's of strength 2. Daich [89], [90] implemented a tool based on similar ideas that used a spreadsheet that also removes redundant tests where possible. Cohen *et al.* [76] performed the most recent work. In this work, researchers stitched together smaller sub-arrays created by other techniques such as simulated annealing and/or constructive algorithms such as the greedy heuristic methods from Cohen *et al.* [67] with good results.

However, although the use of orthogonal arrays is an active research area in mathematics it has received less attention than other techniques for generating tests. The majority of the effort has been focused on using greedy heuristics or techniques based on large search spaces.

The original work with greedy algorithms was undertaken by Sherwood and documented in two ATT Technical reports by Sherwood [281], [282][15] and in a later publication [280]. This work was developed further in the Automatic Efficient Test Generator (AETG) by Cohen *et al.* [68], [70], [66], [67]. The AETG tool extended the capabilities of the higher factor *t*-way adequate tests. A similar test generator for 2-way tests was proposed by Tung and Aldiwan [305] and used in Smith *et al.* [287], [286] in testing spacecraft navigation software. Later Colbourn, Cohen and Turban [79] presented a deterministic algorithm for generating 2-way adequate tests using similar principles to AETG and work by Bryce, Colbourn and Cohen [46] provides a framework for encapsulating all of these techniques within a single structure.

A different approach for generating 2-way adequate test sets was taken by Lei and Tai [212], [292] who generated an initial set of vectors adequate for the first two variables and extended this initial set for each subsequent variable. This work was extended to *t*-way adequate test sets as reported in Lei *et al.* [211], [210] and applications of using the approach reported in Kuhn and Okun [201] and Kuhn *et al.* [204].

The simplest variation of a *t*-way adequate generation scheme – the "base choice" method was formally identified by Ammann and Offutt [9] although it has probably been used if not defined previously. This method has also been used by Cohen *et al.* [69] and re-invented by Xu *et al.* [339]. The "base-choice" method is a single factor experiment i.e. $t = 1$ where variables are changed one at a time for each of their selected values.

Metaheuristic search techniques have also been applied to the problem of generating covering arrays and *t*-way adequate test sets. As noted above, Stevens and Mendelsohn [291] used simulated annealing in conjunction with existing orthogonal or covering arrays. Investigations of metaheuristic techniques for generating covering arrays by

[15] These can be found at http://testcover.com/pub/background/cats1.htm and http://testcover.com/pub/background/cats2.htm as of 22 April 2005.

mathematicians have found simulated annealing to be an effective technique, work having been done by both Nurmela and Ostergard [245] and Stardom [290]. Stardom also investigated Tabu search and genetic algorithms. A large amount of work on generating covering arrays for testing has been performed by Cohen *et al.* [76], [77], [72] who also investigated the generation of variable strength covering arrays, i.e., those which are *t*-way adequate for some subset of the variables and *(t-n)*-way adequate for all variables

Several researchers have investigated the use of genetic algorithms for producing covering arrays. Early work by Ashlock [21] reported disappointing results on larger systems. Similarly, Stardom [290] also reported that genetic algorithms performed worse than either simulated annealing or Tabu search. Shiba, Tsuchiya and Kikuno [284] investigated the application of genetic algorithms and ant colony optimization (Dorigo and Gambardella [105]) to the problem and found that the size of test sets generated by these methods were comparable with test sets generated using simulated annealing by Cohen *et al.* [76], with IPO by Lei and Tai [212] and with AETG from Cohen *et al.* [67].

Other techniques have been applied to the problem of *t*-way test set generation. For example Williams and Probert [332] developed a formal framework for thinking about various types of interaction coverage included *t*-wise coverage criteria. Building on this framework, Williams [333] reformulated the task as a {0, 1} integer programming problem. However, the reported results were disappointing even for small systems and the method was deemed to be impractical because of the time required to solve the problems and because of resource consumption. Kobayashi *et al.* [196] proposed yet another method for generating 2-way adequate test sets.

Hnich, Prestwich and Selensky [160] used a SAT[16] formulation of the problem and constraint programming to find provably minimal covering sets for a small number of problems with Boolean values and improved on their results in [161]. However, these researchers also concluded that their technique may only be useful for problems up to a certain size. Building on the work by Hnich *et al.*, Yan and Zhang [340] applied special purpose SAT solvers utilizing exhaustive backtracking search. As with Hnich *et al.*, they concluded that the time complexity might limit the use of their technique. It will be

[16] SAT is shorthand for the satisfaction problem, the first problem to be formally proved as NP-complete.

interesting to see if any more attempts are made to purse this avenue of research as a direct means of generating covering arrays.

The final technique that needs to be mentioned here are random designs. While these strictly speaking are not necessarily *t*-way adequate (depending on their construction method), they have been used with some success by Dunietz *et al.* [106] and Schroeder *et al.* [277], who concluded that for the same size test sets random designs were as effective as *t*-way adequate designs of the same size. The one main advantage of random designs is that their construction complexity is much lower.

The work cited above concentrates on finding methods to generate the smallest possible covering array and largely ignores some of the problems that can possibly occur with real systems such as constraints on allowed variable values in a vector. A list of issues that can occur is given in Czerwonka [87] where a *t*-way test generation tool developed at Microsoft is described. These issues include:

- seeding – where the vector set is initially seeded with a set of vectors defined elsewhere;
- mixed strength covering arrays – mentioned by Cohen *et al.* [67] where seeding is a suggested implementation mechanism and by Cohen *et al.* [77] who build mixed levels into the generation process;
- constraints or exclusions – that is pairs, triples etc. of parameter values that are not valid within a single vector. Sherwood [280] suggested using disjoint subsets of the input model to deal with this issue.
- negative or error values – again mentioned by several authors, including Cohen *et al.* [67] but not covered directly by any generation tool;
- adding weighs to parameter values - a method that may allow the generated covering array to be biased to better coverage of selected values.

The addition of weights to variables has been investigated in several studies led by Colbourn and Bryce. In [78] a theoretical study was undertaken to demonstrate how the use of assigned weights can be used to force evaluation of pairs of dynamically selected web services and to prioritise testing based on how "trusted" a web site was and to delay testing of less probable pairings. In [43] and [44] weights were used to prioritise the generation of vectors that covered as many new *t*-way tuples as possible. The reasoning

behind this approach being that for covering arrays with high factors ($t > 3$) it may not be possible to execute all tests and, given work by Kuhn *et al.* [313]. [202], [203] that shows factors greater than three are often required, it is *assumed* that preference should be given to the vectors that cover the most interactions. In [44] it is also shown that weights can be used to limit, but not eliminate, the number of invalid value combinations that can occur in a vectors. That is, adding weights can implement a weak form constraint.

This work was continued in Bryce *et al.* [45] where a hybridised vector generation technique is investigated. The technique used a greedy algorithm such as AETG to find an initial vector and then applied a search technique such as simulated annealing or Tabu search to improve the vector.

The issue of dealing with constraints fully was tackled in a series of papers by Cohen *et al.* [74], [73], [75] using a hybrid approach where a greedy algorithm (AETG) is used to produce candidate vectors. This is supplemented by a SAT based constraint solving system that checks that the value to be added to a vector is valid and violates no constraints. This seems to be a remarkably elegant and efficient solution to the problem. The use of the SAT solver allows the constraints to be specified as Boolean expressions in conjunctive normal form and, despite the extra work required to check for validity, the approach results in both a saving in time and in slightly smaller sets of test vectors.

Equally importantly, this series of papers justified the need to be able to be able to accommodate constraints in real work test situations by examining the options available for several large, configurable sets of software. These included the SPIN model checker, the GCC optimiser, the Apache HTTP server and Bugzilla. The papers showed that on average the AETG algorithm with constraint handling produces system test vectors where only 3% of the vectors contained no constraint violations.

2.3.4.3 Field studies

Field studies fall into a number of classes. For instance, there are observational studies such as those conducted by Dalal *et al.* [93], [92] and Bell and Vouk [32] that provide some evidence for the effectiveness of the technique and that are useful because they make interesting observations on issues involved in using the techniques in practice. There are other similar reports on the use of the techniques which provide some details on what was

done but that do not supply the details that are needed to allow a direct comparison with other techniques, for example Perkinson [263], or only estimates the improved effectiveness of the technique as in Burroughs *et al.* [51] and Huller [173].

There is also a body of work that provides better evidence for the techniques effectiveness in practice, including various papers by Cohen *et al.* [68], [70], [67] that contain partial results from various studies and that seem to be strongly related to work reported in Cohen *et al.* [69].

Work with more detail, i.e., work that provides measured data not just estimates is supplied by Burr and Young [50] for code coverage and by Smith *et al.* [287], [286] in relation to faults discovered by applying different test generation strategies. Whereas the first of these studies demonstrates reasonable code coverage, the second indicated that 2-way testing at least is not always effective.

The final set of papers discussed in this section is a set of experiments in which the set of faults being studied is not known, i.e., incompletely controlled experiments. Pan, Koopman and Siewiorek [258] applied complete testing on a set of Unix commands (all option combinations). Although this is strictly outside the area studied, this work demonstrates that for some problems such testing is possible. In addition their approach to the oracle problem is of interest. Yilmaz, Cohen and Porter [341] also apply large scale testing to program options and compared complete testing and t-way adequate testing on a much larger problem. Interestingly they worked backwards to determine the effectiveness of t-way test techniques. The final set of papers considered in this sections is a group of studies by Wallace and Kuhn [313], Kuhn and Reilly [202], Kuhn, Wallace, and Gallo [203]. This series of papers examined faults discovered and then determined the number of variable interactions required to expose the faults. The major conclusions of these final studies was that only small factors, i.e., $t <= 6$ are required to expose faults.

2.3.4.4 Empirical Studies

Given that combinatorial testing using covering arrays has been in use for some time, there are surprisingly few controlled experimental studies that have been published. There are two possible reasons for this. First that much of the early work was dominated by researchers at Bellcore (Bell Laboratories, now Telcordia Technologies). Secondly because

much of the early work has been focused on developing algorithms for finding *t*-way adequate test sets.

Seven major experimental studies deal directly with the detection of coding errors as follows:

- Cohen *et al.* [69] who performed several coverage experiments;
- Dunietz, Mallows and Iannino [106] who also addressed code coverage;
- Nair, Ehrlich and Zevallos [239], Schroeder, Bolaki and Gopu [277], Grindal *et al.* [138] and Kuhn and Okun [201] which addressed the techniques effectiveness at detecting seeded faults;
- Kobayashi, Tsuchiya, Kikuno [197] who examined the techniques ability to distinguish logic mutants.

A few other empirical results have also reported. Arguably Mandl's seminal paper [220] that introduced the use of design of experiments techniques to compiler testing with all 2-way interactions using Latin and Greco-Latin squares was based on empirical work. Unfortunately presents no results or comparisons.

Cohen *et al.* [70] present a small set of empirical results for block coverage on the UNIX sort command taken from [69] and compared their results 86% to 95% block coverage with a study by Wong *et al.* [334] on the same command (73% block coverage with random test sets). A subset of the results are reused in Cohen *et al.* [67].

Finally, two experiments by Hoskins *et al.* [166], [167] that compared the MCA and D-optimal designs to approximate full factorial designs.

2.3.4.5 Assessment

The large bulk of papers produced on *t*-way adequate testing are on methods for generating test sets, with a strong emphasis on making those test sets smaller. However, there are two issues with this, first much of the work has been done with low factors, e.g., $t = 2$ or $t = 3$. Second the work by Kuhn, Wallace, Reilly and Gallo [313], [202], [203] strongly suggested that factors of five or six are in general required to assure that all interactions are covered. In general, large factors will lead to large sizes for test sets. Whether saving a few or even a few dozen tests makes any difference in this situation is possibly a non-issue.

The other feature of research into generating test sets is the number of techniques that have been used. However, despite this algorithmic methods based on the AETG algorithm remain dominant, possibly because these seem to be able to be readily adapted to new area of interest, e.g. work on weighting by Colbourn and Bryce [78], [43] and on constraints by Cohen *et al.* [74], [73], [75].

Another unusual feature of work on evaluating the techniques is the early dominance of field studies which although certainly increasing interest in using *t*-way adequate test sets. However, it does not actually appear to have advanced the understanding of why the techniques are effective or indeed, how effective they are relative to other techniques, the amount of comparative information available being low, especially in early work.

The final point to make is the relative lack of controlled experimental work that has been performed using *t*-way adequate test sets and particularly the amount of work that has made comparisons with other techniques.

2.3.5 Mathematically Inspired Techniques: Summary

This section has discussed three techniques that have been loosely grouped together under the banner of being based directly on some mathematical principle.

In this section, what really stands out is that only one of the techniques has achieved wide acceptance in both the academic community and in actual use as a practical technique.

The anti-random testing technique proposed by Malaiya [219] appears to have fallen out of favour with researchers and to have been shown to be less effective than *t*-way adequate test sets by Kobayashi, Tsuchiya, Kikuno [197]. One possible reason is that the technique does not seem to offer any real advances over random testing given results from Mayrhauser and Bai [311] and Yin [342]. Little other comparative work seems to have been performed.

The boundary following techniques proposed by Hoffman *et al.* [163], [165], [164] covered in section 2.3.3 likewise suffers from a lack of interest in it as a technique for automatically generating tests despite it being based on a strong idea, that the boundaries are where the bugs tend to congregate (Beizer [31] pg.198). Some work in this area has been performed by researchers using adaptive testing techniques, for example by Jones *et*

al. [183] who investigated boundary value testing and by Gallagher and Narasimhan [128]. The boundary definitions presented in Hoffman *et al.* [164] however seem to have vanished as an area of active research. Why this is so is unclear, however we can surmise two possible reasons. First, for any sufficiently complex problem, automatically locating and defining boundaries is far from simple and Beizer [31] devotes an entire chapter of his book to the topic. Second, the *t*-way adequate combinatorial techniques discussed in section 2.3.4 allow the inclusion of boundary values in a more straightforward manner for corner points coming closer to the definitions proposed by Myers [238] and Howden [172].

Of the techniques discussed here, *t*-way adequate techniques have received by far the most attention. Why this is so is not clear – however its use in actual testing of systems and its reported advantages may be one factor in its adoptions. Another reason for the amount of work being performed is that it seems to fit in a natural manner with existing, well established concepts, namely equivalence partitioning and Ostrand and Balcer's [256] category-partitioning technique.

One final point should be noted, as with random testing, all the techniques presented in this potentially suffer from the same problem, the size of the test sets, which could potentially number in the thousands. This of course makes having a solution to the oracle problem imperative if the techniques are to be used on a large scale. However, apart from the use of formal models as suggested by Kuhn *et al.* [204] no solution to this problem has been suggested in the literature.

Figure 7 shows the chronologic sequence of the major items of work discussed in this section and shows some of the connections between the major themes explored (dotted lines).

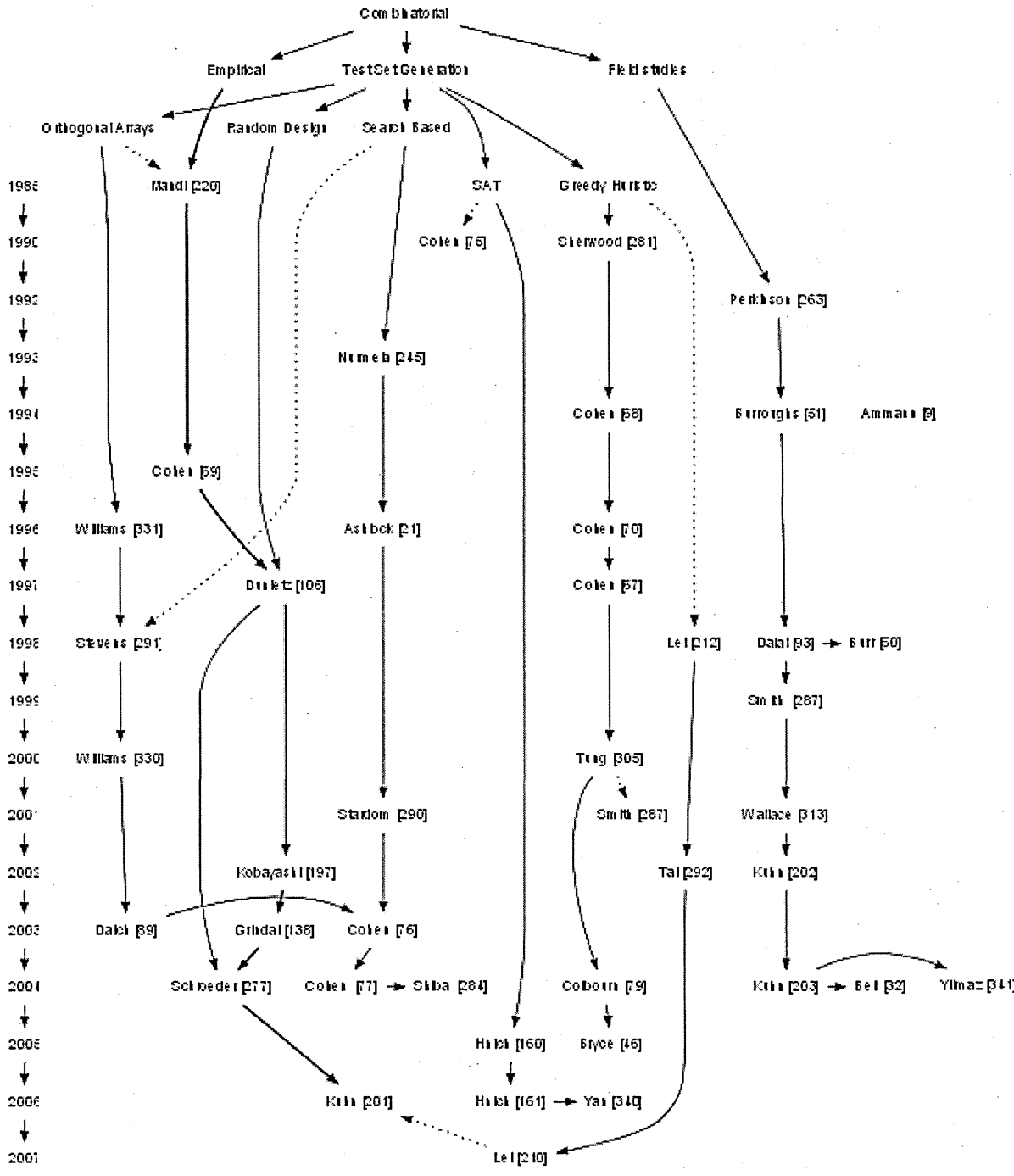

Fig. 7. Chronological order of major work examined under combinatorial testing, organized by area of activity and date of publication.

2.4 Adaptive Testing

2.4.1 Adaptive Testing: Introduction

Adaptive testing covers a number of related techniques where information is gathered from the execution of a program with a known test vector or vectors. The information gathered is then used to derive new test vectors in an iterative process. The information gathered during program execution can take on multiple forms depending on the type of test activity being performed. For example, it can include information based on statement, branch or path coverage (white box testing) or alternatively, it can search for data in an attempt to violate some expected property that the code is expected to have (functional or black box testing). The process of adaptive testing is thus a directed search (Clarke *et al.* [62]) of the input space for data that meet the testing criteria. The search being directed via the evaluation function, which is used to measure numerically the "goodness" for each individual vector. The numerical nature of the "goodness" evaluation allows the search to be driven as a minimisation or maximisation problem.

The process is started by defining an initial test vector t_i and using this as the input to program P to find a value for the evaluation criteria or "fitness function" $F(t_i)$. The information contained in t_i, P, and $F(t_i)$ is then used by the adaptive test generator to adapt or modify the initial test vector t_i to produce a new test vector t_{i+1}. This process is iterated using the accumulated information until either the value of $F(t_{i+n})$ is minimised or the process is otherwise halted. This can occur either after a fixed number of steps or when some other halting criteria is satisfied.

Adaptive test generation has been employed primarily in two different modes in the literature. The first mode utilises a fitness function defined externally to the program under test and is exemplified by the early work of Cooper [80] on system performance. Other examples in this area include attempts to drive systems into error states for robustness testing by Schultz *et al.* [278], [279], Wegener and Bühler [319] and experiments performed by Grochtmann *et al.* [141] to establish worst case timing for real time software.

The second mode in which adaptive techniques have been used is one in which the structure of the program itself provides the function to be minimised. Most of this work is aimed at meeting white-box testing criteria such as branch and path coverage. For

example, Miller and Spooner [235] based their minimisation function on the conditions required to traverse a particular path. Similar work is undertaken by Korel [199], Jones *et al.* [184], Tracey *et al.* [301] and Wegener *et al.* [318].

A variation on this theme is work where rather than monitor control flow other properties of the code are targeted. For example, Andrews and Benson [10] investigated assertion violations, Jones *et al.* [183] applied adaptive testing to boundary value testing and Tracey *et al.* [302] used it to try and trigger run time exceptions.

2.4.2 Adaptive Testing: Early Work, Setting the Foundations

The initial work on search based testing using external fitness functions appears to have been performed by Cooper [80]. This author developed an adaptive test system designed to maximise the stress on a system under test in order to empirically determine whether that system met performance goals such as timing and to establish the sensitivity of the system under test. A number of search techniques were proposed by Cooper to aid the selection of the next test including gradient descent, probabilistic search [180] and a "heuristic" search using a database of transformation rules to aid selection of the next test.

Miller and Spooner [235] put forward the idea of dealing with the test generation problem as a numerical maximisation problem by treating paths in a program as straight line programs and dealing with predicates as numerical constraints. However, their work only dealt with variables with a floating point representation, requiring integer constraints to be determined manually.

Andrews and Benson [10] implemented a system that searched for test data that violated assertions placed in code by using both the complex method [271] for constrained optimisation and a systematic grid search of the input space. Results reported from the paper suggest that the technique was effective at finding seeded errors and that it detected some previously unknown actual errors in the subject code associated with boundary conditions. As expected, the optimisation approach required fewer tests than the grid search. However, the optimisation based search missed four assertion violations that were located using the grid search based on stepping through the input space at regular intervals. Reasons for this discrepancy were not given but presumably it was because of the brute

force nature of the grid search did not become trapped in local minima (or maxima) as observed in latter work (e.g. Michael *et al.* [232]).

The basic approach used by Miller and Spooner [235] seems to have been reinvented by Korel [199] who replaced straight line programs by tracing specified paths though the control graph representation of the program. In Korel's work, each branch predicate was a goal node and input data sets that take the desired branch were identified by treating the predicate as a constrained optimisation problem and were solved using a direct search method of alternating variables where the function was minimised with respect to one variable at a time. The technique therefore used two distinct types of moves: exploratory searches and pattern searches. An exploratory search perturbs the subject variables to identify the direction which results in an improvement. A pattern search forces a series of moves in the direction selected (Glass and Cooper [130]). The predicates dealt with were limited to simple relational operators and the input space to integer variables. However, an interesting aspect of this work is the use of backtracking to derive data for structures involving pointer references.

Ferguson and Korel [118] extended Korel's [199] work by replacing the concept of following a specified path with the concept of aiming to reach specific goal nodes irrespective of the path taken. They also introduced the idea of "chaining" subgoals dynamically, an approach in which each sub-goal is identified as having to be executed before the target goal. This effectively produced a dynamic data dependency graph for the program. An advantage of this chaining approach is that it is simpler to backtrack and to attempt to find more advantageous paths to the end goal. Importantly, the chaining approach mirrors what a human tester would do in circumstances where it was proving difficult to achieve coverage of a node. Experimental results from 11 small programs that compared random, path oriented, goal oriented (Korel [199]), and chaining approaches suggest some advantage for more complex code examples. The chaining approach achieved coverage greater than 10% greater than any other technique in 2 of the 11 subjects and more marginal improvements in two further cases[17].

[17] Statistical significance was not investigated.

The construction of a chain is best explained by a simple example (from McMinn and Holcombe [227]). A chain is a sequence of events such that $e_i = (n_i, C)$ where n_i is the node and C is the set of variables that form the constraint set of variables that must not be modified until the next event in the sequence. Given the simple program in Figure 8 and assuming f as the target node and e as the problem node then there are two sequences defined as follows;

(1) $< (a, \{flag\}), (e, \varnothing), (f, \varnothing) >$

(2) $<(d, \{flag\}), (e, \varnothing), (f, \varnothing))>$

where (1) is the sequence where flag is not redefined at node d and (2) is where it is redefined at d.

```
void loop_assignment (int a [10])
{
    int i;
    int flag = 1;                  /* a */
    for (i =0; i < 10, i++)        /* b */
        if (a[i] != 0)             /* c */
            flag = 0;              /* d */
    if (flag)                      /* e */
        /* target node */          /* f */
}
```

Fig. 8. A simple example of how chaining is applied from a target node f with the predicate at node e.

Watkins [317] provided an early example of the application of genetic algorithms to the generation of test data. The fitness function applied was based on path traversal, with the most often traversed paths having the lowest fitness.

Roper [272] also investigated the use of genetic algorithms to generate branch adequate test data. This work was preliminary and used a simple fitness function based on control branches reached. Nevertheless, it suggested two avenues of approach that were taken up by other investigators: the use of a separate store for good individuals e.g. Michael *et al.* [232] and the use of human generated data from functional tests as the initial test vectors by Wegener *et al.* [321].

While the majority of the work to date has been used to locate test data to meet code coverage criteria, other avenues have also been looked at. For example, Andrews and Benson [10] used assertion violations and Tracey *et al.* [299], [301] also examined this

avenue with some success and extended the work to search for exception violations [302]. Jones *et al.* [183] applied adaptive testing to boundary value testing.

This then forms the foundations for most latter work where for the most part metaheuristic search methods such as genetic algorithms have replaced optimisation techniques such as those used by Cooper [80], Miller and Spooner [235] and Korel [199]. Latter work is heavily focused on overcoming some of the problems identified in the early work (section 2.4.2.1) and on looking for better fitness functions (section 2.4.3.1) and search techniques (section 2.4.3.2).

Figure 9 shows the chronology of the major works discussed in the previous section and this organised about the major uses to which the technique has been used. For clarity, work associated with timing has been shown using a dashed line. Other themes such as the fitness functions used for what box testing and search methods used in the work are described in Table 4 and Table 5.

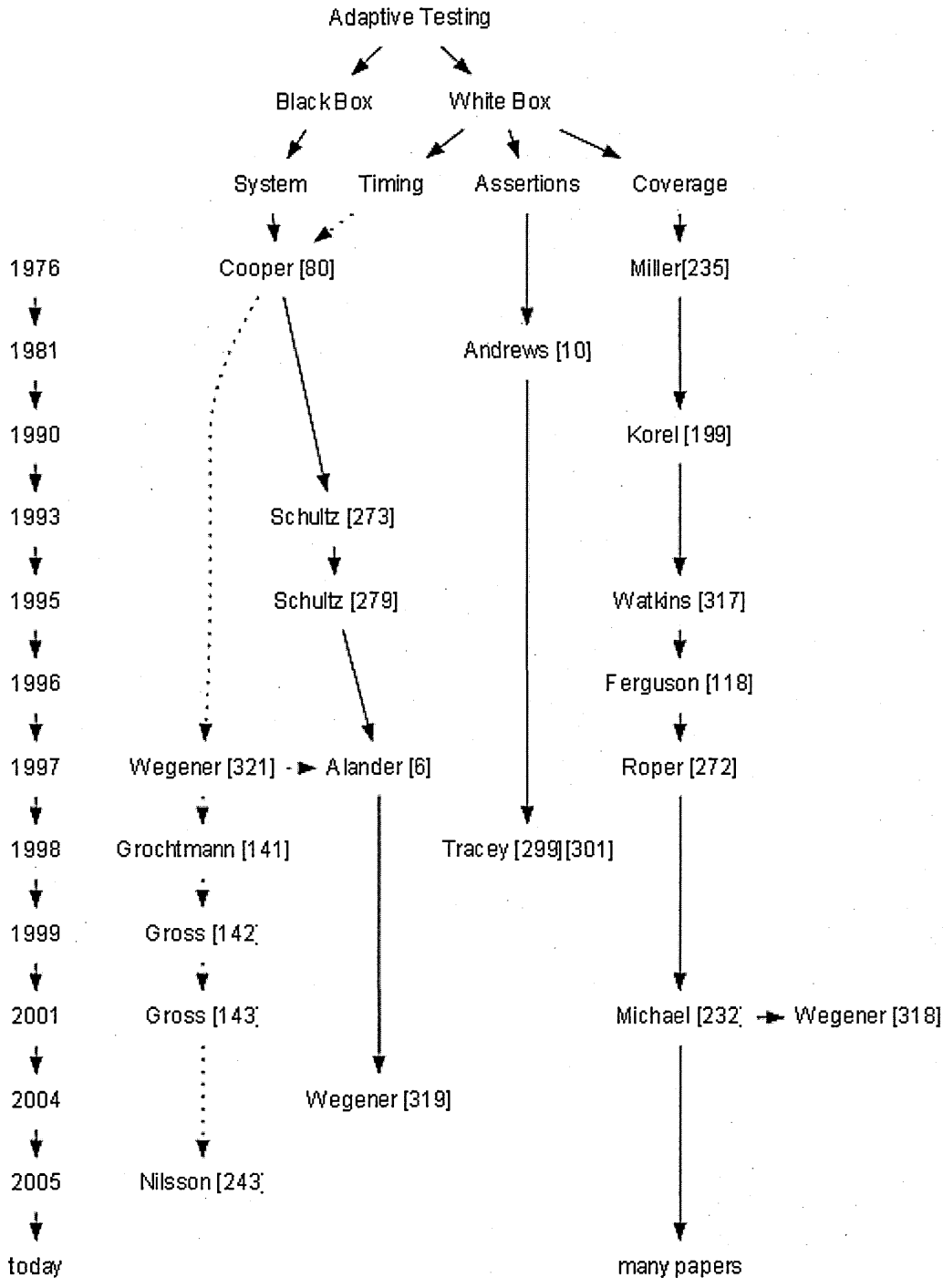

Fig. 9. Chronology of the foundation work undertaken in adaptive testing showing the initial major contributions.

2.4.2.1 Adaptive Testing: Software Path Testing

Reported results indicates that adaptive techniques can be an effective way of generating tests in unit testing as characterised by work by Korel and Ferguson, [199], [118] Jones *et al.* [183], Tracey *et al.* [301], [303], Michael *et al.* [233], [231], [232] and Wegener *et al.* [318]. In this work, success is measured in terms of being able to find data that causes execution of specific statements.

In particular, adaptive techniques seem to cope relatively well with a number of issues that appear problematical for other techniques such as procedure calls, arrays and in particular array references in conditions (Korel [200], Michael *et al.* [232], and Wegner *et al.* [318]). Work by Korel [199] also seems to show that adaptive techniques may be effective in dealing with pointers; though this particular application required the introduction of a specialised mechanism for backtracking and needs further empirical investigation.

However, the adaptive techniques are not without their problems, and one of the more notable aspects of research into application of these techniques is the number of issues that it has thrown up. The most notable concerns[18] Boolean variables and in particular those that are set during the execution of code (the "flag" problem). Other problem areas were noted by Wegner *et al.* [318] and Michael *et al.* [232] who highlighted a number of significant issues as a result of using industrial code. The following issues were all found to cause difficulties:

- binary values (Boolean flags) generated within the code (Wegner *et al.* [318], [232]);
- decisions that contain multiple conditions (Michael *et al.* [232]);
- side effects in conditions that makes insertion of instrumentation problematic (Wegner *et al.* [318]);
- short circuit evaluation in C that results in an artificial narrowing of the search (Wegner *et al.* [318]);

The Boolean flag problem is of particular interest for two reasons. First it has spawned a massive amount of follow-on research in the adaptive testing area from Bottaci [39]

[18] Certainly in terms of the number of papers generated.

Harman *et al.* [155], [157], Baresel *et al.* [23], [22], McMinn [224], Harman *et al.* [156], Baresel *et al.* [22], and Liu *et al.* [216].

Second, it seems to present a *fundamental* problem for automatic test data generation. The problem is basically one of information loss and flattening of the fitness function. The problem arises when internal Boolean variables take values derived from the input and the state of the Boolean flag is used to control program execution. This results in large areas of the fitness function evaluating to the same value. The comment that the Boolean flag problem seems to represent a fundamental issue comes from an observation made by Vinter *et al.* [308] from tests of an aircraft gas turbine controllers in simulation against single bit memory errors. Indeed the issue was observed in this work during the testing of the heapsort function in section 5.4.1.4. Research in this area is on-going.

2.4.2.2 Adaptive Testing: Worst Case Execution Times

The adaptive testing has been put to a number of uses aside from software path testing. As stated in the introduction, the technique has been applied to looking for worst case execution times, especially in the area of safety related software. Wegener *et al.* [321] suggested its use for best and worst case timing as a complementary technique to functional testing. This work was extended by Wegener and Grochtmann [320] by using simulated annealing for the more local search phases with no improvement. Grochtmann *et al.* [141] repeated this work with further examples and compared results from tests derived by human subjects with ambiguous results.

Alander *et al.* [6] investigated the use of genetic algorithms to test embedded systems in a simulation environment; the authors extended the work in [7] using the same environment to test the timing properties of a microprocessor-based relay used in electric grid applications.

Work has also been performed by Mueller and Wegener [236], Gross, Jones and Eyres [142], Gross and Mayer [143] and Nilsson and Henriksson [243].

2.4.2.3 Adaptive Testing: Applied to Systems

The final application area for adaptive testing is stress and robustness testing. The original work was performed by Cooper [80] on system performance. Schultz *et al.* [278],

[279], used adaptive testing in the context of functional testing of autonomous air vehicles (UAV). Using system simulation they examined the robustness of a control system in the presence of fault conditions. Evolutionary testing was used to drive the control system towards errors and combinations of errors that would result in the loss of the vehicle during landing manoeuvres.

Buehler and Wegner [48] used evolutionary testing and simulation in a hardware in the loop (HIL) environment to investigate the ability of adaptive testing to find scenarios where the control system for an assisted parking system failed. The work was extended by Wegener and Buhler [319] to compare the original fitness function based on distance with another based on area.

The same technique was applied to a brake-assist system linked to adaptive cruise control by Buehler and Sthamer [47]. The experiments were successful in detecting several errors in the systems investigated and interestingly for the brake-assist system the technique located an error that was not near operational boundaries. Pohlheim *et al.* [266] also applied the same procedure to hardware in the loop testing of an adaptive cruise control system.

One issue with the work reported above is that though errors were discovered there were no experimental controls. That is, because we have no idea of the total number of errors that were present, the work provides no information on the relative effectiveness of the technique. That being said, that is the situation in the real world where there is no *a priori* information about what errors exist.

2.4.3 Adaptive Testing: Fundamental Issues

There are two major issues and several minor areas of interest concerning the use of adaptive testing for test generation. The major issues are:

- the generation of effective fitness functions,
- techniques for performing the optimisation.

2.4.3.1 Fitness Functions

The construction of good fitness functions appears to be the critical area that determines the effectiveness of the search for test data. This fact is amply born out by considering the level of consignation caused by the Boolean flag issue.

Research has followed two major themes. The first is a pure path-based approach in which test data is used to execute a particular specified path, or all paths are sought. However, the majority of the work has focused on the second theme in which the evaluation of branch predicates is considered in addition to considering path-based information. There has also been a small amount of work that has examined predicate functions that are completely detached from these path based criteria, the most notable of these is the use of mutation adequacy by Baudry *et al.* [30], [29].

In more detail, the majority of the work on fitness functions has followed the approach first proposed in Miller and Spooner [235] which converts a predicate of the form

$$h(x_i)\ op\ g(x_i)$$

to a fitness (or cost) function of the form

$$f = F\ (|h(x_i) - g(x_i)|)$$

A number of authors have suggested variations on this theme and a summary of these proposals is presented in Table 4 where the notation has been normalised to that used in Gallagher and Narasimhan [128].

The first modification to the basic scheme proposed in Miller and Spooner [235] is found in both Gallagher and Narasimhan [128] and Jones *et al.* [183] where the basic fitness function is modified to give it a more "shaped" form. For example, Gallagher and Narasimhan [128] suggested a fitness function of the form,

$$G = e^{w_i |g(x)|^{-1}}$$

where w_i is a weighting factor. This function was selected to explicitly convert problems with non-linear path constraints into an equivalent unconstrained linear problem. This is necessary here as the optimisation technique selected (quasi-Newtonian) cannot deal with

the former class of problem. In addition Gallagher and Narasimhan introduce functions to deal with logical operators (and, or, not).

Jones *et al.* [183] suggested a family of fitness functions based on the reciprocal of the distance i.e.

$$f = F\,(|h(x_i) - g(x_i)|)^{-n}$$

Experiments were conducted with values of $n = \{1, 2, 3\}$ but no advantage was found in not applying the inverse square law. Note that here the optimisation technique used (genetic algorithm) did not place the same constraints on the form that the function can take as it did in Gallagher and Narasimhan [128]. These two examples are, however, the exception rather than the rule, and the majority of more recent work dispenses with this shaping. Whether this is an advantage or not does not appear to have been evaluated to any extent.

Tracey *et al.* [299] introduced a "punishment" factor that ensures that non-optimal solutions are clearly recognised as such and discussed the use of an offset K to enable values to be selected near boundary values. However, the impractical application of a punishment factor may be limited as it would be necessary for it to be selected on a per-predicate basis unless a data dictionary were available then conceivably it might be possible to automate this approach. Zhan and Clark [345] make a similar point when discussing an arbitrary value of 10 assigned to K. Another feature of adding K is the fact that if a predicate is satisfied, then the value of the fitness function is, of necessity, set to zero.

One area of particular interest is the evaluation of the logical conjunction operator (i.e. and). Table 4 gives the impression that the use of the addition operator is almost universal. This is not quite the case. For example Diaz *et al.* [103] used a modified version and a number of authors have used max(G(x), G(y)) e.g. Cheon and Kim [59]. Bottaci [40] analysed the performance of various options for both conjunction and disjunction: both in terms of desirable properties and practical application.

- P1 :cost (a or b) <= cost(a) and cost (a or b) <= cost (b)
- P2 : cost (a and b) >= cost (a) and cost (a and b) >= cost (b)
- P3 : the cost of logically equivalent expressions should be equal.

Importantly Bottaci [40] emphasized the point that the cost functions are only *heuristic* rules of thumb.

The other major approach taken by researchers is to construct fitness functions based on path coverage. Approaches in this category range from the simple approach taken by Watkins [317] where commonly executed paths are penalised, through to complex evaluation schemes such as that used by Lin and Yeh [214] in which similarities between path segments and complete control flow paths are evaluated.

Unfortunately the only back-to-back comparative work in this area by Watkins and Hufnagel [316] was not able to define what an optimal path based function would comprise of. It did however support the assumption that neither a pure predicate-based nor a pure path based fitness function would be optimal; but that rather both types of information need to be taken into account as for example by Bueno and Jino [49].

g(x)	Gallagher [128][19]	Wegener [318]	Tracey [299]	Diaz [103][20]	Zhan [345]
Boolean			if TRUE then 0 else K	0	as for Tracey [299]
x = y	$g(\mathbf{x}) \rightarrow x - y = 0$ $G = e^{-w \lvert g(x) \rvert - 1}$	G = | x –y |	if abs(x – y) = 0 then G =0 else G = |x – y| + K	G = | x – y |	as for Tracey [299]
x ≠ y	$g(\mathbf{x}) \rightarrow x - y \neq 0$ $G = ke^{-w \lvert g(x) \rvert}$	n/s	if | x – y | ≠ then G = 0 else G = K	G = | x – y |	as for Tracey [299]
x < y	$g(\mathbf{x}) \rightarrow y > x$	n/s	if x – y < 0 then G = 0 else G = (x – y) + K	G = y – x	as for Tracey [299]
x <= y	$g(\mathbf{x}) \rightarrow y >= x$	n/s	if x – y < 0 then G = 0 else G = (x – y) + K	G = y – x	as for Tracey [299]
x > y	$g(\mathbf{x}) \rightarrow x - y = 0$ $G = e^{-w\, g(x)}$	n/s	if y – x < 0 then G = 0 else G = (y – x) + K	G = x – y	as for Tracey [299]
x ≥ y	$g(\mathbf{x}) \rightarrow x - y = 0$ $G = e^{-w \lvert g(x) + \delta \rvert - 1}$	G = | y – x |	if y – x < 0 then G = 0 else G = (y – x) + K	G = x – y	as for Tracey [299]

x ∧ y
(and)
	G = G(x) + G(y)	G = G(x) + G(y)	G = G(x) + G(y)	if x and y TRUE then
				G = min(G(x), G(y))
				else
				$G = \sum_{c_i FALSE} G(c_i)$
					if x and y unsatisfied
					G = G(x) + G(y)
					otherwise 0

x ∨ y **(or)**	G = min(G(x), G(y))	G = min (G(x), G (y))	min(G(x), G(y))	G = min (G(x), G(y))	if x and y unsatisfied G = G(x) * G(y) / G(x) + G(y) otherwise 0 *after Bottaci [40]*
¬x	inverse relational operator	n/s	negation removed	negation moved inward and propagated over x	

Table 4. Summary of cost functions applied for selected authors, n/s is where information was not explicitly stated. In both Tracey, Clark, McDermid and Mander [303] and Zhan and Clark [345] K is a "failure constant" to further "punish" data that fails. In addition Tracey, Clark and Mander [299] convert to disjunctive normal form implying that a finer grained search results: the complete list is taken from Tracey *et al.* [302].

[19] There appears to be an ambiguity in the table for x ≠ y.

[20] There is an error in the table as printed, personal communication with author.

2.4.3.2 Search Techniques

The number of different optimisation techniques applied to adaptive test data generation is impressive. Table 5 gives a summary of techniques discussed in the literature, and includes a reference to the work, the search methods used and comments on any adaptations that may have been introduced and/or other techniques considered.

Details for all of these methods are beyond the scope of this review, but a number are covered in detail by Michalewicz and Fogel [234] and a good introduction to the different paradigms of evolutionary computing is provided by Eiben and Smith [109].

The majority of work has been performed with what today can be considered fairly standard optimisation techniques. However, there is a clear trend towards the use of meta-heuristic techniques, specifically genetic algorithms and simulated annealing. The reasons for this are clear. With meta-heuristic techniques it is possible to develop software that is highly discontinuous in nature and to target functions derived from predicates in particular tend to be either non-differentiable and/or not continuous. In addition meta-heuristic techniques are, in general, better at dealing with this type of optimisation, as they tend to avoid local minima [234].

The basic search techniques seem to perform well for straightforward code but it has been found necessary to extend the basic optimisation paradigm to deal with specific issues. For example, the introduction of backtracking to deal with pointers in the case of Korel [199], the addition of chaining by Ferguson and Korel [118] and a similar adaptation discussed by Wegener *et al.* [318] and applied in conjunction with genetic algorithms in McMinn and Holcombe [227]. In addition both Korel and Tracey noted that further optimisation can be obtained by limiting the variables used in the optimisation by using either data flow analysis as in Korel [199] or by limiting optimisation to those variables involved in the predicate currently being examined [299]. Results from Diaz, Blanco and Tuya [102] with scatter search suggest that artificially limiting variable ranges might also be advantageous.

Study	Search Method	Comments
Cooper [80]	heuristic (rule based) search	also considered gradient decent and probabilistic search
Miller [235]	numerical optimisation	limited to real values and excluded integer values
Andrews [10]	complex search	geometric manipulation of a surface called a complex
Korel [199]	direct search, gradient decent	further optimisation via data flow analysis heuristics
Schultz [278]	genetic algorithm	applied to system testing
Ferguson [118]	direct search, gradient decent	backtracking supplied via "chaining" of intermediate goals
Jones [184]	genetic algorithm	
Gallagher [128]	quasi-Newtonian numerical	explicit conversion of constrained optimisation problem to unconstrained problem
Jones [183]	genetic algorithm	suggests looking at Tabu search
Tracey [299]	simulated annealing	suggests restriction of variables included in optimisation
Tracey [300]	simulated annealing	deferment of loop predicates
Pargas [260]	genetic algorithm	selection of fittest members for next generation
Tracey [302]	genetic algorithm	
Michael [232]	genetic algorithm	auxiliary table to track branch coverage, differential genetic algorithm and, gradient decent used as reference algorithms
Wegener [318]	genetic algorithm	partial goals meet recorded
McMinn [226]	genetic algorithm and ant colony optimisation	
Diaz [103]	tabu search	suggests scatter search, used in [102]
Blanco [37]	scatter search	compares tabu and scatter search
McMinn [227]	genetic algorithm and chaining	

Table 5. Summary of optimisation techniques used and additional heuristics that were applied.

Other problems with search techniques seem to be lurking in the wings however. For example deeply nested decisions and those containing multiple conditions (Michael *et al.* [232]) appear problematic because of the manner in which the evolutionary systems solve one constraint at a time. Baresel *et al.* [25] investigated pre-evaluation of the predicates and McMinn *et al.* [225] and Harman *et al.* [155] examined the use of program transformations, which in their test subjects increased the speed of convergence. Likewise, there may be issues with loops, short circuit evaluation of logical predicates and side effects, again stated as a topics for further investigation by Baresel *et al.* [25].

An interesting approach to using evolutionary programming is suggested in [28] in which the crossover operator is dispensed with completely. The authors compared their approach with bacterial evolution and showed that it offered some advantages over other

approaches for the code that they tested. For example, they claimed that population size does not need to be constrained, that the problem space changes as mutants are removed from consideration and that only two parameters need to be tuned, namely the number of individual saved and the minimal size of the bacteria . This mutation-only approach fully follows the evolutionary programming methodology and given the possible advantages, it deserves further investigation.

Which of the optimisation techniques discussed above is superior cannot readily be addressed largely because the only direct comparative work was performed by Blanco [37], who compared scatter and tabu search on a very limited number of examples. These experiments did find that tabu search techniques were superior, but the results are too limited to have any real significance. However, one very interesting point arises from this review, namely that nowhere else in the testing literature has there been anywhere near the same number of potentially significant issues raised. This observation does not suggest that the problems are more difficult, just that they seem numerous, even compared with symbolic execution based techniques. This is itself interesting as the search techniques that both systems (adaptive & symbolic) employ are very similar. For example, work by Miller and Spooner [235] and Coward [84] used very similar numerical optimisation techniques. In the area of adaptive testing, the work that most closely approximated symbolic execution was Tracey Clark and Mander [299] where the preconditions and negated post-conditions are converted to disjunctive normal form which mimics, at least in part, the construction of the predicate condition from symbolic execution.

2.4.4 Adaptive Testing: Summary

The feature of research in adaptive testing that stands out is the variety of different things that have been looked at, in terms of problem areas examined and optimisation techniques but possibly more importantly in terms of finding suitable fitness functions that will actually allow the techniques to perform to their potential. This is potentially a serious problem, given that what is being attempted is to embody a set of general path following rules in a single numerical expression in a similar manner to software complexity metrics.

It is also remarkable the number of issues that research in this area has thrown up, issues which are probably associated with *all* test data techniques but which have not been

reported directly elsewhere, the most obvious being the issue with computed Boolean variables (flags) report by Wegner *et al.* [318] and Michael *et al.* [232].

However, not withstanding the above comment, the technique is very effective at both finding test data to meet code coverage requirements and in functional testing, finding situations that actually cause failures. However, at the code level, the technique is essentially path following and code coverage alone is no guarantee that the software is free of errors, or rather that the test set is "good". As Beizer [31] observes, path coverage is not capable of detecting missing paths. This is of course not an issue when the technique is applied in functional testing as demonstrated by Schultz *et al.* [278], [279] and Buehler and Wegner [48].

In addition, Michael *et al.* [232] have observed that the vectors obtained tend to be unusual and the implication is that deciding the correctness may be problematic. This is in contrast with observation by Jones *et al.* [183] that the vectors generated tend to be "uninteresting" which is equally problematic in that uninteresting tests are unlikely to discover errors.

2.5 Symbolic Testing

2.5.1 Symbolic Testing: Introduction

As with adaptive testing, symbolic execution takes as its primary input, the program under test. Symbolic execution differs from normal execution in that it involves the replacement of each variable with a symbolic value, rather than with a numeric value. Where the input parameters are referenced in the software, symbolic values are substituted. These values are propagated through to all variables on each execution path selected. The process is best described by looking at the example in Table 6 adapted from King [192].

	Statement	a	b	c	x	y	z
1	int sum (int a, b, c) {	v_1	v_2	v_3	-	-	-
2	x = a + b;	-	-	-	$v_1 + v_2$	-	-
3	y = b + c;	-	-	-	-	$v_2 + v_3$	-
4	z = x + y - b;	-	-	-	-	-	$(v_1 + v_2) + (v_2 + v_3) - v_2$
5	return (z); }						$v_1 + v_2 + v_3$

Table 6. An example of symbolic execution for a simple C program, adapted from King [192].

The program in Table 6 takes three inputs *a*, *b* and *c* and has the same number of internal variables, *x*, *y* and *z*. In statement 1, the variables are assigned the symbolic values *v1*, *v2* and *v3* respectively. In statement 2, the internal variable *x* are assigned the symbolic value of *a* + *b*, which symbolically is *v1* + *v2*. In the same manner, *y* is assigned the symbolic value of *b* + *c* i.e. *v2* + *v3* in statement 3. Likewise, in statement 4, *z* is assigned to the current symbolic value of *x* and *y* in terms of the input parameters: that is, *v1* + *v2* and *v2* + *v3* and *b* in terms of its input value *v2*.

In statement 5, the value assigned to *z* in statement 4 has been simplified in relation to *v2*. This demonstrates one of the strengths of symbolic execution - its ability to represent the complete function being calculated in a simplified form.

As the example above demonstrates, symbolic execution is conceptually simple for straight line code. However, when branches are introduced, there is a requirement that the predicates that determine which specific path is taken at each branch are tracked. This is accomplished by building a path condition (*pc*). At the start of the execution the *pc* is initialised to TRUE. Then as each branch point is encountered the predicates for that branch are co-joined with the current value of the *pc*. The difficulty with this approach is that the decision about which branch is to be taken cannot be determined during symbolic execution, so both branches must be taken as long as the *pc* for both branches remain feasible. That is, the *pc* for both branches being followed has a valid solution. This is a significant issue for loops where the number of iterations may be bounded by values dependant on input values. For example, consider the code fragment in Figure 10 where *v* and *limit* are the input values.

```
while (v < limit)
{
    if (v > limit / 2) v = v - 1;
    else v = v + 3;
}
```

Fig. 10. An example where it is difficult to statically determine a closed form of the *pc*.

The resulting *pc* for this code is a tree where, for each possible branch, after the first pass;

(1) $pc = \text{true} \wedge (v \geq limit)$ ***or***

(2) $pc = \text{true} \wedge (v < limit) \wedge (v > limit/2)$ ***or*** {also implies $v \leftarrow v - 1$}

(3) $pc = \text{true} \wedge (v < limit) \wedge (v \leq limit/2)$ {also implies $v \leftarrow v + 3$}

For a second pass, the path conditions (2) and (3) are extended in the same manner, for example (2) is extended as follows;

(4) $pc = \text{true} \wedge (v < limit) \wedge (v > limit/2) \wedge (v - 1 \geq limit)$

(5) $pc = \text{true} \wedge (v < limit) \wedge (v > limit/2) \wedge (v - 1 < limit) \wedge (v - 1 > limit/2)$

(6) $pc = \text{true} \wedge (v < limit) \wedge (v > limit/2) \wedge (v - 1 < \text{limit}) \wedge (v - 1 \leq limit/2)$

In this case the series can be extended ad infinitum. For example, given initial values of $v = 3$ and $limit = 5$ the loop will terminate. However, given initial values of 5 and 10 the loop will cycle indefinitely.

To generate test data for any path through the code, the path conditions need to be "solved". This can involve two steps. First, it can be advantageous to simplify the *pc*. For example, the path constraint given in (4) is inconsistent in that it requires that $(v < limit)$ and $(v - 1 \geq limit)$, which cannot be satisfied and so represents an infeasible path. Early detection of infeasible paths is advantageous in that it reduces the amount of work that needs to be undertaken and can provide useful information in terms of whether code meets its requirements. For example, an infeasible path may indicate a coding error. Second, for those constraint systems that are feasible, a solution to the system of equations represented by *pc* needs to be located; this solution provides a set of test data that will cause the paths to be executed.

In addition to constructing the path constraint, symbolic execution systems also needs to keep track of all the operations that affect the outputs, in terms of inputs along the path being traversed. This is referred to as the output conditions.

2.5.2 Symbolic Testing: Review

A large body of work on symbolic execution of programs was undertaken in the middle 1970's and early 1980's, early work being undertaken by Boyer *et al.* [41], King [191], [192], Clarke [63], Ramamoorthy, Ho and Chen [268] and Howden [170], [171]. All of whom performed significant work on the "reliability" of symbolic execution for detecting errors.

King [191] provided a brief description of symbolic execution and the EFFIGY system, which was developed as a debugging and test generation aid for a simple PL/I like language with limited data types. Symbolic manipulation and simplification were based around the King's earlier work on program verification and path constraints were solved using linear programming techniques. Path selection was made manually via a user interface though the possibility of performing this function automatically was suggested

Boyer, Elspas and Levitt [41] provided a more substantial coverage of the topic and introduced many of the points covered in more detail in latter work. Their system, SELECT - was based around a subset of Lisp and that was used to advantage with the path and output conditions being stored as Lisp lists. SELECT also allowed the program under test to be annotated with assertions. As with King [191], expression simplification was performed by adapting a program verification tool which allowed the early detection of infeasible paths. The SELECT tool was originally designed to be used interactively. However paths can be generated automatically and the common practice of dealing with loops by executing them a user selected number of times was introduced. Of particular interest are the three approaches taken to generate numerical test data from the *pc*. These are the use of integer programming, the use of mixed integer programming for dealing with floating point variables and because these two techniques are limited to dealing with constraint systems with linear relationships, the use of hill climbing. In an aside, the authors comment that to deal with non-linear constraints such as "$X*Y + 10Z - W \geq 5$" (pg 238) by using integer or mix integer solvers they would have to be "*prepared to assign to X a trial value, and then attempt a solution*" (ibid), very similar to the approach to that taken by constraint solving systems.

Clarke [63] described a system for symbolically executing FORTRAN programs. As in King [191], path selection was performed manually, but the feasibility of path conditions

was not decided until after the complete path was constructed. However, the order in which conditions were added to the *pc* was maintained to ease detection of where an infeasible clause was added. Solutions to constraint systems were found using linear programming. Clarke also briefly discussed issues associated with aliasing of array elements.

King [192] expanded on his previous paper, discussing the issues surrounding syntax versus semantics that were introduced in Boyer [41]. King clearly pointed out that arithmetic and logic as implemented is not the same as their counterparts from mathematics. This paper also discussed issues with array indexing and put forward two possible solutions: exhaustive case analysis and leaving the ambiguity unresolved but storing the output conditions that set the value of the index variable. Neither of these solutions appear to have been implemented and nether appear to be totally satisfactory due to the volume of information that would need to be maintained.

Ramamoorthy, Ho and Chen [268] implemented a symbolic execution system that constructed paths and output conditions by working backward from the outputs, rather than forward as demonstrated in section 2.5.1. The authors also took a novel approach to finding numeric data to satisfy the *pc* that they described as "*systematic trial and error*" (pg. 296).

Another interesting feature of this system was the explicate use of backtracking, which took forward the idea from Boyer [41] of assigning a variable a trial value to a variable in the *pc* and then attempting a solution. To some extent, this technique anticipated the use of constraint solving techniques as in Hentenryck *et al.* [159] and Nikolik and Hamlet [242]. Furthermore, a novel solution to arrays was suggested, in which the creation of new "nearly identical" instances of the array were created.

Howden is associated with the development of a symbolic execution method [168] and the DISSECT [171] tool which is based on it. Although these developments do not add substantially to the work cited above and it is unclear whether they are capable of generating test data, the use to which they were put is interesting. In these two papers [169], [171], the path testing strategy was pitted against programs with known errors. The major result was that for one set of programs the technique was reliable for only 65% of the errors [169] and that for the other symbolic testing only resulted in a 10-20% increase in effectiveness and was reliable for 18 of the 28 known faults [171]. While this may seem

a disappointing result, branch adequate test sets were reliable for only six of the 28 errors, and special value testing was reliable for 17 of 28. Howden's conclusion was that "*no one program analysis technique or program testing strategy should be used to the exclusion of all others*" [171] (pg. 394).

Darringer and King [96] looked at a number of issues associated with the use of symbolic execution in the context of testing. In particular, they noted several reasons why it is desirable to generate actual test data, namely that:

- the semantics of the symbolic system and the actual system may differ;
- we may need to obtain performance (timing) information;
- actual outputs may show errors that may otherwise be missed.

Much of the latter, post 1980 work is to a certain extent derivative. For example work, on the SADAT tool by Voges *et al.* [310] contains little detail on test data generation and was mainly concerned with the integration of different program analysis functions within a single framework. Work by Kemmerer and Eckmann [188] only extended the paradigm to the Pascal language. All the early work and closely related static analysis issues have been surveyed in detail by Coward [83], [82].

Taken together, the work cited above does illuminate the main issues that symbolic execution needs to deal with, namely:

- path selection of branches and in loops;
- the aliasing of variables (arrays, pointers, function/procedure calls);
- solving the path constraints.

In most cases, early symbolic execution systems required the user to select the paths to be taken interactively. The exceptions being Boyer [41], who attempted to cover all paths and Ramamoorthy *et al.* [268], who targeted all branches. However all the systems either required the user to specify the number of times that loops were iterated, or to supply a maximum number of iterations.

In the late 1980s Coward [83], [84], [82] undertook the construction of a symbolic execution system for COBOL. This system is interesting for a number of reasons:

- it considered the issue of symbolic execution for non-numeric data e.g. character strings;
- it considered how records should be incorporated;

- it used a technique of splitting branches three ways to deal with non-equalities of the form $a \neq b$ that cannot be solved numerically;
- it introduced the concept of selecting paths based on the amount of a variable's input domain that can be covered.

Lindquist and Jenkins [215] described a static analysis tool that utilises most of the features of symbolic execution to perform static analysis of subset Ada. Their paper focused on the IOGEN symbolic execution tool but considered the application to test generation. However, their description of this lacks detail. The paper brought out some interesting points including the observation that the test adequacy criteria of executing all paths once cannot be reliable against errors such as divide by zero.

The next significant body of work was undertaken by Offutt in conjunction with DeMillo [99], [100], King [193] and Seaman [254]. This research concerned the development and integration of the Godzilla test generation tool with the Mothra (King [251]) mutation system. In this work, the *pc* was constructed as shown above to define what the authors' term the reachability condition. The authors co-joined the reachability condition to what they termed the necessary condition. This condition constrained the data generated for the reachability condition so that it was able to differentiate the original source program from a mutated form of the program thus incorporating mutation testing as proposed by Hamlet [154] and DeMillo *et al.* [98], directly into the test data generation process. Solutions to the constraint systems thus constructed are made via algebraic simplification of the path condition, domain reduction to reduce the number of feasible values, and special purpose heuristics to select trial values from domains (Offutt [249]).

Work along similar lines, by Goldburg *et al.* [132] and Jasper *et al.* [181] on symbolic execution for test generation which used a theorem proving system as a base, reported positive results for the limited amount of production code that they tested to date. The system itself deals with a restricted subset of the Ada language. However, the authors noted that this was not as limiting as it might seem because the code being tested was targeted at embedded systems, and these tend to use a limited subset as a matter of course. The observation is of interest because of such subsets are the accepted norm in safety-related work. Examples of this are the MISRA C subset of the C language [18] and the SPARK subset of Ada [26].

Girgis [129] detailed a FORTRAN system for performing symbolic execution and discussed the path selection problem in terms of loop traversals zero, once and two times. More interestingly, given the scarcity of results in this area, this paper reported results on testing five small, mutated programs to examine the effectiveness of the technique. For the eleven classes of fault reported, it was found that faults were found most effectively either during the symbolic execution phase or during test execution by comparing generated and expected results. This result indicated that the two phases are complementary to one another: symbolic execution appearing superior for errors involving references or definitions of variables and test execution better for errors involving operators or constant values.

Nikolik and Hamlet [242] examined part of the symbolic execution problem associated with ambiguous array references and presented a solution that involved the substitution of indexed terms with index-free terms using constraint programming languages (Cohen [71]). Note however that this work was performed as a standalone exercise and was not integrated into a general purpose test generation tool. It is however significant in that it demonstrates that at least part of the aliasing problem is tractable.

Gotlieb *et. al.* [136] produced a symbolic execution tool that operated on a subset of the C programming language that excluded difficult-to-deal-with features such as `goto` statements, pointers and dynamically allocated structures. They also pre-processed the code to be analysed into a static single assignment form (Cytron *et al.* [86]) that removed much of the possible ambiguity for variable references.

Lapierre *et al.* [208] described a symbolic execution system and test data generator for a subset of the C language. Rather than working directly from the control flow graph, the approach these researchers took was to apply the procedure suggested by Bertolino and Marre [33] for converting a control flow graph into an execution tree, and to then use the unconstrained arcs to determine a minimal set of paths that need to be traversed. Of 684 edges in their set of test programs, only 124 were unconstrained. This combined with a process of generating trees for zero, one and two iterations of loops, was claimed to lead to smaller paths. To find numerical data that satisfy the *pc*, the *pc* was converted into a system of linear constraints and solved using mixed integer linear programming techniques. The paper reported results for a non-trivial mix of programs, specifically

selected to include features such as the use of pointers and pointer arithmetic. Edge coverage was obtained for eight of the ten subject programs. This prompted the authors to come to the conclusion that "*full automation of test data generation was unattainable*" [208] (pg. 196) but observe that human intervention can be minimised.

Meudec [230] constructed a tool (ATGEN) using the ideas from constraint satisfaction to generate test data for SPARK Ada programs using the ECLiPSe (Wallace, Novello and Schimpf [314]) constraint solving library for Prolog. As in Gotlieb *et. al.* [136] the code was pre-processed - in this instance by encapsulating the Ada syntax in a Prolog wrapper; to allow the source to be operated on directly. The system itself had a layered approach to the constraint solving problem, and while it used the basic Prolog backtracking mechanism, there was a high level of reliance on heuristics to intelligently select and label (instantiate) variables for the data sets which would be assigned first. The example was given of finding solutions to the constraint, $x \times 2 + y = 10$ where it was noted that if the assignment to x is made first then the resulting problem is much easier to solve, than if a value was first assigned to y.

Another system based on the ECLiPSe system was presented by Gouraud *et al.* [137]. Again, the source code was pre-processed as a list of atoms with the constraint resolution system being similar to the two systems discussed above. The work is novel because of the manner in which it generated control paths to be tested. Simplistically it treated the problem as being the same as generating all paths of length $\leq n$ in a regular language and drew them randomly with a uniform probability from the complete set to obtain coverage.

The efficiency of constraint satisfaction techniques depends on the heuristics used to guide the selection of variables to label. For example, Meudec [230] demonstrates that early selection of variables with non-linear terms was advantageous; however, it is to be assumed that otherwise standard heuristics were employed. In Gotlieb *et. al.* [136] the heuristics are not explicitly stated but standard constraint solving techniques such as smallest domain, most constrained values and bisection of domains were mentioned. In Gouraud *et al.* [137] the exact process is given explicitly. For constraint solving the following rules were applied for selecting which variables to be assigned first:

- Variables that do not depend on other variables.
- Variables that occur first in execution order.

- Variables with the smallest domain (fail first).
- Variables that bring into consideration the largest number of constraints.

The first two were claimed to be novel and the second two, were described as standard heuristics. Remaining variables were instantiated randomly which the authors claimed increases the ability to detect infeasible paths over multiple attempts.

Another variant on symbolic execution is to hybridise it with actual execution of the code as in adaptive testing. Gupta *et al.* [145], [146], [147] have proposed a technique that applies a number of symbolic- and execution-based techniques to find a linear approximation to the function being computed on any one path. For programs that have linear predicates, the authors suggested the technique should be able to compute vector increments in a single pass - with no back tracking. The authors also reported practical issues with the use of Gaussian elimination: free variables were assigned ad hoc values, which can cause the system to become inconsistent. There were also issues with convergence. The authors stated that their method should be expected to behave like Newton's method. However, Newton's method is not guaranteed to converge (Michalewicz and Fogel [234]). In their latter work [146], [147] Gaussian elimination was replaced by an interior point method based on least square errors, which has the advantage that any solution is acceptable.

Offutt, Jin and Pan [250] introduced a variant of symbolic execution that incorporated some of the ideas from adaptive testing in that the variable domains were trimmed as the path was followed, thus ensuring that a feasible solution was maintained. The difference between this and earlier work is that in this work symbolic representations of the output values are not explicitly required at any point. In many respects, this variant of symbolic execution mirrored the process used in the constraint satisfaction techniques with in built backtracking and a selection process for the next variable to be instantiated. Thus this approach appears equivalent to the "fail-first" strategy of variable ordering [27] and to the domain bisection technique in Gotlieb *et al.* [136].

Dillon and Meudec [104] reported results on a development of the ATGEN tool [230] for C language programs. Data for two sets of programs were examined; namely code used

by Wegener *et al.* [318] and industrial code supplied by Ellims[21]. Path coverage results obtained from vectors generated by ATGEN were compared with results for a commercial tool C++Test [259] and the ATGEN test sets achieved significantly better results in five of twelve cases reported with usually fewer test cases. With the test code supplied by Wegener, the C++Test tool performed very poorly, but results for the industrial code were better. This is not completely surprising as the industrial code was specifically designed with unit testing in mind

Lee *et al.* [209] constructed an integrated symbolic execution system for Java code and the paper focuses primarily on design decisions that were made in the construction of the system notably in the area of path enumeration, and on a possible approach to the issue of aliasing of indexed arrays.

On the first topic, the authors presented an extended discussion of the choices that were made when deciding which paths to include in the construction path predicates. In summary, their approach was similar to that taken in previous work in which loops (in particular) were executed zero to L times where L was a tuneable parameter. In addition, the authors stated that the path generator was designed to enumerate all possible combinations of paths though control statements. As part of their analysis of how many paths could be generated they provided a recursive equation for the number of paths though a control node N_i. For one of their examples - a program with two nested while loops and an if statement at the innermost level - for $L = 1$ there were 9 paths, for $L = 2$ there were 343 possible paths and for $L = 3$ there were 33,825. For $L = 4$ there were nearly two billion paths.

Xie *et al.* [338] described a system for dealing with object-oriented code that generated method sequences targeted at assertion checking (pre and post-conditions) and robustness testing. This work is unusual for the number of methods that were used to deal with symbolic constraints. Theorem proving systems were used for simplification and for testing whether the system is consistent, while a constraint solving system was used to generate actual test data.

[21] Details of the code can be found in Ellims, Bridges and Ince[112].

The DART system developed by Godefroid *et al.* [131] provided an example of what can be achieved by the hybridisation of two techniques. The system was designed to co-execute the program under test using both symbolic execution and randomly generated initial test vectors. The immediate advantage of this was that concrete values were available to the symbolic execution at points in the execution where a) the constraint solver/theorem-proving system was unable to find solutions or b) symbolic information was not available such as with library calls. Co-execution such as this has two advantages, information from symbolic execution can be used to guide the selection of new data values and the system also has a fallback mode in cases where symbolic execution becomes "stuck", execution in these instances can be restarted with a new random vector.

The DART system was targeted at executing all paths using a depth-first search of the tree but the subject of loops was not explicitly dealt with. The authors did, however, address the oracle problem, the tool being targeted at locating execution failures (exceptions) and assertion violations. Empirical evaluation of the DART system was performed on two programs: a small air-conditioning control example (17 lines) and an implementation of the Needham-Schroeder public key authentication protocol (400 lines)[22]. In both cases assertion violations were located in reasonable time periods (1 second & 22 minutes) in neither case did random testing find solutions after several hours of searching. A larger investigation was reported on an open-source implementation of the Session Initiation Protocol[23] (30,000 lines) that located hundreds of references to null pointers and one security violation.

2.5.3 Symbolic Testing: Issues

Historically, there are several fundamental issues associated with symbolic execution of code that need to be examined in more detail. These are:

- path selection, branches, loops and infeasible paths;
- the issue of aliasing of variables;
- the difference between the syntax and the semantics of the program under test;

---

[22] Ross Anderson of Cambridge University though this highly interesting (personal communication).

[23] http://www.gnu.org/software/osip/osip.html

- solutions to the resulting constraint systems.

2.5.3.1 Symbolic Testing: Paths and Path Selection

One of the problems traditionally associated with symbolic execution is the selection of paths to be tested, where a path is generally taken to be a control flow path in the control flow graph of the program under test. This problem is typified by early work by Clark [63] and Howden [170], [171] and Voges *et al.* [310] in which the selection of the path is left to the user of the system. However, work by Boyer [41] attempted to meet the all-paths criteria and the majority of latter work attempts to meet the all-branches criteria (e.g., Ramamoorthy *et al.* [268], DeMillo and Offutt [99] and Gupta *et al.* [146]). Some latter work is directed at more stringent criteria such as decision coverage (Meudec and Dillon [104]) and some work uses more "unusual" criteria such as data-flow adequacy (Clarke [64]) or basis sets (Gupta *et al.* [145]). However, this type of research represents a small minority of such work on path selection.

How to deal with loops that depend on input data is a general problem for symbolic execution. Given that it may not be possible to establish fixed criteria for loop termination in general, a pragmatic approach such as ensuring that loops are executed zero, once and twice is often adopted (e.g. Ramamoorthy *et al.* [268], Girgis [129] and Lee *et al.* [209]).

The flaw in this approach is that it is possible for the loop to fail on the n^{th} iteration for a reason such as overflow, underflow or access off the end of some data structure. It is also common practice to walk though a set of options using switch or case statements with a loop. In cases such as this limiting the loop to zero, one or two iterations may not even attain statement coverage.

Other approaches have been taken to the loop traversal problem, most notably by White and Wisziewski [327], [328] who used analysis of the control flow graph to extract simple loop patterns forming sets of regular expressions that describe paths that could be taken though the loop. These were used along with requirements for domain testing [326] to determine a "minimal" set of tests. White [328] described a tool for performing such an analysis but this does not appear to have been coupled with a tool to automatically generate test data.

Closely allied with the issues associated with loops is the issue of detecting infeasible paths. In general determining whether a path can be taken is undecidable because it is equivalent to the halting problem. For symbolic execution, the problem arises as we are attempting to locate a set of data that will execute a given instruction; the equivalence arises because we can replace any arbitrary statement with a HALT statement [153]. The infeasible path issue has been investigated in a number of papers. For example Woodward *et al.* [336] found that for some numerical routines in the NAG library [19] the number of infeasible paths increases exponentially compared with the number of feasible paths as the path length in units of a linear code sequence and jump (LCSAJ) increases. Similarly Gupta *et al.* [147] found a number of infeasible paths in routines taken from Numerical Recipes in C [267].

The usual solution to the infeasible path issue is to have the search for test data to halt after some fixed time period if no data has been found. Indeed, it is difficult to see how else the infeasible path issue can be addressed in a simple manner. One variant worth noting, however, was introduced in Gouraud *et al.* [137] where a time limit was used, but combined with multiple attempts. This variant was claimed to effective because of the introduction of a random component in the labelling process of a constraint programming language (CPL) based system.

Unfortunately the use of time to limit the depth or breadth of a search makes direct comparisons of efficiency between different techniques difficult because the amount of computation possible in any period changes over time and because different techniques require different amounts of work to achieve the same result.

2.5.3.2 Symbolic Testing: Aliasing

The next issue to be examined is the aliasing of variables, where it is difficult to uniquely determine the memory location that is being referred to. The most common occurrences of aliasing occur with array references, pointers references and function calls.

A number of solutions of varying worth have been proposed; however none are completely satisfactory in that they only solve part of the problem. For example a number of authors have side-stepped the issue, by solving for one element such as Coward [84],

[82] or as in DeMillo and Offutt, two [99], [100] array elements, however this is clearly unsatisfactory.

Other approaches have been proposed. Clarke [63] proposed complete enumeration of all possible values, but did not implement this. Ramamoorthy *et al.* [268] proposed a system that created new instances of an array whenever an assignment to the array takes place is introduced. Resolution of array elements is then delayed until after assignments have been made to the indexing variables.

In a similar vein Goldberg *et al.* [132], [181] arrays are dealt with via a "symbolic history list" similar to a stack, representing sequential assignments. These, in turn can be treated as logical propositions and passed to the theorem prover for resolution. An alternative method is suggested in [181] where the assignment history is recorded as a series of disjunctions. However this was considered by the authors impractical because of the possibly huge number of terms involved.

A more successful approach in general seems to be the use of constraint programming languages, The method proposed by Nikolik and Hamlet [242] where complex array indexes are substituted for simple variables appears to be effective if rather involved, but the process is completely mechanised. Meudec [230] reported the use of a CPL based system and gives a reasonably complex example of performing insertion sort on an array of structures; as with [268] the labelling of array variables is delayed until after the labelling of the index terms. However Medec reported that run time degrades rapidly as the size of the array increases.

Unfortunately arrays are not the only instances where aliasing can occur. Lee *et al.* [209] highlights the issue of aliasing parameters and in practice any large or dynamic structure is likely to be problematic.

2.5.3.3 Symbolic testing: syntax versus semantics

Another of the major issues with any symbolic manipulation of programs is that the values being manipulated are not the same as the values that are manipulated in mathematical expressions (King [192], Goldberg [133]). For example, relationships such as associative laws for addition of floating points do not necessarily hold. Consider the following expression;

$$a + (b + c) = (a + b) + c$$

This may not be true if the values of b and c are small relative to a. In this case it is possible that $a + b = a$ and that $a + c =$ a but that the sum $b + c$ is large enough to affect the value of a. Worse is the expression;

$$a + (b - c) = (a + b) - c$$

This inconvenient property affects different symbolic test generation system in different ways. For example, King [192] noted that this property precluded the use of many powerful simplifications. This point is echoed in work based on theorem-proving systems reported in Goldburg *et al.* [132] where the absence of the associative property for floating point variables means that axiomatisation of floating point properties is difficult. Furthermore it was also noted that the axiomatisation of integer variables in general assumes that they are unbounded (i.e. overflow is ignored). In many situations this is an invalid assumption, and in languages with small integer types such as C the wrapping of unsigned values from their maximum value to zero is often done deliberately by programmers. Accidental overflow is off course always an issue.

Although the issue of semantics is most critical for symbolic testing approaches that use theorem proving, it is also an issue in any system where simplification is employed, for example, in the optimisation of floating point arithmetic in compilers (Goldberg [133]) as well as issues such as removal of "unnecessary" code such as timing loops. However, to a large extent the issue is not usually addressed. For example, Clarke [63], Howden [171] and Bicevskis *et al.* [35] all mention simplification but do not consider this issue.

Semantics is also an issue in systems that use techniques other than symbolic execution, for example, in systems that use constraint programming languages or that are based on Prolog, e.g., the ECLiPSe system [314] used by Meudec [230] and Gouraud *et al.* [137], Meudec noted that floating point variables are approximated by rational numbers as they are for example in Spark Ada (Barnes [26]).

2.5.3.4 Symbolic Testing: Constraint Solving

A number of different techniques have been applied to the problem of solving the constraint systems that form the path predicates. These can be roughly grouped into three broad categories as follows:

- numeric techniques where standard linear programming optimisation techniques are employed;
- heuristic based methods, including theorem-proving systems;
- systems that use some form of constraint logic programming.

These three categories are very general and in practice there is considerable overlap, especially between the latter two. For example, Ramamoorthy *et al.* [268] employed a technique described as systematic trial and error, which, as noted previously, was conceptually similar to the labelling and backtracking mechanism employed in constraint logic programming based systems. Both Bicevskis *et al.* [35] and Offutt *et al.* [250] have employed domain reduction techniques that also have a similar parallel.

Numeric optimisation techniques suffer a number of problems, chief of which is the fact that they are usually targeted at finding an optimal solutions to a given problem. Coward [84] addressed this problem by selecting a technique that could provide any solution. However, the majority of techniques are not able to obtain multiple solutions. Other weaknesses associated with numeric optimisation techniques include sensitivity to internal parameters and the need to form the problem in a linear form (Gupta, Mathur and Soffa [147]).

The use of methods that are constrained to so that they can only be applied to linear problems has been justified by reference to work that shows that very few conditional tests have a non-linear component. For example, studies by White and Cohen [326] suggested that non-linear predicates may be rare. However, this could be an effect related to the problem domain of the programs being tested and it is conceivable that in other domains this would not hold as strongly. For example in engine control systems where physical behaviour is being modelled, non-linear effects feature quite strongly. Although this is a specialised application area, one only has to consider that modern vehicles can have tens of embedded systems on board and the scale of the potential problem becomes apparent.

The second major grouping listed above is rule-based methods, which include both the use of theorem-proving systems and heuristic methods. Work here has been limited to a small number of isolated instances, presumably because of the complexity of underlying systems. Thus, it is difficult to draw any strong conclusions as to the prospects for systems based around these types of tools.

The final grouping is systems based on constraint logic programming systems. In many cases, these systems have been built using special purpose libraries that extend the basic Prolog (Clocksin and Mellish [65]) logic programming paradigm to take advantage of the inbuilt backtracking mechanisms. Specialised domain reduction systems have also been constructed by Bicevskis *et al.* [35] and Offutt *et al.* [250]. An overview and summary of how this categorisation maps onto the work cited above is given in Figure 11.

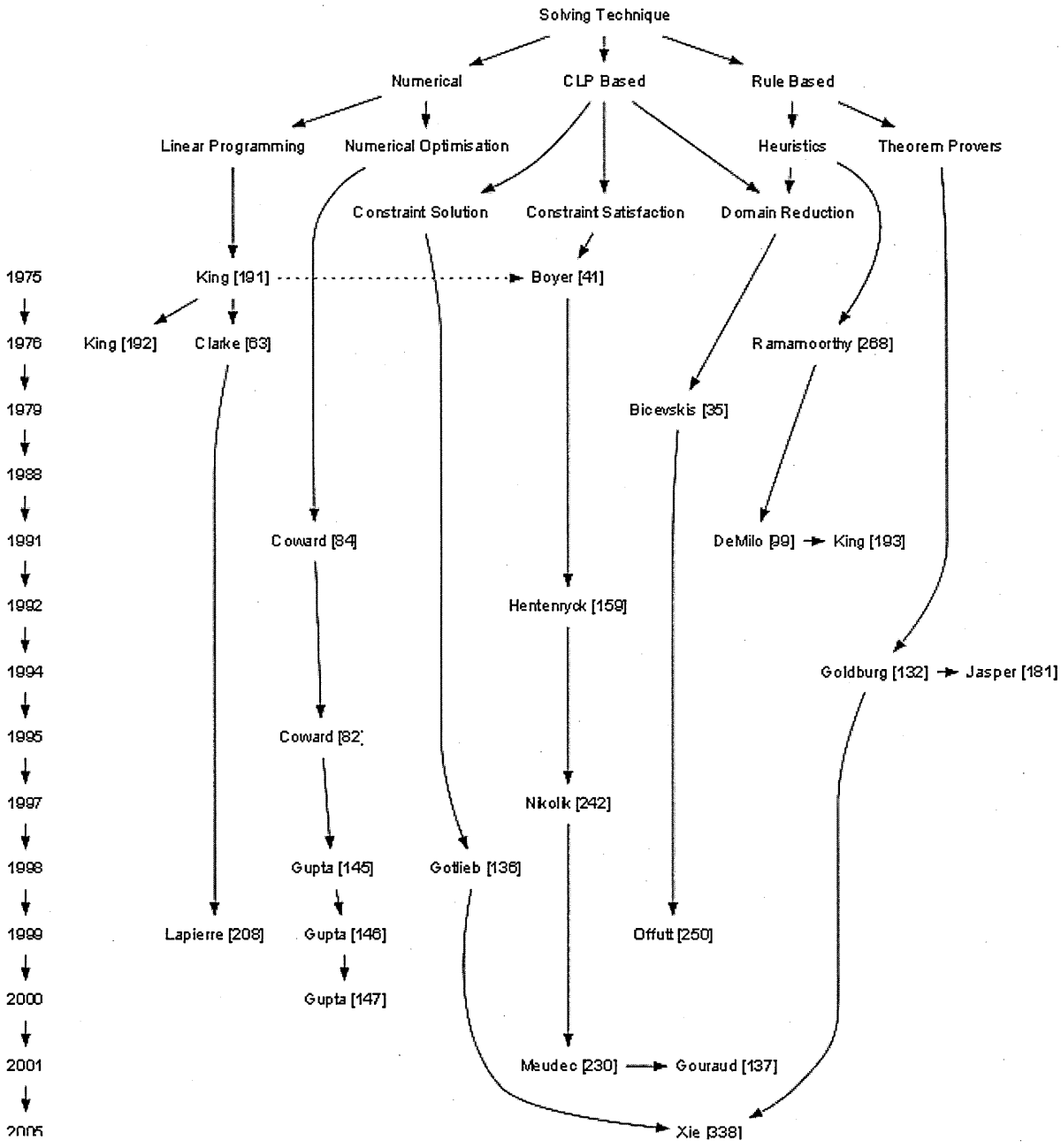

Fig. 11. Summary of the main techniques used in research to find test data.

Note that while the division given above is useful in considering how to organise the work and tease out trends and patterns, it is an artificial construct. For example, if we consider the subject of constraint programming as a whole then the numeric techniques can be considered as special purpose techniques for solving specific constraint problems for example, systems of linear equations. Constraint logic programming systems on the other hand are more general search systems. Indeed, Apt [20] notes that many constraint solving systems can be naturally characterised using a rule-based framework. Therefore, Figure 11 can be viewed as a ordering from specific to more general techniques.

2.5.4 Symbolic Testing: Summary

Symbol testing continues to make progress as witnessed by the replacement of specific techniques for solving numeric problems with techniques that are more general and with possibilities of hybridisation as shown by Godefroid *et al.* [131] with the DART system. The complete failure of random testing here shows the technique has promise. However problems remain, as noted by Lee *et al.* [209] and Xie *et al.* [338], complex data structures is one area for future research.

Early work by DeMillo and Offutt [99], [100] with mutated code suggests that it also strongly supports the idea that the technique has a high potential for error detection which is often missing in studies that concentrate solely on surrogates such as code or path coverage. Here of course the symbolic execution system has been given additional information, which is a set of target mutations to kill, or to differentiate from the original code. The provision of additional information appears to be a strong factor in the success of the symbolic execution technique, for example Clark [63] used additional constraints to detect certain types of errors. Likewise Godefroid *et al.* [131] provided additional information using co-execution.

This of course has to be tempered by the observations by Howden [169], [171] on the reliability of the technique. It should also be noted that symbolic testing is also constrained by some of the problems associated with path based testing in general, i.e., that as with code based adaptive techniques, Beizer's [31] comments on missing paths are also applicable. Howden [170] reports that the technique is partially unreliable for path-domain errors.

2.6 Opportunities for Further Research

The preceding sections have been a high level survey of the majority of work that has been done in the area of automatic test data generation, with the proviso that work based on formal methods has been excluded. This exclusion has been made because these methods are not widely used in an industrial setting.

In deciding where to perform further research, a number of factors need to be taken into account including the following:

- there has to be a clear path towards making a contribution to knowledge, an essential requirement for the program of study;
- it has to be practical to perform the work;
- ideally, the work performed has to have potentially immediate, useful and practical application.

The first of these requirements implies that it is possible to make a non-trivial contribution to the literature in the area where research is undertaken. For example, while adaptive testing is an attractive area to do research in, at the current time it is an intensively active research area. As part of a larger research group that had performed work in this area this would be an attractive proposition aside from the one main issue with the technique, that it is purely path following and as pointed out in section 2.4.4 not reliable for missing paths.

The second implies that the tools to perform the work need to be available at low cost or are able to be built within a reasonable period. This requirement for example makes it less desirable to work in an area such as symbolic execution. While it would be possible to build the required tool sets using Prolog and the ECLiPSe in a manner similar to Dillon and Meudec [104], it is unclear what this would achieve as Dillon and Meudec used essentially the same code base[24] as the work present in this thesis. While a different approach could be used, for example, using mutation adequacy rather than path coverage as an acceptance criterion it is not obvious that there would be sufficient novelty to differentiate any work carried out from either Meudec's work or from the earlier work of

[24] The author of this work made an early version of the Wallace code available to Dillon and Meudec for their study.

Offutt *et al.* [99], [100]. Thus while work on symbolic execution may be possible it potentially fails the first of the criteria set.

The third criterion given is purely practical. As a practicing engineer with an interest in software safety, I undertook this program of study to find a technique that could be applied to the systems that I oversee development of.

The discussion above has ruled out working in two of the major areas of automatic test data generation. Of the remaining areas, random testing has little to recommend it as a practical technique compared with hand-generated tests based on results from Frankl and Weiss [121], Reid [269], [270] Deason [97] and Michael *et al.* [232]. Likewise, the anti-random testing techniques proposed by Malaiya [219] have little to recommend them and the work on boundary following discussed in section 2.3.3 is attractive because it offers strong potential for being useful. However, the limited amount of published literature that has been located suggests that there may be some hidden pitfalls that are not readily apparent.

The area that does however appear to be open for some useful work is combinational *t*-way test sets generation. There are a number of reasons for this:

- work by Kuhn *et al.* [313]. [202], [203] on the analysis of real faults discovered in code, and the observation that for a relatively low factor very good results can be achieved in practice with automatically generated test sets;
- the limited amount of empirical studies performed is a clear indication that there are opportunities to make a strong contribution to the field;
- The technique itself meets the criterion set out at the start of this chapter (section 2.1) that it should be able to be used with information generally available using current software development techniques.

The next chapter therefore is a more detailed survey and analysis of the available literature on *t*-way adequate combinatorial techniques.

3. Combinatorial *t*-way Techniques

3.1 Introduction

The literature on combinatorial testing can be divided into two major areas: first research into techniques for generating *t*-way adequate test sets and work that evaluates the technique. The latter area, in turn, falls into two main categories: reports of the tools in field use and a small body of experimental work conducted under laboratory conditions.

A complicating factor with this classification scheme is that some work falls into more than one category. For example, Yilmaz, Cohen and Porter [341] could be classified either as field evaluation or as experimental work because although the paper describes an experiment, the authors lack control over certain aspects of the experimental design, specifically what faults are present and the details of the regression test sets used to expose those faults. Here I have taken the possibly pedantic view that to be classified as experimental work, *all* relevant aspects of the work have to be under the control of the researchers.

This chapter is divided into four main sections as follows:

- section 3.2 examines techniques used for generating *t*-way adequate test sets;
- section 3.3 looks at field evaluation of the techniques;
- section 3.4 examines detailed empirical work;
- section 3.5 considers in detail what weaknesses are present in the work reviewed in sections 3.3 and 3.4

As with the previous chapter, in each section there is a detailed review of the work conducted, which is followed by an analysis and finally a summary. It should be noted that this chapter does not describe *t*-way adequate test sets and section 2.3.4.1 should be consulted.

3.2 *t*-way Test Set Generation

3.2.1 *t*-way Generation: Detailed Review

Given that the focus of this thesis is not the generation of *t*-way adequate test sets this section examines only a subset of the work on test vector generation in detail. Indeed much

of the work is essentially derivative or has lead nowhere (for example Williams [333] work on integer programming). In practice, the dominant techniques for generating *t*-way adequate test sets are based around the principles presented by the AETG algorithm or the IPO algorithm. This is evidenced by the fact that tools are available for these systems. The AETG tool itself is provided as a web service by AETG[SM] Web [91] and Testcover from George Sherwood [283] provides a similar service. A tool based on IPO (FireEye[25]) is available from the web and latter work included in this thesis was performed using a tool called `jenny` [182], again freely available on the web.

The original methods used for generating *t*-way adequate test sets used orthogonal arrays, the simplest examples of which are Latin and Greco-Latin squares. These arrays can be readily sourced from both books, for example Diamond's book [101] contains appendices devoted to their enumeration. In addition, there are databases such as the one provided by the National Institute of Standards and Technology (NIST)[26] that list orthogonal and covering arrays. However, Williams and Probert [331] list some of the issues usually associated with employing such arrays for test data generation such as:

- not all factors have the same number of levels;
- not all parameters are independent;
- insufficient Latin squares exist, either because there are too few or no such squares or because there are more than $L + 1$ parameters where L is the number of values.

Given the above, Williams [330] detailed a method for building CA's from smaller sub-arrays including orthogonal arrays and reduced sections of those arrays as and from special arrays that they used to "fill" holes that are left over. The technique was implemented in the TConfig tool and Williams [330] provided experimental results that compared his technique with the IPO algorithm from Lei and Tai [212], [292]. Although the method showed an obvious time advantage, there appeared to be no significant gain in terms of the size of the covering array generated. A construction for a CA(15, 13, 3) from Williams [330] is shown in Table 7 where R(6,3,3,4) is a OA(n^2, n=1, n) , I(9,1) contains all ones

[25] Available from http://ranger.uta.edu/~ylei/fireeye/

[26] http://math.nist.gov/coveringarrays/

and is 9 rows by 1 column in size and N(6,3,1) is an array of the form (n^{2-n}, n, d) containing a n by d block of twos concatenated with an n by **d** block of threes etc.

Table 7. Example of the scheme for constructing a covering array from sub arrays from Williams [330].

OA(9,4,3)	OA(9,4,3)	OA(9,4,3)	I(9,1)
R(6,3,3,4)			N(6,3,1)

In general, the use of existing covering arrays has received less attention than algorithmic methods for generating the required covering arrays. The area that has received the most attention to date are algorithms that use greedy heuristics to generate the required array. Perhaps the most widely discussed of these algorithm is that used in the AETG tool that was described in a number of papers by Cohen *et al.* [68], [70], [66], [67]. The algorithm is a greedy search that attempts to maximise at each step the number of combinations covered by selection from a large set of randomly generated vectors. The algorithm for 2-way adequate test sets is outlined below.

```
Assume test vectors v1..vi-1 exist
UC is the set of all pairs of values not yet covered in the set
v1..vi-1
FOR N iterations DO
    a) select the variable and value included in most pairs of UC
    b) select remain variables in random order
    c) for the sequence in step b, select the value included in
       most pairs of UC
select as vi the vector that covers the most pairs
```

As stated, the algorithm is quite simple but this belies the complexity of directly implementing it. A major issue exists in step *c* where the set of uncovered pairs (UC) needs to be examined to find the most numerous unused value. Simple search strategies are unacceptably time consuming and practical solutions encode the pairs and use techniques such as perfect hashing. For example, the `jenny` tool [182] would appear to be an outgrowth of work in this area[27]. The algorithm as given is also incomplete. For example, again in step c, it does not specify how to resolve ties, i.e., where two or more values meet the criteria.

Another algorithm (or rather pair of algorithms) for generating covering arrays was developed by Lei and Tai [212], [292]. This algorithm which they named In Parameter

[27] Information gathered from reading the source code to the jenny program and from other comments on the web site.

Order (IPO) avoided the random component in the AETG algorithm. The strategy adopted was to generate a complete set of 2-way adequate tests for the first two parameters presented. This minimal test set was then extended one parameter at a time until all parameters and pairs were covered. The process of extending the original test set involved "growing" the original vector horizontally by expanding existing tests (horizontal growth) and selecting the new element to cover as many remaining pairs as possible. If the process of growing existing test vectors failed to cover all pairs then new vectors were selected to cover as many remaining pairs as possible (vertical growth). Results in the paper indicated that efficiency in terms of test cases generated and time required by the IPO tool was comparable with the AETG tool.

Recently, Lei *et al.* [211], [210] discuss the extension to the IPO 2-way algorithms to higher factors i.e. *t*-way adequate sets of test vectors. These papers were notable for the depth of discussion on the problem of being able to efficiently identify and remove from further consideration pairs, triples etc. As noted above, this issue appears to be the determining factor in the speed at which the algorithm can operate.

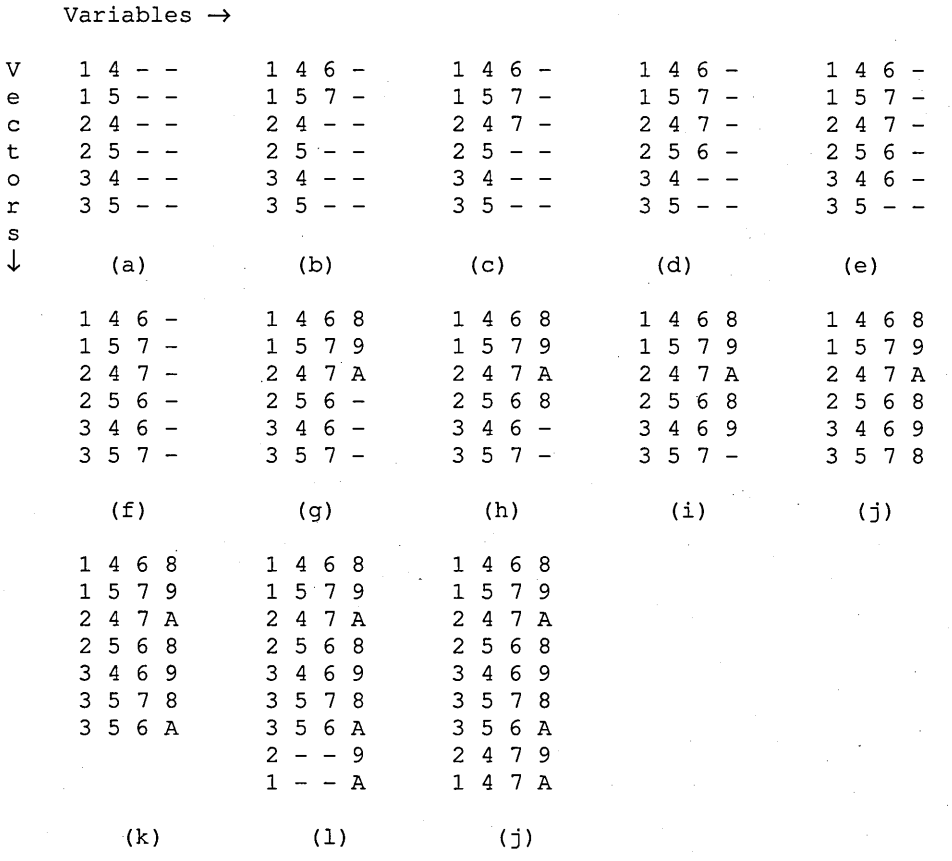

Fig. 12. Example of the IPO generation process for four variables with 3, 2, 2 and 3 values, (a) shows the initial state with all pairs for the first two parameters, (b) to (f) fill in values for the third parameter and (g) to (l) add values for the final parameter. All operations in (b) to (j) involve horizontal growth. In (k) to (l) vertical growth is used to give coverage for the remaining uncovered pairs for parameter one and parameter four.

3.2.2 t-way Generation: Analysis

The upper limit is represented by a test set generated by taking all combinations of n variables with v values taken t at a time where t is usually between two and six. There the number of pairs, triples etc. at the upper limit can be readily calculated. For example, the number of pairs is given by the following equation:

$$\sum_{i=1}^{i<N-1} \sum_{j=i+1}^{j<N} v_i \quad v_j$$

Where N is the number of variables and v_i and v_j are the number of distinct values that have been selected for use by variables i and j. Thus, if three values - for example, the minimum, the middle and the maximum had been selected for the first variable then v_i would be three. If the variable was an enumeration with five values, and all values were used then v_j would be five. Thus, for these two variables there would be 15 pairs. Extensions to higher factors are straightforward but it can be seen that the number of pairs, triples etc. that have to be dealt with grows quickly. However, not as quickly as all combinations, which requires that every combination of values be selected which is given by the following formula (Grindal *et al.* [139]);

$$\prod_{i=1}^{N} v_i$$

However, what is not obvious is the number of vectors that are required to cover all pairs etc. as each vector will cover multiple pairs. Cohen *et al.* [67] showed that the bound for the number of test cases N for a covering array of degree t is logarithmic in the number of parameters k, and provide a constructive proof for $t = 2$. Lei and Tai [212] proved that the problem of generating a *minimum* pairwise test set is NP-complete. Using curve fitting Tai and Lei [292] estimate that for systems with 10 variables and between 5 and 30 values (v = 5...30) for $t = 2$ the number of test vectors required to cover all pairs is $O(v^2)$ for their IPO strategy.

Colbourn, Cohen and Turban [79] presented a detailed analysis of the bound on the size N of the covering arrays provided by Cohen *et al.* [67] for the number of parameters k. To meet the logarithmic bound, they showed that it is necessary only to have a method that covers the average number of uncovered pairs. An algorithm was constructed (DDA) to

ensure this behaviour that was based on the "density" of uncovered pairs this being taken as a surrogate for expected number of new pairs to be covered.

There are a number of different variations on the basic AETG algorithm (Cohen *et al.* [67]). For example, Tung and Aldwan [305] described work that used similar principles to the AETG tool, but that had a lower reliance on random parameter and value selection. Specific differences included sorting parameters on the cardinality of the values, taking the largest value first rather than using random selection, and always selecting for the least used values when a tie in the number of new pairs covered occurred. However, in general this approach did not produce test sets with sizes that vary significantly from those produced by the AETG algorithm.

Bryce, Colbourn and Cohen [46] extended the work in [79] to build a general framework that encompassed a whole class of greedy generation algorithms including AETG [67], TCG [305] and DDA [79]. The framework was used to generate large numbers of CAs and MCAs with varying parameters in order to statistically investigate which features had the most effect on the size of the generated test set. The results indicated that it is the lower level decisions that had the most effect. In particular the value selection decisions (inner loop) and, to a lesser extent, the parameter selection criteria appear to be dominant.

Recent work that investigated extensions to the basic test generation process, notably the work led by Colbourn and Bryce [79], [43], [44], [45] on test generation prioritisation that used weights and the work by Cohen *et al.* [74], [73], [75] that added constraints to the test set generation process is of interest to this discussion. Notably they reused the basic AETG generation structure, extending it rather than replacing it. This suggests that, for the foreseeable future, AETG-like techniques may remain the dominant generation tools.

3.2.3 *t*-way Generation: Summary

Original research in this area appears to have been motivated by a desire to improve the testing processes. This has resulted in a number of workable, if less than perfect, algorithms and systems that create sets of vectors up to small *t* factors, e.g., AETG from Cohen *et al.* [67] and IPO and its latter derivatives by Lei *et al.* [211], [210]. Interesting work has also been done with other techniques such as metaheuristic search methods and the inclusion of several algorithms in one framework by Colbourn *et al.* [79]. However,

some of the work appears to have been conducted more from academic curiosity than practical necessity. For example, work with integer programming by Williams [333] and more recent work using SAT solvers by Hnich *et al.* [160], [161] and Yan and Zhang [340].

Research in this area has perhaps also been slightly blindsided in the drive to construct smaller sets of test vectors for factors of $t = 2$ and $t = 3$, and although improvements have been made they are often marginal. Given the early indication from Wallace and Kuhn [313] that higher factors, e.g., $t = 5$ or $t = 6$ and hence *much* larger tests may be required in practice, focus should perhaps have been directed sooner towards more "real world" issues such as those investigated in the latter work on weighting and constraints.

The work on weighting by Colbourn and Bryce [79], [43], [44], is interesting but needs further development. For instance, one of the stated primary goals is to generate test vectors that cover as many high *t*-way interactions as early as possible because this, it is assumed, will also reveal the most errors as early as possible. Although this reasoning appears to be sound and it is known that some errors require high factor tests, the assumption that generating these early will also result in the majority of errors also being discovered early is untested as yet and requires empirical evaluation.

The value of the extensions to greedy algorithms introduced by Cohen *et al.* [74], [73], [75] that incorporate hard constraints on what vectors can be legally generated is less ambiguous and represents a real contribution to the field.

3.3 Field Studies

3.3.1 Field Studies: Detail

Perkinson [263] studied several different strategies for testing an integrated services digital network (ISDN) system and described a number of techniques that were applied for generating effective test sets smaller than those required for exhaustive testing (181k test cases). Three methods are discussed briefly: defaults with user control where the tester selected additional paths and conditions, guided walk where a single parameter is automatically altered on each test , which has strong similarities to the base choice

technique defined by Ammann and Offutt [9], and the *proposed* use of orthogonal arrays. The last method is suggested to deal with a problem encountered with the other two techniques, namely that they were "*not robust in terms of parameter coverage*".

Brownlie, Prowse, and Phadke [42] performed system testing on the AT&T PMX/StarMAIL system using the Orthogonal Array Testing System (OATS) which is described in Harrel [158] and that built on work from Mandl [220]. They compared expended effort with the *expected* effort required for conventional testing where the test plan is constructed by hand. In terms of time, the authors suggested a 3:1 efficiency ratio in favour of OATS and in terms of faults detected they estimated that OATS was 2.6 times as efficient as conventional testing[28].

Unfortunately while interesting and suggestive these two papers (Perkinson [263] and *et al.* [42]) provide too little evidence to be really valuable.

Burroughs *et al.* [51] described a protocol testing application that used covering arrays generated using the Automatic Efficient Test Generator (AETG) tool developed by Cohen Dalal and Patton [68]. The paper compared the size of test set generated using the ATEG tool with test sets generated using two more traditional strategies and compares the 2-way coverage of interactions manually. The authors concluded that the modified AETG tests are superior in terns of breadth of coverage, i.e., that no significant holes were left. Interestingly, the authors reported that they chose to modify the AETG generated test sets to obtain a better balance, which seems to contradict their conclusions.

Cohen *et al.* [68] presented information on use of the AETG tool on two releases of production software where it is reported to have found more faults than standard test techniques. However, the researchers did not specify what the standard techniques were used so it is difficult to draw any strong conclusions. These results appear to be derived from earlier work (Cohen *et al.* [69]) that is covered in detail in section 3.4.2.

Two papers by Dalal *et al.* [93], [92], examined the use of the AETG system for high level, requirements based testing of production systems at Bellcore. In general the results were positive and for some of the failures detected the authors concluded that they would

[28] Unfortunately, in common with much on field testing exactly what "conventional" testing comprises is not explicitly defined.

only be revealed with certain combinations of factors. A large proportion of both of these papers were devoted to discussion of the issue of data modelling. That is of construction of models of the systems to be tested that are needed so the AETG tool can be used to generate the test sets. Data modelling appears to be a non-trivial issue with several of the models requiring several iterations to obtain a satisfactory result.

A number of advantages and weakness associated with the use of the AETG tool set were identified in these two papers. Major advantages were that:

- the models coupled tests to requirements [92];
- the ease with which the data models could be generated and iterated [92];
- the ability to regenerate models in response to change [92].

On the down side Dalal *et al.* also noted that there were issues with the following:

- that testers needed to have development skills and required domain knowledge to be effective [92][29];
- the development of test scaffolding dominated effort in at least some cases [93];
- there was an oracle problem, i.e., it was sometimes difficult to analysis the large amount of output generated, and that automated tests were in some case more difficult to understand than hand-crafted tests [92].

All of the above suggests that the human aspects of testing are still a significant issue and that the tools alone may not be sufficient without intelligence to guide them.

Burr and Young [50] reported the use of the AETG tool for testing an email system against requirements derived from a standard. Notably, the standard was expressed in terms of Backus-Naur Form (BNF) and the test process involved the translation from BNF to AETG constructs. Results were given in terms of percentage code coverage attained[30], approximately 93% for block (statement) branches. However, what is interesting in this work is how the coverage increased from only 50% block/branch coverage over the course of the project as the AETG data model was developed. A comparison between AETG

---

[29] However it should be noted that in the authors experience this is usually the case. The distinction here being possibly the use of a separate test department.

[30] Normalised for blocks that could be covered, much of the code proved unreachable as it was designed to deal with failures.

generated test sets and test sets generated via conventional techniques and default value testing (i.e. base choice [9]) is shown, with the latter two techniques attaining only 85% block and branch coverage.

Pan, Koopman and Siewiorek [258] described an experiment that applied *all* possible combinations of parameters to automatically test for robustness faults in POSIX API function calls on 15 different implementations. Faults were detected when the process hung requiring a task to be killed, or aborted which caused an abnormal termination (core dump). While the process is not directly applicable to less catastrophic faults, these results indicate that in some circumstances massive sets of tests can be applied. However, of particular interest to the work in this thesis is the finding that for the events being generated, single parameter failures accounted for over 80% of the failures observed. Unfortunately percentages for pair, triples etc., were not provided.

Smith *et al.* [287], [286] discussed the use of two combinatorial techniques for testing the Remote Agent eXperiment (RAX) planning system, which formed part of NASA's Deep Space 1 program. The two techniques used were 2-way adequate test sets and a technique they termed all-values, the second of these techniques again appears to be identical to the base choice technique from Ammann and Offutt [9]. The effectiveness of the two techniques is measured against the total set of faults discovered during development and is classified in four areas according to where the fault was located. In this work, the base choice technique outperformed 2-way adequate test sets in all areas, and by substantial margins. However, the test sets also seem to have been complementary in that they revealed different faults. For the convergence and correctness aspects of the RAX planner the authors reported that 88% of the total known errors were found using a combination of base choice and 2-way tests but that only 50% of the faults for the interface and engine control software were revealed by these techniques. The authors do not suggest a reason for this discrepancy.

Huller [173] discussed system testing to reduce both the time and cost of delivering product and *claimed* a 70% reduction over conventional quasi-exhaustive test suits generated by hand. However, they admitted that verification of this claim would be difficult in practice. The technique used was based on manually generating the required combinations and was a variant of Lei and Tai's [212], [292] IPO algorithm that was

optimised by including the parameters with the highest number of values first to reduce the number of vectors generated[31].

One set of "field studies" of special interest is a large series of studies all of which involve Richard Kuhn, that examined real world system failures and classified the variable interactions that caused the activation of the faults that lead to those failures. Wallace and Kuhn [313] looked at software failure modes in data collected by the Federal Drug Administration (FDA) that involved the recall of medical equipment over a 15 year period. They concluded that the majority of failures involved only two variables and that only a small number involved three or four and that, therefore 2-way adequate test sets would have detected the majority of the failures.

Kuhn and Reilly [202] examined the Mozilla and Apache open source projects using their bug tracking databases to determine the number of conditions required to trigger the fault. Finally Kuhn, Wallace and Gallo [203] performed the same analysis on a large distributed system being developed by NASA. In both these cases, the results mirrors those from the earlier FDA study, indicating that in practice, a "small" *t* factor of between four and six would have been required to reveal all the faults reported.

Table 8. Results for Kuhn and Reilly [202] and Kuhn *et al.* [203] showing the required *t*-way adequacy to locate all known faults. Data from the TCAS experiment, Kuhn and Okun [201] is in the last line for comparison (see section 3.4.1).

System Studied	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
Mozilla cumulative faults %	28	76	95	97	99	100
Apache cumulative faults %	41	70	89	96	96	100
NASA GSFC %	68	93	98	100	-	-
TCAS Experiment %	-	53	74	89	100	-

It should be realised, however, that the term small is relative. A small *t* factor can easily translate into a very large number of vectors. For instance Kuhn and Wallace [203] suggested that "*then a small multiple of 10,000 tests would be needed to cover all 5-tuples*" (pg. 420). Dealing with this number of tests is non-trivial. Kuhn, Lei and Kacker [204] suggested that the solution to this problem might be to use "formal" models of the system under test and a model checking paradigm as automated oracles. However, this leaves open

[31] Small scale experiments using IPO showed similar results.

the question of how to verify that the model is correct compared with the real world rather than just internally consistent.

The study presented by Yilmaz, Cohen and Porter [341] is interesting precisely because exhaustive testing was performed. The faults examined were associated with build options for the ACE+TAO open software toolkit for building distributed applications. The primary thrust of the work was to examine the effectiveness of building fault classification trees from failure information by comparing the performance of *t*-way adequate test sets with results from exhaustive testing. The researchers found that for the full set of faults 2-way sets performed poorly compared with exhaustive tests. However, for a reduced set of faults that were considered to correlate strongly with build options, *t*-way adequate sets performed almost as well, with higher factor sets becoming more accurate at classifying faults. Impressive time savings were observed, one day for 2-way test sets versus a year for running the exhaustive test. A comparison with randomly generated test sets was also performed. This found that *t*-way adequate test sets found slightly more failures than the randomly generated test sets, with less variability in the number of failures located and produced more reliable models, i.e., without extraneous features. The paper did not define what it meant by failure or how failures were recognised. However, earlier work by Memon *et al.* [229] suggests that failures were recognised by running a large set of existing regression tests written by hand.

Bell and Vouk [32] reported on the fault detection effectiveness of applying 2-way testing to two security products with known errors and found that the number of faults detected was strongly related to the amount of expert input used to define interactions between the parameters.

3.3.2 Field Studies: Analysis

Data from field studies does not present a consistent view of whether the combinatorial techniques for generating test sets are useful. Early studies suggested that efficiency gains or other advantages can be made with using the technique Perkinson [263], Brownlie *et al.* [42], Burroughs *et al.* [51] and Huller [173] but provided little in the way of hard evidence. Therefore, these studies can only be taken as an indication that further investigation would be worthwhile. Likewise, the studies by Cohen *et al.* [68], [70], [67] suggested that the

method might be effective but the results are somewhat scattered and the original technical report on which the work was based on gives a better view and is dealt with in section 3.4. Burr and Young [50] provided more concrete evidence of the utility of combinatorial testing *versus* the other methods examined but results were for coverage alone and not conclusive.

Perhaps more interesting are the results from Smith *et al.* [287], [286] that indicated that 2-way testing (pairwise) was not always effective. The poor performance of 2-way test verses other techniques, i.e., base-choice indicates that further work needs to be done.

In a similar vein to the generally negative assessment above is comments by Dalal *et al.* [93], [92] and Bell and Vouk [32] commented that the production of good data models is not a trivial task and that it requires expert input. This is counter to some advice proffered in recent books on testing such as Copeland [81] whose advice is to "*determine the number of choices for each variable*" (pg. 71) and Kaner, Bach and Pettichord [187] who advise that values be selected by domain partitioning. A more recent text by Ammann and Offutt [8] provides more complete advice and suggests several different ways in which the input domain can potentially be modelled.

The most impressive set of studies in this group for consistency of results are those that have determined the number of variables required to trigger an actual observed failure in the software. The series of studies involving Kuhn, Wallace, Reilly and Gallo [313], [202], [203] provide the best evidence available that supports the proposition that combinatorial techniques are a useful means of generating effective test data.

This idea is further supported both by the work of Pan, Koopman and Siewiorek [258], who observed that the vast majority of failures involved only a small set of values and, more strongly by Yilmaz, Cohen and Porter [341] who directly compared the effectiveness of both 2-way and 3-way adequate test sets against complete testing.

However, it needs to be noted that there is a major difference between the conclusions presented by Kuhn *et al.* and earlier work. Prior to these studies, all authors have examined pairwise (2-way) testing, yet the studies by Kuhn *et al.* suggest that we need to do more work than this in suggesting that examination of factors up to six may be necessary with a corresponding increase in the number of vectors generated.

3.3.3 Field Studies: Summary

The field studies examined (in particular, the early work) have one great weakness, and one great strength. The weakness is a lack of comparative data in the studies reported by Perkinson [263], Brownlie *et al.* [42], Burroughs *et al.* [51], Cohen *et al.* [68], Dalal *et al.* [93], [92] and Huller [173]. Latter work is far better in this regard. However, the early work did serve to raise the profile of combinatorial testing and interest in its application.

The strength of the work presented here is that it draws attention to the difficulty of applying combinatorial testing in a completely satisfactory manner, as high-lighted above in the comments from Dalal *et al.* [93], [92] and Bell and Vouk [32].

However, the work that really stands out is the body of work presented by Kuhn, Wallace, Reilly and Gallo [313], [202], [203] which puts practical bounds on what can potentially be achieved by applying *t*-way adequate testing *independently* of actually applying the technique.

3.4 Empirical Studies

3.4.1 Empirical Studies: Detail

The technical report by Cohen *et al.* [69] covered a substantial amount of ground, including three sets of experiments; testing data input screens to a database; testing a group of ten UNIX commands, and additional testing on database screens to study fault detection.

The first section tested three input screens and for one reported a comparison of coverage metrics for several generation techniques including 2-way and 3-way combinatorial generation, random generation, and a technique they termed "default testing" which appears identical to the base choice technique proposed by Ammann and Offutt [9]. Although complete results for only one screen are reported, all the techniques were capable of producing around 90% block coverage except for random testing. The results for Unix commands are shown in Table 9. As can be seen, these showed a slight advantage to 2-way tests over the base choice technique (shaded cells). It was stated in the conclusions that 3-way test sets showed no gains over the 2-way (pairwise) test data.

Table 9. Results for block and decision coverage for the ten UNIX commands experimented on in Cohen *et al.* [69] using 2-way (AETG) adequate tests and base choice test sets (BC).

Command	AETG block	BC block	Difference	AETG decision	BC decision	Difference
sort	95	86	9	86	75	11
basename	100	86	14	100	94	6
cb	96	86	10	89	89	0
comm	98	97	1	90	87	3
crypt	92	92	0	90	90	0
sleep	100	100	0	100	100	0
touch	86	71	15	81	68	13
tty	100	100	0	100	100	0
uniq	100	98	2	98	91	7
wc	100	79	21	91	63	28

Unfortunately, the work in Cohen *et al.* [69] was aimed at determining whether 2 and 3-way tests gave good coverage rather than at comparing techniques. Consequently, although reasonably good coverage was demonstrated nothing can be said about how well this compares against a conceptually simpler[32] technique such as random testing.

Dunietz *et al.* [106] compared the code coverage of random experimental designs without replacement with the coverage obtained from systematic designs with the same number of vectors. They concluded that for block coverage, low factor (i.e. 2 or 3) *t*-way designs could be effective *if* it was necessary to keep the number of tests to a minimum. Higher factor tests produced results which were more reliable but at the expense of executing far more tests. Furthermore, results for path coverage, probably a more complete indicator of test quality, strongly favoured higher *t* factors (4 or 5).

Nair *et al.* [239] investigated random testing without replacement and no partitioning versus partition based testing and showed that, in general partition testing should be more effective. The particular case of partition testing that they investigated - an application of *t*_way experimental design - showed that the probability of detecting a failure for simple random testing was significantly lower than with partition based techniques. It is interesting to note that this paper is sourced from the literature on statistics rather than from the literature on computer science. The authors pointed out that partition testing can

[32] This is not to suggest that actually constructing random test sets is simple from a implementation perspective. Both the data space can be difficult to deal with as reported by Bird and Munoz [36] and random number generation is far from trivial as reported by Wichmann and Hill [329].

be considered a case of stratified sampling and that they also wrote "*it is well-known in the statistical literature that stratified sampling enjoys many advantages over simple random sampling*"[33] (pg. 168).

Kobayashi *et al.* [197] examined the fault detecting ability of specification based (Weyuker, Goradia and Singh [322]), random, anti-random (Malaiya [219]) and t-way techniques when applied to the testing of logic predicates against mutations of those predicates. Specifications for logic predicates taken from Weyuker *et al.* [322] for the TACS II aircraft collision avoidance system contain between 5 and 14 variables. The authors concluded that 4-way tests were nearly as effective as specification techniques and better than both random and anti-random test sets. The authors also specifically noted that 2-way tests did not perform as well as expected or as well as reported in previous studies such as Dunietz *et al.* [106].

Grindal *et al.* [138] [139] examined the fault detecting power of a number of different combinatorial strategies including 1-way (each choice), base choice (a single factor experiment), pairwise (2-way) using the AETG algorithm and 2-way using orthogonal arrays. Their experimental work was performed on code seeded with hand-generated faults used in defect detection studies by Kamsties and Lott [186]. The data they obtained for branch coverage is consistent with other experimental results. However, after examining the data in detail, the authors concluded that code coverage methods might also need to be employed. As in [286], the authors found that the base choice technique performed as well as orthogonal arrays and 2-way adequate test sets in three out of five problems. However, it is interesting that no technique detected fewer than 90% of the detectable faults, which suggests that the target code was perhaps not ideal for the experiment.

---

[33] They also commented that "*for any given partitioning of the input domain, gains in efficiency can be achieved by judiciously choosing the test allocation scheme. The importance of doing this does not seem to be fully appreciated in the software testing literature*" (pg.168).

Table 10. Summary of the functions used by Grindal *et al.* [138] in testing combinatorial testing strategies.

Program	Functions	Lines of Code[34]	decisions	global	nesting
count	1	42	8	0	4
token	5	117	22	4	3
series	1	76	10	5	2
nametbl	15	215	21	5	3
ntree	9	193	34	0	3

Schroeder *et al.* [277] examined effectiveness in terms of code coverage for *t*-way versus "random selection" (actually a random design) with replacement on code with hand seeded faults. Although this experiment produced results that broadly support the results from other experimental work, each technique only detected between 45% and 55% of the injected faults. Furthermore, it was also found that values for *t* greater than four were required to reveal some faults. The researchers also concluded that *t*-way test sets were no more effective than test sets constructed using random designs for sets of the same size.

Kuhn and Okun [201] examined the almost ubiquitous TCAS program introduced by Hutchins *et al.* [174] and the ability of *t*-way adequate test sets to detect seeded faults. They found that there was an increase in the number of faults detected as *t* increased until $t = 5$. However, the size of the test sets reported, i.e., that $t = 5$ has 4200 vectors and $t = 6$ has 10902 - suggested that first, determining the correctness of a response could be difficult or perhaps impossible without the formal model that they used, and second that randomly selected values could have performed as well. However, whether randomly generated tests would be effective was not investigated.

Hoskins *et al.* [166], [167] investigated the ability of MCA covering arrays and D-optimal designs to approximate full factorial designs. A full factorial design is one "*in which every setting of every factor appears with every setting of every other*"[35] [12]. Note that here "every setting" refers to every selected level of a factor, not every possible level. D-optimal designs are algorithmically derived designs for specific models, that are

[34] It is not stated whether this total lines of code, or lines of executable code.

[35] Section 5.3.3.3 of the NIST e-Handbook of Statistical Methods [12]. Note that the electronic handbook does not contain page numbers.

commonly used when other experimental designs are not appropriate, and have a well established record. The authors concluded that covering arrays are "competitive" with D-optimal designs in approximating full factorial designs.

3.4.2 Empirical Studies: Analysis

Table 11 summarises the techniques that different empirical studies have applied to code when evaluating *t*-way techniques, a "y" in a cell indicates that that the technique specified in the first row was applied. Unfortunately, there is less overlap between the studies than first appears. For example, several authors reported the use of "random testing". However, Nair *et al.* [239] and Kobayashi *et al.* [197] use random testing without replacement where as Schroeder *et al.* [277] applied random designs from partitioned values without replacement and Grindal *et al.* [138] and Kuhn and Okun [201] ignored random testing completely.

Table 11. Summary of techniques that have been investigated to determine their fault revealing capability.

Study	Random	Anti-Random	1-way	Base choice	$t=2$	$t=3$	$t=4$	$t=5$	$t=6$
Cohen *et al.* [69]	y		y	y	y	y			
Dunietz *et al.* [106]					y	y	y	y	
Nair *et al.* [239]	y				y				
Kobayashi *et al.* [197]	y	y			y	y	y		
Grindal *et al.* [138]			y	y	y				
Schroeder *et al.* [277]	y				y	y	y		
Kuhn and Okun [201]					y	y	y	y	y

The level of factor coverage also varied quite widely between studies and although some have used high values (t >= 4), only Dunietz *et al.* [106] and Kuhn and Okun [201] have used values of five or greater. Given that examination of field data in field studies by Kuhn, Wallace, Reilly and Gallo [313], [202], [203] suggested that that values of $t = 5$ or $t = 6$ are necessary for the reliable discovery of all faults present, this is perhaps unfortunate

albeit completely understandable since the early empirical studies were conducted before these field studies were undertaken.

The data shown in Table 12 provides summary data for the subject programs used in the empirical work described above. As before, the first column is the reference for the study. The second column is a summary of the type of program that the study was undertaken on and the third column, "lines" provides an indication of the scale of the study and the column "factor" is the maximum level of interactions the test sets were adequate for. The column "block coverage" is the percentage block coverage reported. The column "faults" is the number of faults injected in the subject code for those studies that examined fault detection and the next column "detection" is the percentage of those faults that were detected or where reported, a range. The final column indicates the level at which the testing was conducted and is an educated guess based on comments as this was not explicitly stated in any of the papers.

Table 12. Summary of test subject features, for the main experiments reviewed in section 3.4.

Study	Examples	Lines	Factor	Vectors	Block Coverage	Faults	Detection	Application level
Cohen *et al.* [69]	3 "screens"	not stated	2-way	200	91 – 92%	n/a	n/a	module?
	10 Unix commands	total > 800	2-way	not stated	92 - 100%	n/a	n/a	program
Dunietz *et al.* [106]	5 C/C++ functions database access	not stated	random designs	range 0 - 150	0 – 100%	n/a	n/a	unknown
Nair *et al.* [239]	1 "screen"	not stated	2-way	not stated	not stated	n/a	n/a	module?
Kobayashi *et al.* [197]	20 predicates	n/a	4-way	not stated	n/a	33 - 327	44 -100% mean 81%	predicate?
Grindal *et al.* [138]	5 programs (Unix commands)	453		12 - 64	> 80%	128	98-100%	program
Schroeder *et al.* [277]	2 programs (part of)	> 1000	4-way	364 - 370	not stated	82	58%	program
		> 1000	4-way	375 - 384	not stated	88	51%	program
Kuhn and Okun [201]	1 program	138	6-way	> 10,000	not stated	41	100%	program

Notes:

- The number of lines reported for Schroeder *et al.* [277] is an estimate.
- Block coverage for Grindal *et al.* [138] is estimated from the decision coverage reported.
- The line count for Kuhn and Okun [201] is taken from Rothermel *et al.* [273] but the value is consistent with Jones and Harrold [185] and Andrews *et al.* [11] who used the same TCAS program

Table 12 shows several interesting features that will be discussed further in the following sections. First, it shows how the emphasis has changed over time from demonstrating coverage to measuring fault detection. Second, there is less variety in the programs being tested than is desirable. Finally, although the coverage results are good and consistent, the data on fault detection are more ambiguous and in addition, given the third point, an additional subsection that considers the nature of the faults being tested and, finally, a subsection on program size.

3.4.2.1 Emphasis

The emphasis in empirical testing work has clearly changed over time, from demonstrating that the technique can achieve good coverage to showing that it is effective at detecting faults.

The principle reason for wanting to demonstrate that good coverage can be achieved is the *a priori* assumption that good coverage is indicative of good fault detecting ability. This assumption is supported by empirical work from Piwowarski, Ohba and Caruso [265] and from Wong *et al.* [335]. However, it needs to be kept in mind that although coverage is a necessary condition for fault detection, it is not a sufficient condition as observed by Weyuker [324].

If the view that coverage begets good fault detection were correct in an absolute sense then one could expect that for software tested to the highest coverage criteria levels we would see no errors. For example, avionics software which is covered by the DO-178B standard [14], requires MCDC coverage (Chilenski and Miller [60]) at the highest levels. However, despite claims of its effectiveness, there are more sceptical evaluations such as Bhansali [34] and we do in fact see faults in software in flight critical software (i.e. Class A) as noted by Shooman [285].

From this perspective, the shift to directly examining fault detection rather than coverage is welcome.

3.4.2.2 Variety

Another weakness in the empirical research conducted to date is that there appears to be very little variety in the types of programs tested. For example, Cohen *et al.* [69], Dunietz

et al. [106] and Nair *et al.* [239] all examine data input processing "screens". It seems plausible that these may have all been taken from the same or at least similar systems as all of these papers have close ties with AT&T and Bell Labs, as does much of the work in this field. Unix commands or Unix-like commands are studied in both Cohen *et al.* [69] and Grindal *et al.* [138], and both Kobayashi *et al.* [197] and Kuhn and Okun [201] have based their work on the TCAS system, although Kobayashi *et al.* concentrated on the specification and Kuhn and Okun on actual code. Schroeder *et al.* [277] broke this pattern and used two "production" programs that they had direct access to. However, they only tested a sub-section of each program.

3.4.2.3 Coverage and Detection

Although good coverage results (e.g. greater than 80% blocks) seem to be universally reported, there is less consistency in the results for fault detection. Some authors - for example, Kuhn and Okun [201], Grindal *et al.* [138] - have obtained good results on fault detection, and Smith *et al.* [286] reported one set results consistent with these two afore mentioned experiments. Other authors have seen less success, i.e., Schroeder *et al.* [277] reported that only 50% to 60% of faults were detected. Results from Kobayashi *et al.* [197] are mixed, ranging from approximately[36] 44 to 100% fault detection with the mean around 81%. Smith *et al.* [286] also reported poor results with one particular class of fault, interface faults. Here their results were consistent with those reported in Schroeder *et al.* [277].

3.4.2.4 Faults

There are several confounding aspects to those studies based around fault detection that have reported good results. As stated above, Kuhn and Okun [201] offered no comparison to the technique that they used (i.e. there was no control) so it is open to speculation whether any other technique, e.g., random testing, would perform as well. Similarly, Grindal *et al.* [138] reported that *all* the techniques reported good results so the detection of defects on its own is in itself not good evidence that the technique is effective. It is

[36] Figures were estimated from the histograms presented in the paper.

possible that in this case at least the faults may have been too simple to truly stress the techniques applied.

We also have to consider the possibility that the sets of faults examined may not be consistent with real faults or that they are distributed in a different manner to real faults. The majority of studies inserted faults by hand, i.e., Kuhn and Okun [201], Schroeder *et al.* [277] and Grindal *et al.* [138] who took the existing set of faults from Kamsties and Lott [186] and added additional "mutation like" faults by hand. Only Kobayashi *et al.* [197] used a systematic technique, namely mutation. Interestingly, these authors observed different fault detection rates to the study by Kuhn and Okun [201] for similar "code".

Whether or not this variation in how faults are inserted introduces a bias in the results has been investigated by Andrews, Briand and Labiche [11] who compared the fault detecting ability of existing test set suites against both code with hand inserted faults and code that had faults inserted by a mutation tool. In general, they found that automatically generated code mutants tend to be easier to detect than hand seeded faults. For the one program that had a set of known actual faults there was little difference in the difficulty of detection between these and faults introduced by mutation.

Closely associated with how faults have been inserted, is the number of faults that have been included. For example, Kuhn and Okun [201] used 41 faults (versions) in the TCAS program whereas Kobayashi *et al.* [197] used 327 for one single predicate from the TCAS specification. Indeed the contrast between the success rates reported by these two groups on similar problems[37] raises some doubts on general applicability of the results in Kuhn and Okun [201]. Even taking into account that fact that Kuhn and Okun used higher factor tests than Kobayashi *et al.* [197] (6-way *versus* 4-way) this does not completely account for the differences between the two sets of results. The same observation applies to number of faults used in Schroeder *et al.* [277] and Grindal *et al.* [138] where only a small percentage of the number of possible faults were examined.

[37] It should be noted that Kuhn and Okun used the TCAS program code whereas Kobayashi *et al.* used the specification from TCAS II which includes extra logic for conflict resolution not present in TCAS I specification. Therefore, it is possible that the TCAS II study considered complex predicates not present in the TCAS study.

3.4.2.5 Program Size

Related to the discussion above is the level at which the techniques were applied. Kuhn and Okun [201] performed the testing at the level of the program, i.e., by manipulating the external interface in contrast to Kobayashi *et al.* [197] who must have effectively worked at the level of a single function in order to isolate each of the predicates that they were investigating, although they did not state this explicitly.

Likewise, Schroeder *et al.* [277] and Grindal *et al.* [138] performed their investigations also apparently at the program level. This may have been a deliberate attempt to study the effectiveness of the technique in a black box environment or might perhaps indicate that the information necessary to perform testing at a lower level was not available.

Importantly however, this does not change the fact that testing at the lowest level, often referred to as unit testing [13] - allows more control of the environment and greater access to the results of the test process. Freedman [127] investigated the problem of how testable software was and observed that two factors have a large bearing on this matter. They are:

- observability: the ease of determining whether specified inputs affect the outputs;
- controllability: the ease of producing a specified output from a specified input.

One area that touches these issues is the use of static data such as counters, timers etc. especially when this data is hidden from external view. Baresel *et al.* [24], Lammerman *et al.* [206], Gross *et al.* [143], [144], as well as McMinn and Holcombe [226], [228], have all noted that variables declared as static in the C language presents a problem for evolutionary test generation. Primarily because if the static data is embedded with the function under test, the variables are not controllable nor in particular observable.

3.4.3 Empirical Studies: Summary

As noted by Tichy *et al.* [298], there is in general a lack of empirical work in computer science and, at this point, Tichy's [297] plea for more experimental investigation only needs to be repeated. There are too few experimental results on too few different test subjects with too few sets of comparative data to be able to definitely know whether combinatorial techniques will live up to its promise of being effective at detecting software errors.

3.5 Weaknesses

If we examine the field studies described in section 3.3 and the empirical investigations detailed in section 3.4, we find that the following points stand out:

- Direct comparisons with other techniques are either absent or are not consistent (for example the comparisons with randomly generated tested discussed in sections 2.2.1 and 3.4.2).
- The comparison with human testing although present is flawed in that too few details are presented on how testing was performed, how test sets were selected and what adequacy criteria were meet, e.g. statement coverage.
- Also in relation to human generated tests, it has yet to be shown that the technique is least as good as good as a human tester, a necessity if it is to be applied in critical applications.
- Less attention than desirable has been given to the oracle problem. Although the technique may be able to detect errors, but for factors greater than two it also generates a large number of tests. Some method of reducing the number of tests that need to be examined by people must, therefore, be developed.
- In addition, there have been very few attempts to estimate the significance of the research in any formal sense for example by using statistical hypothesis testing.

3.5.1 Comparisons

The evidence on the utility of the technique that is presented in section 3.3 is especially weak in the early work on software testing; with the studies with the least comparative information being highlighted in section 3.3.3.

The number of formal (i.e. measured) comparisons with other test data generation techniques is minimal. As stated in section 3.4.2, some empirical work has made comparisons with random testing, i.e., Cohen *et al.* [69], Nair *et al.* [239], and Kobayashi *et al.* [197] but particular comparison has not been universally applied. It is especially disappointing that the two studies by Grindal *et al.* [138] and Kuhn and Okun [201] did not include it particularly since Ince [175] suggested some twenty years previously that random testing be used as base method for comparison with other more complex techniques in experimental studies. Indeed, random testing has been used widely as a

comparator in empirical studies outside this area such as Frankl and Weiss [125], [121] and in the large study by Hutchins *et al.* [174] which continues to form the cornerstone of much empirical work in testing research such as that performed by Reid [269], [270].

A number of field studies including Perkinson [263], Burr and Young [50], Smith *et al.* [287], [286] and empirical work by Cohen *et al.* [69] and Grindal *et al.* [138] [139] have also investigated the use of single factor experiments as a technique. However, this group of authors came to different conclusions. Those who conducted field studies generally found that the technique was less effective than suggested by the empirical study performed by Grindal *et al.* [139]. It should be noted however, that the results of the field studies performed by Smith *et al.* [287], [286] tended to agree with Grindal *et al.* [139] rather than with the other field studies. It would therefore seem profitable to include the base choice generation technique in any comparative work undertaken.

3.5.2 Human Test Sets

The purpose of automating the test case generation process is to remove or reduce the need for human intervention. However, this is only useful if the tests generated without human intervention are at least as effective as test sets for the same code generated by human testers. Although the field studies suggest that *t*-way adequate test sets may fulfil this criterion, the evidence is by no means conclusive. Although the studies reported by Dalal *et al.* [93], [92] state that more errors were found using *t*-way adequate techniques, they provide virtually no information on exactly what the effectiveness of the technique is being compared with. Later studies such as that by Yilmaz, Cohen and Porter [341] are better but they are still lax in detailing exactly what is being compared with what and, just as importantly, in detailing *how* errors are being detected.

To gauge how useful the technique is in practice it needs to be directly compared with human generated tests. However, as noted by Ellims, Bridges and Ince [112] very few studies have been performed that examine human testing performance. Fewer still directly compare performance of humans versus automated techniques directly. Indeed the only study known to the author is by Grochtmann *et al.* [141]. Here test sets generated by students were compared directly with path following genetic algorithms to determine the maximum execution time of several procedures.

3.5.3 Test Reduction

The primary motivation for using *t*-way adequate tests, at least initially, was *improved* testing with *smaller* test sets. Although the second feature may be true of 2-way adequate test sets, as noted in section 3.3.1 the number of tests required for higher factors is very large, at least relative to what would or could be generated by hand.

By itself needing such a large number of tests is not a problem except that they may take an unreasonable time to execute. However as pointed out by Ould [257] the major problem with testing is determining whether a test passed. That is, whether the software under test produced a correct result or not. Therefore for the technique to succeed there needs to be either an oracle such as the formal models proposed by Kuhn *et al.* [204] that can determine whether a result is correct *or* there has to be a mechanism for determining the quality of a test vector so that a small set of vectors can be selected from the larger set. If this can be done then the number of vectors that need to be examined by hand can be minimised.

A mechanism that can do this needs to have the property that it can distinguish a good test case from an average or perhaps even a bad test case. The usual metrics used to measure code coverage such as statement and branch adequacy are in some ways inadequate for the purpose. As noted by Weyuker [324], these criteria are necessary but not sufficient. Coverage of all lines of code does not imply that code has been fully tested and it is trivial to provide examples where this would be true. The most obvious case would be straight line code that performed large amounts of possibly complex arithmetic. A single test vector would be sufficient to provide coverage but not actually do anything very useful, a point addressed by Hamlet on several occasions [149], [151].

3.6 Final Appraisal

It is reasonably clear that little opportunity exists for the development of new tools for generating *t-way* adequate test sets. Although some effort was put into this activity and an adequate tool for generating 2-way (pairwise) test sets was created, activity in this area was overtaken by events. More efficient tools have since become available including the `jenny` program [182] and tool sets from the research conducted by Lei *et al.* [211], [210] have become more widely available (Kuhn *et al.* [204]).

Therefore, the research in this thesis will focus on the effectiveness of *t*-way combinatorial test sets for detecting errors in code and on finding an efficient mechanism to make the technique tractable without resorting to formal models as suggested by Kuhn *et al.* [204]. The desire to avoid the use of formal models are derived from the fact that these techniques are not currently widely used in industry. There also remains the question of testing the model itself, as while it may be possible to "prove" that it is complete against some criteria there is no way of proving that it actually does what is intended.

For example, it may be necessary to convert a model or other formal definition of a program back into English so that the end customer can understand it[38]. Likewise, the end customer may use other methods to define functionality. The example is given by McDermid *et al.* [221] where the end customer may use their own sets of formal, or as is more usual, semi-formal methods that are not compatible, which presents a two way translation problem.

[38] Personal Communication from Professor Martyn Thomas at 9th Australian Workshop on Safety Critical Systems and Software.

4. Program of Work

4.1 Introduction

The research presented here focuses on determining the effectiveness of combinatorial test detecting errors in code and on finding an efficient mechanism to make the technique tractable without resorting to formal models as suggested by Kuhn *et al.* [204]. This second aspect of the research is important because in practice, as formal models are not currently widely used in industry at present, there remains the question of testing the model as while it may be possible to "prove" that it is complete against some criteria, there is no way of proving that it actually does what is intended.

For example, it may be necessary to convert a model or other formal definition of a program back into English so that it can be understood by the end customer. Likewise, the end customer may use other methods to define functionality. An example is given by McDermid *et al.* [221] in which the end customer may use their own sets of formal methods or, as is more usual, semi-formal methods that are not compatible, which presents a two way translation problem.

The aim of the research to be conducted is twofold. First to determine whether *t*-way adequate test sets are a "reliable" method of automatically generating test sets and second to determine whether test set minimisation can be used to reduce the number of tests that need to be considered by a human examining the output for correctness to a manageable level. Here, the term "reliable" is used in the sense that what is wanted is a method that can be expected to have a good chance of detecting errors.

To directly address the weaknesses raised in section 3.5 I want to cover the following points:

- I want to provide a good comparison with other automatic generation techniques;
- I want to be able to compare these results with human generated tests;
- I want a method that can determine whether an individual test vector is good and therefore worth pursuing to make the oracle problem manageable.

The first point is derived directly from observations made in section 3.4.2 and section 3.5.1 where it was noted that one of the features of empirical work is the lack of comparison with other techniques, with the exception of the studies performed by

Kobayashi *et al.* [197]. The critical point being that any technique for automatically generating test data has to be shown to be more effective than other, possibly less expensive, techniques for generating data, e.g., random testing. In the case of *t*-way adequate techniques this has not yet been done.

The second point comes from a desire to be able to use automatically generated tests in both safety related and safety critical applications. However, if automatically generated tests are to be used in those situations, they have to prove themselves at least as effective as the human generated tests that they are intended to replace. The ideal situation would of course be that they were shown to be superior. This is a central issue in being able to use automatically generated tests, in practice, however there is virtually no literature that addresses this point in a completely satisfactory manner[39]. The field studies examined in section 3.3 provide some evidence that *t*-way adequate tests may be as effective as human generated tests but as pointed out in section 3.3.2 many of these are flawed in that they provide too little information on techniques that *t*-way adequate tests are being compared with.

The third point raised above is derived directly from the need to be able to deal with potentially thousands of test vectors that *t*-way adequate test sets can contain. Other methods may exist, e.g., the use of formal models by Kuhn and Okun [201]. However, this begs the question of why, if a formal model exists, why do we not then directly derive the code from it? Other possibilities for addressing the oracle problem, for example Kuhn has suggested assertions embedded within the code could be used[40]. However, if we consider the current state of practice in industry now, it is probable that for some time "the oracle" will be the engineer performing the testing, hence there is need to reduce the number of vectors to a manageable minimum number.

The remainder of this chapter is set out as follows:

- Section 4.2 briefly covers the type of work to be performed and looks at the techniques that are available to perform that work;

---

[39] The only study that the author is aware of that addresses this issue directly is [141] which examined temporal correctness, used students not professional programmers but did include random testing and was performed on industrial code.

[40] Personal communication, December 2008.

- Section 4.3 looks what I am attempting to achieve in a more formal manner and puts forward the hypotheses to be tested in this thesis.

4.2 Foundation work

4.2.1 Problem Overview

If we take the three specific points brought out in the introduction above, then it seems clear that to deal with the first point any work that compares different test generation techniques to be acceptable, it should be compared directly with other techniques. The primary method of interest that all work such as this should be compared with is random test generation, in its simplest form, i.e., without replacement. Given the work performed in other empirical studies the candidates for this thesis that stand out are random designs as used by Schroeder *et al.* [277] and the base choice technique as proposed by Ammann and Offutt [9] and used in the study by Grindal *et al.* [138].

The second point can, in turn, be addressed in two ways: by either taking a body of available code and defining a set of "good" hand-generated vectors for it or by locating such a set of code that already has such test vectors. Given the authors position in industry, the second option is available and has been used in this study. To the best of the author's knowledge this is a unique situation.

The final point in section 4.1 has two components The first is to define what we mean by "good" and the secondly is to extract those "good" tests in such a manner as to avoid reducing the overall error detection ability of the test set. Given the potential weakness of code coverage as a measure of a test vectors ability to detect errors and the potential number of vectors that are candidates for inclusion in the final test set, it was decided that a more rigorous measure of a test vectors error detection ability was required. The current best candidate for determining the goodness of a test appears to be mutation adequacy and evidence for its effectiveness is detailed in section 4.3.2.

The second part of the problem, as stated above, is how to reduce the size of the test sets without sacrificing quality; section 4.3.3 is a brief overview of work that has been performed in the area of test suite reduction or minimisation.

4.2.2 Mutation

In section 3.4.2.1 it was noted that there had been a marked shift in empirical studies from using coverage criteria as a measure of effectiveness to studies that directly examined the error detection potential of the technique. To date all studies that have used error detection as a criteria have used hand seeded faults. In the study performed by Grindal *et al.* [138] it was noted that mutation like hand seeded faults were used. The obvious extension to this is not to use hand-seeded faults, especially those that mimic code mutation, but rather to use code mutation directly.

Code mutation as a technique appears to be a good candidate for investigating the fault detection properties of *t*-way adequate test sets. At the least, it can be used to compare one test set against another with perhaps greater fidelity and a higher level of discrimination than metrics based purely on the code structure such as statement and branch coverage.

Ould [257] argued that an automatically generated set of test vectors is only effective if it generates test cases that are likely to expose errors. Therefore the critical question is, does code mutation meet this criteria?

Offutt *et al.* [246], [247] performed a set of experiments which used test sets that were adequate for revealing single mutation faults on programs that contained double mutation faults on the same execution path. They concluded that, for the programs studied, that test sets that were adequate for revealing simple faults (i.e. single mutations) were also adequate for detecting more complex faults, i.e., high order mutants (mutant of mutants).

However, a study by Frankl, Weiss and Hu [121] performed a similar experiment and reached a different conclusion. Namely that compound mutations are not a good model for faults. However, their results are less clear cut than those of Offutt and 50% of their results did demonstrate good coupling. Moreover, their data show that in five out of ten of the subject programs, mutation adequate test sets performed better than all-uses adequate test sets. In another two cases, they worked as well. These findings suggest that mutation is at least no worse than all-uses as an adequacy criteria.

To further look at this question, the seminal work performed by Andrews, Briand and Labiche [11] used the large set of programs and associated test vectors from Hutchins *et al.* [174], which have hand-inserted faults and the `space` program developed by the European Space Agency, which contains a known set of *real* faults. They found that mutants do not

appear to be either easier or harder to detect than real faults. This is in contrast to hand seeded faults[41], which were found to be harder to detect than mutants. Andrews, Briand and Labiche [11] concluded that "*mutants, based on the mutation operators presented here, do provide test effectiveness results that are representative of real faults*" (pg. 8).

4.2.3 Optimisation and Minimisation

Optimisation of test sets comprises two related areas: test case prioritisation and test case minimisation. Test case prioritisation is used to schedule or order test case execution to maximise some property such as the rate of fault detection (Rothermel *et al.* [274]) and is usually discussed in term of minimising the cost of performing regression testing. By contrast, minimisation attempts to find the smallest possible set of tests that meet some criteria such as maintaining statement or branch coverage.

Prioritisation has seen a reasonable amount of work over time with notable recent contributions by Rothermel *et al.* [274], Rothermel, Untch and Chu [275] and Jones and Harrold [185], who also addressed the minimisation problem. However, as the emphasis here is on reducing the cost of performing regression testing, prioritisation is of only passing interest to the problem at hand.

To deal with the potentially large number of test cases that can be generated and to make the oracle problem tractable we want to be able to reduce to a *practical* minimum the number of test cases that have to be examined. Note that the size of the test case set does not have to be as small as theoretically possible, just small enough that it becomes tractable.

Test set minimisation has received minimal attention in terms of automatic test set generation although an initial investigation conducted by Ince and Hekmatpour [177] that used randomly generated test sets and statement coverage adequacy demonstrated some initial success by reducing the number of test cases that needed to be examined manually.

---

[41] A code faults are of course hand seeded, how ever the difference here is the difference to those faults inserted by stupidity versus those inserted by malice (with apologies to Winston Churchill). Andrews concluded that faults inserted deliberately by hand were more difficult to detect than those occurring as part of writing the code.

Much[42] of the other work performed in this area has examined how minimisation affects the fault detecting ability of the test cases. Wong *et al.* [335] concluded that "*there is little or no reduction in its fault detection effectiveness*" (pg. 368) for more demanding criteria. However, Rothermel *et al.* [273] reached the opposite conclusion, stating that minimisation can compromise the effectiveness of a set of tests where all-edges is used as the adequacy criteria. The conclusions from these two studies are not exactly comparable, because unfortunately they both used different adequacy criteria. A study by Jones and Harrold [185] investigated two algorithms for test set reduction (minimisation) for the MC/DC criteria and found that the fault detecting ability of the minimised test set could be reduced depending on the program and nature of the faults present. Interestingly, the prioritisation algorithm presented in [185] closely mirrored the selection algorithm used by Sherwood [281], [282] for generating combinatorial data sets.

There are several problems with using adequacy criteria such as code coverage for minimisation. The primary objection being that what constitutes an adequate test set does not have a good theoretical underpinning, moreover most simple adequacy criteria, such as code coverage, are recognised as being inadequate, they are *minimum* criteria. While it can be shown that some conditions are necessary, for example, statement coverage (Weyuker [324]) it has also been shown by Howden [169] that no method can be considered reliable in any absolute sense. In more practical terms Hutchins *et al.* [174] in their investigation of the benefits of data and control flow adequacy criteria concluded that "*code coverage alone is not a reliable indicator of the effectiveness of a test set*" (pg. 191). Therefore, it is unlikely that code coverage adequacy criteria alone are an adequate indicator and thus, by implication, are not the best target for minimisation.

The alternative to code coverage metrics is to use a fault-based strategy that allows fault detection capability of a vector to be measured directly. Two such strategies have been suggested: code mutation as proposed by Hamlet [154] and DeMillo *et al.* [98] and fault injection as proposed by Voas and McGraw [309]. Of these, code mutation has received the majority of the attention to date and has been widely used in studies that compare the effectiveness of test sets, for example by Daran and Thevenod-Fosse [95], Frankl, Weiss

[42] This section does not comprise a complete review of the literature.

and Hu [121], as well as Zhan and Clark [345]. Code mutation also has the advantage that it subsumes some conditional coverage techniques (Offutt and Voas [253]) such as statement and branch coverage.

In addition, some work has been done with minimisation of mutation adequate test sets. An early suggestion by Offutt [248] was simply to ignore vectors that did not kill any mutants. In later work Offutt, Pan and Voas [255] suggested a mechanism for selecting minimal sets of vectors that again removes mutants as they are killed but runs the set of vectors in different orders.

To the authors knowledge, no work has been conducted on minimising complete *t*-way adequate test sets. In the area of combinatorial test sets, the work that comes closest is Dadeau, Ledrun and Du Bousquet [88] in which the researchers minimised the test set via selective pruning of the search tree. Their results are not applicable here because the combinatorial technique they used was not *t*-way adequate and was aimed at producing sequences of function calls and associated input data.

The work on weighting by Colbourn and Bryce [79], [43], [44] is related to the problem considered here but was targeted at situations where it may not be possible to run a complete set of tests. The situation they had in mind was the testing of different configurations. However, as noted in section 3.2.3, no empirical work validating this approach has yet been performed.

4.3 Hypothesis

As stated in the introduction to this chapter there are two primary goals of the empirical work presented here:

- to determine if *t*-way adequate test sets are "reliable";
- to determine if we can reduce the size of the final test set, and hence the size of the oracle problem.

The first of the objectives can be broken down into two sub-goals. The first sub-goal is to determine whether 2-way adequate test sets are a reliable method for generating test sets, thus testing assertions by authors such as Burroughs *et al.* [51], Cohen *et al.* [68], [70], [67] and Huller [173] that this is the case. The second sub-goal is to test the conclusions from the field study work by Kuhn *et al.* [313], [202], [203] and from the

empirical results from Kuhn and Okun [201] that only small factors, i.e., less than or equal to six are required to produce good test sets.

To achieve the aims of this research, it is proposed to evaluate automatically generated *t*-way adequate test sets against a set of code and associated unit (C function) tests from a safety-related application that has undergone multiple levels of test and review and that has been in field use for nearly ten years with no reported errors[43]. The set code to be used has the required unit tests already in place and it is believed that these are of high quality compared with the current industrial state of the art That is, unit tests have been reviewed and have had their code coverage measured directly. The development process for the code and its associated unit tests is covered more fully in section 5.2.2, 5.2.3 and 5.2.4. the remainder of this section coverss the second and third points from section 4.2.

To address the first point from section 4.2 and to provide a comparison with other work it is also proposed that the fault detection ability of the test sets will be evaluated against the following other test generation methods:

- random testing without replacement;
- random designs with the same input ranges, and;
- a single factor experiment, i.e., base choice.

The comparison against a set of human generated tests that are believed to be of reasonably high quality will give an indication of whether the *t*-way adequate techniques can reliably be used as a replacement for that test activity.

Random test sets are of interest because, as suggested by Ince [175], they form a base method for comparison with other more complex techniques in experimental studies. Test generation methods have to do at least as well as randomly generated test sets to be considered effective.

Random designs are of interest because of the work by Dunietz, Mallows and Iannino [106] and Schroeder *et al.* [277] that indicates that the random design generation technique

[43] Engine control unit 47, which recently returned from the field as part of an engine update, was still functioning and had recorded over 50,000 hours of use. In 2006, it was estimated that the total time in use for the software exceeded two million hours.

may do as well as *t*-way adequate test generation techniques but with the advantage that they are both simpler to understand and simpler to implement.

Likewise, single factor experiments, i.e., base choice proposed by Ammann and Offut [9], are of interest because they have been found to be almost as effective as *t*-way adequate tests by at least two research groups, Grindal *et al.* [138] and Smith *et al.* [287], [286], who found them to be more effective than 2-way adequate test sets.

The final weakness identified in section 4.2 concerns the oracle problem. It is proposed that test set minimisation is a suitable method for reducing the number of vectors that need to be considered by a human to a manageable level and to remove or mitigate the requirement that a formal model or specification of the code that is being tested needs to exist.

Formally, the hypotheses to be tested are as follows:

H1 : that 2-way adequate test sets are at least as effective at killing mutants as hand-generated tests.

H2 : that *t*-way adequate test sets for a small factor greater than two, are at least as effective at killing mutants as hand-generated tests.

H3 : that it is possible to construct a minimised test set from a *t*-way adequate test set that is small enough to allow the correctness of results to be checked manually.

5. Experimental Design

5.1 Introduction

This chapter describes the major components of the work undertaken to address the questions proposed in the previous chapter. The work described in this chapter was performed in four phases. The initial work on sort routines was used to support development of the Csaw mutation tool set, sort routines being used as they all have the same functional purpose (i.e. specification) but employ different implementations which allowed the same test scaffolding to be used.

The work with industrial code and its associated hand-generated unit tests implements directly the program of work set out in chapter 4. That is to take a body of code, apply a number of different techniques for automatically generating test data, and compare the results both between those techniques and with the hand-generated test. The aim being to determine empirically which, if any, of those automatic generation techniques are competitive with hand-generated test sets. The work on test set optimisation follows directly from that work.

This chapter is divided into six main sections as follows:

- section 5.2 is a description of the experimental subjects that were used in this research;
- section 5.3 looks in detail at the two major tools that were developed to support the work undertaken and summarises the smaller tools that were developed to support the work undertaken;
- section 5.4 contains the description of experiments performed with sort code;
- section 5.5 contains the first set of experiments conducted with the industrial code and 2-way (pairwise) testing;
- section 5.7 examines a minimisation procedure for *t*-way adequate test sets;
- section 5.8 looks at the effect of including small sets of hand-generated tests on the behaviour of *t*-way adequate test sets.

5.2 Experimental Subjects

5.2.1 Sort Routines

Initial work on the mutation tool on "real" code was undertaken using a number of sort routines made available on the web by Lamont [207]. Sort routines were used because they provided a varying set of code with different levels of complexity, but with the same functional purpose. This meant that it was possible to use the same set of test vectors thus minimising the amount of "support" work that needed to be performed to hand-generate test vectors and to construct an oracle function. The routines used in this phase of the work were C implementations of the bubble sort, insertion sort, heapsort, shellsort and quicksort algorithms. The interface to each function follows the following pattern:

```
void <name>Sort (int numbers[],int array_size)
```

One immediate issue that had to be dealt with was how to treat the array of elements to be sorted. It was arbitrarily decided that a maximum array size of seven would be used and that the array size for a test would be set to a value between one and seven. Each element of the array would then be treated as a separate variable for automatically generating the *t*-way covering sets. This restriction is not considered significant, as other researchers have used limited sized arrays, for example Gotlieb [135] containing "at most 4 integers ranging from 0 to 24". The use of arrays *much* larger than this would introduce errors for indexing at byte, word and long word boundaries but would also require huge sets of test vectors to cover. Other options for dealing with arrays exist as discussed below (sections 5.2.4.1 and 7.3), but they did not seem necessary for the initial investigations undertaken with sort routines.

5.2.2 Industrial Code

The industrial code examples used in this research were drawn from the first release of software for running an industrial engine control unit. This software is considered to be safety related and has been developed using quality control process consistent with a process that meets the criteria for a System Integrity Level of two (SIL 2) [16] has unit testing consistent with a system defined as being SIL 3.

The software itself comprises of major logic sections that deal with the standard engine functionality such as:

- real time processing of engine rotational information;
- running a torque based proportional, integral and differential (PID) control loop synchronous with engine position;
- 5 and 25 millisecond periodic processing of analogue and digital inputs;
- 25 millisecond periodic calculation of spark angle;
- 5 millisecond periodic calculation of desired throttle position based on torque demand;
- 25 millisecond periodic combustion model;
- 5 and 25 millisecond periodic control of analogue and digital outputs.

Code examples have been drawn from the following sections; the PID feedback control loop (GOV), analogue and digital input processing (AIP, DIP), and the throttle control (THC).

5.2.3 Development Process

The manner in which code and the associated unit tests were developed is significant to the study. Many studies that have examined human developed tests have used students to develop the tests, including Laitenberger [205] with the notable exception of a study by Myres [237] who used professional programmers. A possible implication of this is that the hand-generated test sets used in these studies may be biased. For example, it is probable that the test sets are weak as students generally have little experience of performing controlled testing activities unless it forms part of their formal studies. Alternatively, student tests may be too strong if too much effort has been used to develop test sets. However, a review of the topic by Ellims *et al.* [112] suggests that this second scenario is not likely.

The industrial code in this study was developed using an instance of the generic quality management system (QMS) at Pi-Shurlok that is based on a standard "V" model (McDermid and Rook [223]). The QMS has been specifically set up so that it is simple to modify for specific projects as detailed in Ellims and Jackson [115], and thus gives as much flexibility as possible.

The QMS meets the requirements of ISO 9001 [15] and has achieved ISO 15504 [17] Capability Level 3 for all the processes in the defined scope of assessment that cover the requirements of the key European car manufacturers. In addition the specific quality plan and activities for this project (Wallace) have been externally audited to be consistent with the requirements for software to be developed for DO178B [14] Class B applications and with only small changes considered necessary to meet Class A requirements.

The left side of the "V" is comprised of the following major steps: requirements analysis and definition, functional requirements, architectural design, module design and coding. The right side of the "V" contains the corresponding verification and validation activities.

Each of the major components listed above consists of the number of sub-components such as review activities and checklists. Of particular interest here is the process for unit testing. Figure 13 shows the unit test activity and the simplified process flow within the activity. The term *simplified* is used as not all the features are shown, for example the activity Code Correction or Update involves a change request (CR) being raised, approved and acted on.

It should be noted that there are two phases to test execution - one on the host and one on the target system. Host execution is encouraged since running tests on the host environment is far more efficient than running tests on the target environment because of better tools (debuggers) and faster turnaround in the host environment. However, the tests have not passed until they have been executed successfully on the target hardware because issues with processor hardware and compilers are not that uncommon.

An essential point to note is that unit tests were reviewed and weaknesses detected were fed back into the test or occasionally test sets were reworked completely. For example, the main reviewer for the Wallace project (i.e. the author) has rejected tests that take a scattergun approach to the generation of input data. The review process serves two purposes. First, it directly improves the quality of the test sets and second, it teaches the engineers involved what comprises a good set of tests.

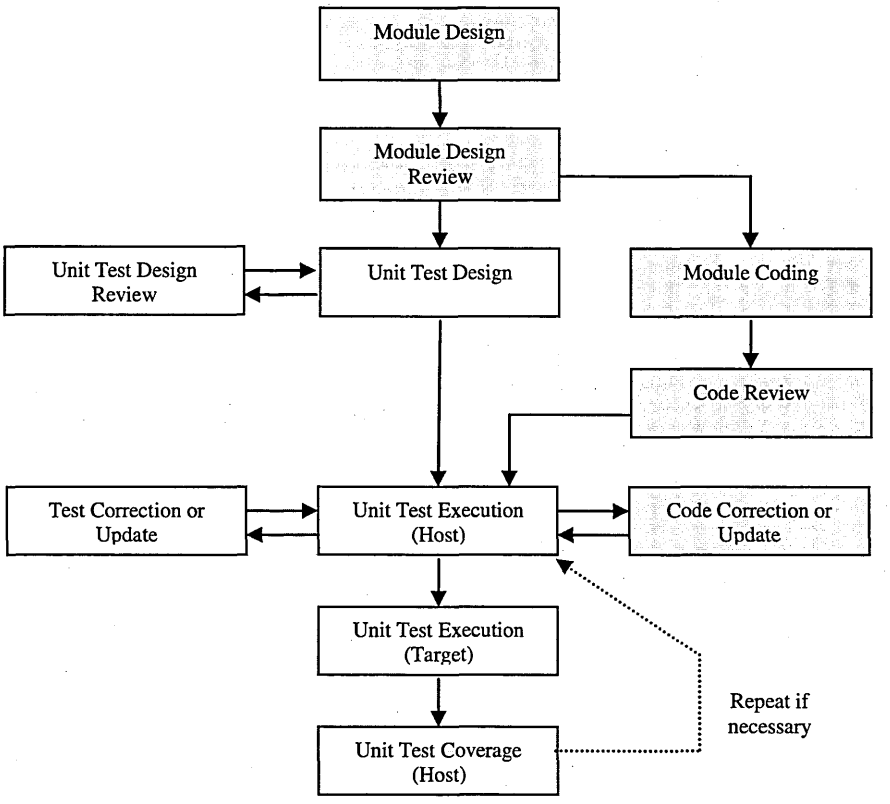

Fig. 13. Simplified process for performing unit tests. Shaded boxes show associated activities that must be completed before or in conjunction with unit testing.

Another feature of the process is that the coverage metrics for statement, branch and LCSAJ developed by Woodward *et al.* [336] were not collected as part of the test development process. Engineers are expected to design adequate tests without requiring direct feedback.

5.2.4 Test Subjects

5.2.4.1 Selection Criteria

Of the available functions, eleven were selected for inclusion in this study. Nearly half of the subject functions were selected (`dip_check_cal`, `dip_debounce`, `_sdc_pre_start`, `_aip_apply_filters`) because they contain known faults that were discovered during the original unit test process. Note that the function `dip_check_cal` was not included in more advanced work because it was so simple.

The remaining functions were selected because they all had the following properties to some degree:

- by the standards of the project, they have a reasonably complex control structure,
- they have a relatively simple interface to the outside world.

By simple, what is meant is that the functions did not rely on large arrays of data that had a significant amount of structure. The reason for excluding functions that relied on large or complex data structures, most of which are large arrays, is simple. It is not obvious how to deal with these within the context of a combinatorial test data generation system.

A number of obvious options were considered for dealing with data of this type:

- Treat each element of the array as a single variable and assign values to each array element independently of all other elements. This is the approach taken in the preliminary work on sorting algorithms.
- Deal with each array as a single object and select between a predetermined set of objects that are manually assigned values.
- Deal with each array as a set of smaller arrays and build the larger structure up by combining different sets of sub-arrays.

Each of these suggested approaches has a number of drawbacks. For large structures the first approach leads to truly huge numbers of possible combinations and correspondingly large sets of test vectors even for quite small arrays. The second option does not really solve the problem because the arrays are still generated by hand and are therefore merely using the combinatorial techniques to select between them. Although this approach may remove some of the bias in the data selection process noted by Teasley *et al.* [293], it is not a complete solution. The third option seems attractive. However, for the type of data usually included in the arrays it would inevitably introduce undesirable boundary effects where the sub-arrays were stitched together.

For some arrays some other options exist. For example, where the array is used in a table lookup operation (i.e. where we are interpolating between data points in an array) and the look-up operation is performed by calling a function, we can replace the function call with a test stub and directly supply the result of the table lookup to the code under test.

Also excluded from consideration were functions that were either written in assembly language or that contained significant amounts of assembly code embedded within them[44].

5.2.4.2 The Test Subjects

This section describes the set of subject functions used in this research. Each function is described to give an idea of its functional purpose and then information on the structure and complexity of each function is presented.

_dip_debounce is used to supply stable values of digital input values to the outside world. A value is deemed stable when it has had the same value for a calibrated time period (number of samples).

_aip_median_filter is used to select the median of the current input value and the previous two input values read. This is probably similar to the mid function used in research by Offutt *et al.* [252], Gotlieb [135] although the parameters here are arrays rather than individual items as would more normally be used.

_sdc_fuel_control is used to decide which state the engine is in, given a small number of inputs, namely engine RPM, start signal, shutdown status etc. and to prioritise among those conditions.

aip_spike_filter is used to remove large values seen in the input data. Its primary use is to cope with inlet manifold explosions where there is a rapid increase in the inlet pressure in a very short period of time.

_aip_apply_filters is used to control which filters (low pass, median, spike etc.) are applied to the raw analogue input values and in which order.

_thc_decide_state is used to decide what mode of operation of the throttle is being controlled. Possible states include, self calibration (location of end stops), recovery from non-volatile RAM failure, whether other control logic is overriding throttle movement and so on.

_thc_autocal is used to control the sequencing of an auto-calibration process for the throttle, waste-gate or bypass valves.

---

[44] Assembly code is quite common in embedded real time systems to access special purpose instructions that have no direct analogue in high level languages, e.g., for implementing semaphores, accessing special purpose registers etc.

_gov_rpm_err is used to calculate the difference between the desired and the actual RPM and apply both a lead/lag and low pass filter with appropriate clipping to avoid over and underflow in the calculations.

_sdc_pre_start is used to control the start or restart of an engine. This includes considerations such as the engine must not be rotating before the starting device is energised; that the fuel valves have been closed and the outlet manifold is flushed of any un-burnt fuel/air mixture.

_gov_gen_ffd_rpm is used to control the pre-loading of the integral term in response to outside stimulus such as a digital input that will indicate that in a known time a load will be placed on the engine. Further details can be found in Ellims and Zurlo [117].

A selection of properties for each of the functions is shown in Table 13 which contains the following information. The first column is the function name and the second column is the number of executable statements in the function, excluding blank lines, comments and braces. The third column gives the total number of mutants generated for each function. The fourth column gives the number of valid mutants[45] that would actually compile (ignoring warning for divide by zero etc). The fifth and sixth columns are the nesting factor and the condition factor as used in Michael *et al.* [232][46]. The seventh column is a simple count of the number of `if` statements in the code, each case of a switch statement being counted as a single `if` statement. The final column is the number of inputs to the function.

The function _dip_debounce stands out here, but this is because the underlying data structure is a set of arrays and the original test set contained data values for the first, middle and last elements of those arrays. As we are mirroring the original testing the original test structure was kept so as to be able to reuse the manually generated vectors.

---

[45] To deal with the mutants that could not be compiled, code that forced a divide by zero was inserted in place of the mutated function body which caused each of these mutants to die.

[46] Nesting factor is the maximum depth of nesting and the condition factor is the maximum number of comparisons performed in a single `if` statement.

Table 13. Summary of some properties of the code under study, with the C functions ranked by the total number of mutations that are generated.

Function Name	Lines	Total Mutants	Valid Mutants	Nesting Factor	Condition Factor	if statements	Inputs
_dip_debounce	12	127	81	2	2	2	17
_aip_median_filter	25	217	217	1	1	4	3
_sdc_fuel_control	17	267	213	2	2	5	9
aip_spike_filter	22	354	178	3	1	4	7
_thc_decide_state	16	387	386	7	2	7	9
_thc_autocal	33	782	669	5	2	8	6
_aip_apply_filters	30	605	311	2	2	4	8
_gov_rpm_err	22	1054	783	2	1	5	9
_sdc_pre_start	51	1472	1237	3	1	8	3
_gov_gen_ffd_rpm	62	1698	1227	4	2	11	16

5.3 Tools

Two large and several small tools were developed to support the work presented here. The first of the large tools was an implementation of the AETG algorithm developed by Cohen *et al.* [67]; the second was a tool to mutate C functions as investigations indicated that no such tool was available[47].

The small tools consisted of a data transformation program for output from the `jenny` tool and a program to generate different types of random data sets.

5.3.1 *t*-way Test Generation Tools

5.3.1.1 Description

Initial work on combinatorial adequate test sets reported in Ellims, Ince and Petre [114] employed a tool based on the AETG algorithm by Cohen *et al.* [67] to generate the *t*-way adequate test sets. However, this tool is inherently inefficient because it performs a linear search to match *t*-way tuples generated in candidate vectors with tuples remaining to be covered. Although adequate for pairwise (2-way) and 3-way test set generation, this tool proved to be infeasibly slow for values of *t* greater than three.

[47] Jeff Offutt who developed the Mothra and Godzilla tool sets [99] was contacted on this matter at the start of this work and indicated that no such tools were available.

Recent work by Lei *et al.* [211], [210] on the generation of *t*-way test sets compares the performance of their tool `FireEye` with other available tool sets. One of those tools is `jenny` which is freely available on the web. Table 14 compares test set sizes (top) and generation time (bottom) in seconds generated by `FireEye`, `jenny` and my implementation of the AETG algorithm on the TCAS code input data definition used in Hutchins *et al.* [174].

It can be clearly seen that `jenny` is far more time efficient than the tool specifically developed for this research. This is significant as for the most complex of the subject functions and for the larger 5-way adequate test sets, `jenny` takes several tens of seconds to generate test cases, my implementation would potentially have taken days. Using the `jenny` tool, is however, not without its drawbacks. For example, the method used to generate the vectors is not explicitly stated and the tool does not deal with single value (e.g. initialisation) data items which all had to be handled manually.

In addition, `jenny` generates completely generic data patterns (e.g. a1 to z255), which have to be converted to actual data values. A small tool was developed that partly automates this process. However, this process adds a non-trivial amount of time to data set preparation time.

Table 14. Comparative performance of three tools for generating 2 and 3-way adequate test sets. For each tool, the size of the test set is given and time taken to generate the test set is given in seconds.

	FireEye	jenny	our AETG
2-way	100	108	105
	0.8s	0.001s	422s
3-way	400	413	418
	0.36s	0.71s	18,986s

5.3.1.2 Validation

Validation of the tool relied on internal consistency checks and testing the tool on some simple examples from published research.

The primary internal consistency checks involves firstly, independently calculating the number of pairs, triples etc. that can be expected to be generated and checking that the code used to generate the actual sets produces the same number of set elements. In addition

at the end of the generation process, the set of test vectors that was generated is used to removed pairs, triples etc. from a copy of the original set generated and ensure that at the end of this process the copy of the of the original set is empty.

External consistency checks involved checking that the numbers of vectors generated with the tool was consistent with the numbers reported in example problems presented in the literature. These example problems were primarily taken from Cohen *et al.* [76] and then from Lei *et al.* [211] after the switch from the AETG based algorithm to the `jenny` tool was made.

The comparison with results from Cohen *et al.* is given in Table 15 and for Lei *et al.* above in Table 14 above. In Table 15, column one is the name applied in [76] to the problem. Column two is the number of vectors generated by the AETG tool as reported in Yu-Wen and Aldiwan [344] and Lei and Tai [212], column three is the number of vectors generated by Cohen *et al.*'s implementation of AETG and column four is the number of vectors generated by the tool developed for this thesis. Column five, is the definition of the problem as given in Cohen *et al.* [76].

Table 15. Performance of AETG based tool for generating 2-way (pairwise) test vectors against examples from literature.

Label	AETG	Cohen	This Work	Definition
CA-1	9	9	9	CA(N; 2,4,3)
MCA-1[48]	15	17	18	CA(N; 2, 3^{13})
MCA-1	19	20	20	MCA (N; 2, 5^1 3^8 2^2)
MCA-2	45	44	47	MCA (N; 2, 7^1 6 5^1 4^5 3^8 2^3)
MCA-3	30	28	28	MCA (N; 2, 5^1 4^4 3^{11} 2^5)
MCA-4	34	35	36	MCA (N; 2, 6^1 5^1 4^6 3^8 2^3)

5.3.2 Mutation Tool

5.3.2.1 Description

The Csaw tool set comprises a number of programs that mutate a function or small set of functions and then run the resulting mutants. The major component is a program `line.c` that processes a specially formatted C program one line at a time, breaking each line down

[48] There are two rows labeled MCA-1, the first is taken from Table 7, the second from Table 5.

into tokens. The program itself has several passes, first to build up a symbol table and then to apply mutations one line at a time. The output from `line.c` is a single file that contains all the mutants. Untch *et al.* [306] suggested that this approach may have a compilation bottleneck. However, this has not been found to be the case.

The second program `driver.c` is used to execute the mutations defined in `line.c` against a set of test vectors defined in a header file. To avoid issues with mutations causing infinite loops and with invalid operations such as divide by zero `driver.c` runs each mutant/vector pair as a child process using the `fork` system call. This allows the parent to monitor the execution without itself being involved; this allows it to kill a runaway mutant or to record an abnormal execution termination.

The effort involved in constructing each of these components is also instructive. It took a little over a month to put the major components of `line.c` code tool in place. By contrast the `driver.c` program however was much more problematic and took nearly twice as long to get functioning correctly[49].

The overall approach taken is not without its problems. One of the more significant problems is that, because of the simplistic approach (one line at a time) the C code is not parsed apart from building the symbol table[50], which requires that the user manipulate the source to put it in a form that can be correctly processed. This means that the approach relies on pattern matching to recognize certain elements of the program such as variable declarations which, in turn, means that recognizable declarations have to be explicitly defined. To reduce this burden, static tables have been used; these can be expanded as necessary. In practice, that has not so far been a significant issue.

The other major weakness of the one line at a time approach is that multi-line mutations are not possible. For example, the SMVB operator[51] defined by Agrawal *et al.* [5] cannot be implemented. In addition, multi-line comments cannot be dealt with and the tool requires that all comments be removed before processing. One novel aspect of the tool is that it will mutate the type specifier associated with a variable declaration. In some

---

[49] The relive sizes of the files used to build each of these two program is 122 Kbytes for line.c and 18 Kbytes for driver.c.

[50] To use the Csaw tool the user is required to manipulate the code into a suitable form, e.g., one statement per line. Details can be found in the Csaw users manual [111].

[51] Move a closing brace up or down one line.

quarters, this may seem unusual. However, in embedded systems where memory is usually at a premium it is common practice to use the smallest possible type to store values. Thus, both char and unsigned char are commonly used to store integer values. In practice, this has caught the author out at least once in production software.

There are, of course, also some advantages to the approach taken. For instance it should not be difficult to adapt the tool to other languages such as C++ and Java, which have similar structures. Further details of the Csaw tool are in the user's manual [111] and a comparison with mutation operators defined for FORTRAN by Offutt and Voas [253] and C by Agrawal *et al.* [5] is in Appendix A.

5.3.2.2 Validation

There are two main programs to be validated in the Csaw tool set, the `line.c` program that is used to create the mutants and various associated files and the `driver.c` program that is used to run the oracle and the mutant with each test vector in a set.

A different approach was taken with each of the two programs. Validation of the mutation program `line.c` was essentially constructive, initially using small test program fragments that were specially constructed to test each new feature as the tool was developed. Final testing was done using the set of sort functions sourced from the internet which overlapped with testing of the driver program.

Testing of the `driver.c` code involved running the completed program. The sort routines were used as they provided some variety in the code but all the code had essentially the same functional specification and could use the same test sets.

The actual verification involved several problems. In particular, mutant code cannot be run as a called function because if it causes a `signal` (Linux was used as the development platform), because of a divide by zero, for example, then the driver program is also terminated. This problem necessitated the use of a parent/child model where the parent is the driver and the child is the code under test.

Two primary strategies were used, therefore to perform validation:

1. The child process (target code) appended a summary of its execution to a text file, likewise the driver code appends information of how the child process terminated.

Crosschecking these against each other confirmed that the driver is correctly reporting results.

2. A small subset of the mutants was selected based on being able to determine *a priori* whether their execution will pass or fail.

The methods used in point 2 eventually lead to the development of a small stand alone test suite where code with hand mutated code is mixed with non-mutated code.

5.3.3 Other Tools

Several other small tools were developed to support the work but did not require the same amount of effort as the large tools described above. A brief description of these is given in Table 16.

Table 16. Small programs necessary to support the work reported here.

Program	Function
comb	A small program used to generate all *t*-way combinations of *n* variables. Used to independently check the adequacy of vectors sets generated by the AEGT implementation.
mcomp	A program that takes in two lists of live mutants and then compares them against each other. Output is two unique lists of live mutants.
random	Used to construct both fully random and test sets based on random designs. For random test sets the full range for each variable was used, for random design test sets, we randomly selected values from a set of *n* values.
trans	A small program that was used to transform the output data generated by the `jenny` program into the format used in the `test_vectors.h` file included in the mutant driver code.

5.4 Sort Experiments

5.4.1 Aims

The work doing during this phase had two principle aims. The first was to verify that the Csaw tool set was able to deal with real code both by systematically corrupting it and by being able to execute the resulting mutations. The second aim was to investigate one unusual feature of the Csaw tool. Csaw, unlike any other mutation system that the author

knows about, has the ability to mutate variable type declarations. That is it will replace a type specifier such as "`int`" with alternatives such as "`short int`", "`long int`", "`signed int`", "`signed char`", and so on.

5.4.2 Procedure

Work with sorting functions was carried out in two distinct stages. In the first stage the sort functions were used as input into the mutation tool to check that the tool could deal with actual code. The second stage of work was an ad hoc series of tests that were aimed at testing the mutation driver code.

The initial work with the mutation driver was performed with three test sets: one developed by hand, one generated randomly and a third 2-way adequate test set. Unlike latter tests, the test oracle for this work was a voting procedure that compare the output of the mutation under test, the un-mutated code and output from a third sort routine. Unexpectedly, the small hand-generated test set outperformed both a larger random test set and a test set generated using the AETG algorithm that contained four times as many vectors.

Examination of the test sets showed two significant points:

- The test sets all contained values that could be contained within a 16 bit signed or unsigned integer type.
- A large number of the mutants that survived were type mutations (see below).

As it was probable that the first of these points was the root cause of the low number of type mutations killed, investigating this was the first work that was carried out as a priority.

The method used was as follows:

- Several test sets were randomly created in which the array size was a value between two and seven and elements were randomly selected from values ranging from -2147483647 to 2147483647 with 10, 20, 40, 60 and 80 vectors.
- Another test set was created using the AETG algorithm by selecting six candidate values from points where the number of bits required to store a number changed (i.e. at byte and word boundaries).

- A further hand-generated test set of ten vectors was created using these same values. This set was constructed to ensure that it contained sorted, reverse sorted etc. patterns of data (as had the original hand-generated vectors).
- The voting oracle was replaced by a simple comparison between the mutated function and the un-mutated code. This was done, as usually no such voting procedure is available, as only a single implementation of the function would exist.

5.4.3 Experimental Results

The results from test runs that are shown in Table 17 indicate that there was no strong difference between the effectiveness of the test sets.

An examination of the code of live mutants showed that, for the simpler functions, the majority were type mutations and most of these were from a redefinition of the array size variable, which was constrained to hold a value between two and seven. It is therefore not surprising that these were not detected.

The remaining type mutations are equivalent, e.g., "signed int" for "int", non-equivalent mutations being effectively killed where data is takes on values that would allow them to be killed.

Table 17. Summary of algorithms, and performance for the seven test sets used. The size of the test set is shown and the number of live mutants and the execution time is given for each algorithm.

		AETG	Hand10	R10	R20	R40	R60	R80
		52	10	10	20	40	60	80
Bubble	**alive**	16	16	16	16	16	16	16
	clock time (s)	1607	311	321	598	1024	1807	2395
Insert	**alive**	20	20	20	20	20	20	20
	clock time (s)	1093	199	251	451	944	1491	2018
Heap	**alive**	44	46	45	45	45	45	45
	clock time (s)	3084	727	666	1184	2422	3663	4991
Shell	**alive**	67	67	67	67	67	67	67
	clock time (s)	5099	981	1012	1923	3901	5856	7723
Quick	**alive**	69	113	76	68	68	68	68
	clock time (s)	3767	739	517	964	1876	3047	3911

One of the most interesting features of Table 17 is the consistency of the results. With one exception, quicksort, there appeared to be very little to differentiate between any of the test data generation techniques.

For the more complex of the functions, a significant number of mutations survived. Here two cases stood out: first, the heap sort routine contained an integer used as a Boolean flag. Many of the operator mutants associated with this variable appeared to remain alive. Second, both shellsort and quicksort contained large numbers of mutations that were obviously not equivalent but that were nevertheless not detected.

5.4.4 Boolean Flags

In an examination of the mutations left alive from the heapsort runs it was noted that mutants associated with a Boolean flag used in the while loop comprised a large proportion.

The three uses of the variable were "`done = 0`" and "`!done`" and "`done = 1`". A significant number of the mutants were judged to be equivalent, e.g., with memory initialized to zero mutants such as "`done *= 0`" or "`done &= 0`" had no effect.

To see if breaking the codes reliance on the C language using any non-zero value as TRUE had any beneficial effect I assigned the variable `done` a specific value to indicate TRUE and replaced the clause "`!done`" with a test for equality on that value.

This slightly increased the number of mutants generated but had little effect on those that survived all tests. Similar results were obtained using Boolean values in code used in Ellims, Ince and Petre [113].

The unfortunate implication of this is that it would appear that some common code constructs exist that are going to be intrinsically difficult to deal with. This is of course not the first instance where Boolean flags have caused researchers problems, e.g., Michael *et al.* [232]. Dealing with the effects in an effective manner likewise appears non-trivial.

5.4.5 Time Equivalence

The observation that a number of mutations of both shellsort and quicksort functions were so obviously not equivalent in the code sense but produced identical results was initially perplexing. However, it was obvious that in some manner they must be compensating for the incorrect code by some other means. The most reasonable explanation for how this

compensation might have taken place is that the routines might have performed more work.

If this was the case, then it seemed plausible that it might be possible to observed this by measuring the execution time and detecting a delta.

The system that was used for this work was a 2GHz Pentium with 512 megabytes of RAM running SUSE Linux 9.2. Unfortunately, Linux has very poor timing facilities and the kernel build only supports a resolution of a hundredth of a second. Consequently, it was impossible to measure the time taken to run a single invocation of a function.

The first attempt to address this in an alternative way involved reworking the `driver.c` program so that:

- oracle data was collected for all vectors prior to running any mutant (to be folded into standard version);
- all vectors would be run over a mutant as a single operation.

There were a number of significant issues with this batching approach. First the results were very difficult to replicate with 100% consistency. Second, by having long runs it allowed the O/S more opportunities to interfere with the times generated.

Table 18. Summary of best results for batch timing.

	Delta	R10	R20	R40
Bubble	+/- 5%	-1	0	-
	+/1 10%	0	-1	+1
Shell	+/- 5%	-	0	0
	+/- 10%			

As the approach taken above was problematic, a search for a timing method with more resolution was investigated, namely a technique for directly accessing the clock cycle counter on a Pentium chip (Saikkonen [276]). A small series of experiments were conducted using the shell sort algorithm which again lead to no positive results possibly because of interference from the O/S itself as there were a number of issues associated with disabling interrupts. Other possible problems include cache usage and pipeline effects. It would be interesting to repeat this work in a more controlled environment such as an embedded system with no operating system.

Results for timing experiments with the system clock were rather mixed. Table 18 shows the best results obtained for the tests run with each of the test sets (random with 10, 20 and 40 vectors). A zero indicates no new mutants were killed, a negative number that, many extra mutants died, and a positive number that more survived, a result which is clearly erroneous.

It should be noted that both shellsort and quicksort have final passes that are potentially capable of compensating for other errors in the code; shellsort has a final pass that compares all adjacent pairs and swaps them and this may be sufficient if the error being compensated for is minor and the array is very close to be ordered. The quicksort algorithm performs a single pass of straight insertion sort (Knuth [195]).

5.5 Industrial Pair-wise (2-way) Experiments

This part of the study consisted of one major experiment and two subsidiary experiments.

The major experiment involved running the mutations for each function with the hand-generated test vectors, the vectors produced by my implementation of the AETG algorithm for generating pair-wise adequate test sets, and randomly generated test data of a similar size. Part of this experiment also looked at whether the resulting test vectors were adequate for the data.

The second experiment took a subset of the C functions and examined whether a more sophisticated approach to using the AETG algorithm could produce an improvement in performance.

The third experiment examined whether simply generating more vectors could improve the effectiveness of random testing. This experiment was done in order to establish a comparator for the pair-wise technique as suggested by Ince [177].

5.5.1 Procedure

The procedure employed in these experiments consisted of the following steps:

- From the project archive, I extracted the hand-generated test vectors, the comparisons originally used to determine correctness, and the information used to generate them from the detailed designs and data dictionary.

- The data on comparisons were used to construct an "oracle" that compared result data generated by the mutated function with data generated by the original version of the function. Any differences between the two sets results were flagged as a failure that kills the mutant.
- I then generated a 2-way adequate test set. For numeric variables, I selected the minimum, median and maximum values in the range defined in the data dictionary. For enumeration variables, I used all valid values and one out of range value to exercise the default statement in the code. For Boolean variables TRUE and FALSE were used.
- Finally, I generated a test set of the same size as both the hand and the *t*-way test sets for purely random tests. Numeric values were drawn from the whole range with equal probability and replacement. Enumerations and Boolean values were selected as above. The generator described in Park and Miller [261] was used for this.

5.5.2 Experimental Results

5.5.2.1 Minimum, Median and Maximum Values

The first experiment directly compared the hand-generated test vectors with test vectors generated by the straightforward application of the AETG algorithm using three values (minimum, middle and maximum) for scalar types and all values for enumerations. As a comparison, I included randomly generated sets of test vectors of comparable size.

Table 19 shows the results of this experiment, showing the function name, the number of valid mutants (i.e. those that compile with no errors), the number of vectors in the test set for the hand-generated data and the number of vectors generated by my implementation of the AETG algorithm. The next two columns show the number of mutants left alive after running each set of test vectors on all valid mutants for hand-generated and AETG generated tests. The final two columns detail the results for randomly generated test sets for the same size as the hand-generated vectors (Random versus Hand) and the same size as the AETG generated vectors (Random vs. AETG). Note that where the size of the hand-generated test sets and those generated by the AETG algorithm were similar only one size of random test (the larger) was used.

The main results can be summarised as follows:

- Random testing gives the best kill rate for a single function and AETG for one also, in the remaining functions the hand-generated test perform best (6 of 8).
- Random testing outperforms the AETG algorithm for two functions, and there was one draw. Of the remaining cases the AETG vectors beat the random test vectors by a substantial margin, and the remainder the performance of random testing is close to that of the AETG algorithm.
- For the most complex functions (i.e. those in the last two rows), the hand-generated tests outperforms both techniques by substantial margins[52].

Table 19. Results of the first experiment showing the number of mutants left alive after all test vectors have been applied. The test set that left the fewest mutants alive is in bold.

Function Name	Valid Mutants	Hand Vectors	AETG Vectors	Alive Hand	Alive AETG	Random vs. Hand	Random vs. AETG
_dip_debounce	81	18	17	12	***9***	12	---
_dip_check_cal	97	8	8	0	0	0	---
aip_spike_filter	178	40	15	***18***	42	82	90
_sdc_fuel_control	213	15	15	***21***	107	101	---
_aip_median_filter	217	27	9	***41***	47	53	57
_aip_apply_filters	311	68	21	64	58	***57***	57
_sdc_pre_start	1237	14	16	***675***	746	---	891
_gov_gen_ffd_rpm	1227	14	18	***152***	729	---	744

5.5.2.2 Error Detection

The second experiment in this set looks at the functions that have known five known coding errors committed by programmers during development of the code and extracted from the project change request database. The subject functions are listed in Table 20. As before, column one is the function name, column two shows whether the error was "found" with the hand-generated test set and column three shows whether the AETG test set found the error. Column four shows whether the randomly generated test vectors found the error. The column five shows whether the error was a mutant or not.

[52] A statistical analysis is performed in section 5.6.3.

When I applied the three sets of test vectors used against mutants in the first experiment I found that all of the test data generation techniques appeared to be effective at revealing the actual errors inserted in the code during development. Therefore, this second experiment did not allow us to draw any strong conclusions; but suggested that many real-world errors could, in practice, be quite "shallow" and possibly amenable to being found by any testing technique (automated or non-automated). Duran and Ntafos [107] reached similar conclusions. However, it should be noted that the complex functions were not included in the set of functions that were tested here.

Table 20. Effectiveness of test sets *versus* known actual errors in the code.

Function Name	Hand : error found	AETG : error found	Random : error found	Error is mutant
_dip_check_cal	Yes	Yes	Yes	Yes
_dip_debounce	Yes	Yes	Yes	No
_sdc_pre_start	Yes	Yes	Yes	No
_aip_apply_filters	Yes	Yes	Yes	Yes
_aip_apply_filters	Yes	Yes	Yes	No

5.5.2.3 Improved Data Point Selection

This third experiment looked at some simple ways of making data generated by the AETG algorithm more effective. For example, the following were considered:

- Invert: changing the assignment of values, i.e., converting elements that had the value `TRUE` to `FALSE` and vice versa.
- Interleave: rather than simply selecting the minimum, middle and largest value, I interleaved values that appeared together in conditional statements.
- Invert and interleave: inverts the interleaved values rather than just the minimum, middle and maximum.
- Biasing: altering the distribution of the data generated by adding duplicate elements. For example, specifying three values for a Boolean value as `FALSE`, `FALSE`, `TRUE` rather than just `FALSE` and `TRUE`.
- Interleave + biasing: a simple combination of the two previous techniques.

Interleaving is very similar to domain partitioning. However, because the analysis here has been less rigorous than might normally be required for domain partitioning, the term has not been applied. This less rigorous analysis is intentional in that I was deliberately attempting to avoid performing the type of analysis used in the construction of the original test sets, while attempting to obtain some of the advantages of such analysis. An example of the procedure as applied to the `_sdc_fuel_control` function is shown below.

The function compares four variables (A to D) against a global value for engine speed measured in revolutions per minute (RPM). Values were then allocated as shown in Figure 14.

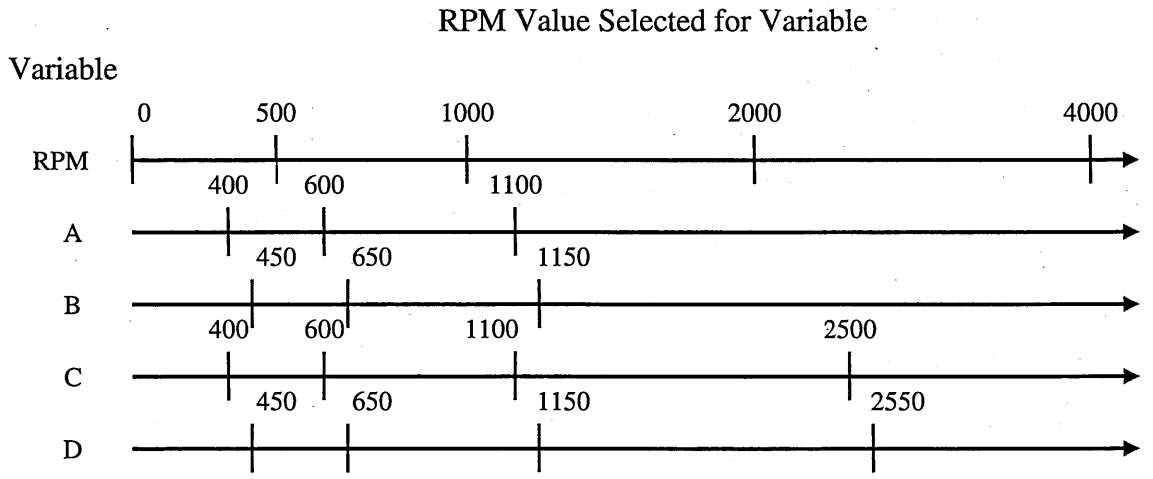

Fig. 14. Allocation of values for variable A to D and RPM for the`_sdc_fuel_control` function when Interleaving (not to scale).

One interesting aspect seen here is that the function `_sdc_pre_start` was not amenable to interleaving, no complex comparisons between variables being present. Exactly why this function seems to be so difficult to test is, at this point slightly bemusing.

There are some hints in the nature of the code however. For example, no computed values are assigned, all assignments are from constant values and many of these are to logical `TRUE` which in C is any no-zero value. Explicitly assigning `TRUE` to a fixed value and testing for that may form part of the solution. However, the driver program did test for equality of returned values so this may not explain why this function is so hard to test.

The results from the third experiment are shown in Table 21. The first column is the function name, the second column (Base) is the number of mutants left alive as in experiment 1. The column labelled "invert" is an inversion of data fields in the vectors used in experiment 1. The column labelled "Interleave" is a new set of test vectors generated using the technique outlined above. The column labelled "Interleave + Inverted" has the interleaved values used in column three inverted. The column labelled "biased" has one or more variables biased with the introduction of duplicate values. The seventh column combines the input values for the interleaved and biased test data generation. The last column replicates data for hand-generated tests for reference.

Table 21. Results for experiment three, to improve the mutant kill rate by modifying the input data points (e.g. interleaving) or the interpretation of those points (inverting). Two sets of data are shown for each function, the top row is mutants left alive and the bottom, the number of vectors

Function	Base	Invert	Interleave	Interleave + Inverted	Biased	Interleave + Biased	Hand
_sdc_fuel_control	107	97	96	25	31	26	21
	15	15	23	23	15	23	15
_sdc_pre_start	746	746	n/a	n/a	956	n/a	675
	16	16	---	---	20	---	14
_gov_gen_ffd_rpm	729	614	584	577	637	562	152
	18	18	26	26	19	27	14

Of the three complex functions retested in this third experiment, one, _sdc_fuel_control showed a significant improvement in the number of mutants killed in both the Interleave + Inverted and Biased test sets, coming close to the number of mutants left alive with hand-generated tests. Some improvement was also evident for the _gov_gen_ffd_rpm function with the Interleave and Interleave + Inverted test sets. However, the level of improvement is nowhere near as high in percentage terms and pairwise testing is still not competitive with the hand-generated tests. As can be seen from the table, there was no improvement in the number of mutants killed for the _sdc_pre_start function.

5.5.3 Summary

This set of experiments gives a clear indication that test sets constructed to be 2-way adequate (pairwise) are not competitive with a carefully constructed set of hand-generated

vectors in this experiment. The third experiment in series also suggests that while better data models, as suggested by Dalal *et al.* [93], [92] and Bell and Vouk [32] may improve the ability of automatically generated 2-way adequate test sets they are not a complete solution. To investigate this further an extended set of experiments using higher factors was performed as detailed in the next section.

5.6 Industrial *t*-way Experiments

5.6.1 Aims

The aims of this experiment were two fold. First, to evaluate the effectiveness of *t*-way adequate test sets relative to a set of high quality hand-generated tests. Second, to compare these tests with other techniques with similar problem analysis complexity.

5.6.2 Procedure

The procedure employed in this experiment consisted of the following steps for each functions:

- Generate *t*-way adequate test set sets for $t = 2$ to $t = 5$. Numeric variables, enumeration and Boolean values were treated as in section 5.5.1.
- Generate a test set of the same size as the *t*-way test sets using random selection from the same set of values with replacement. That is, select one valid value for each variable from the set of values used in generating the *t*-way tests.
- Generate a test set of the same size as the *t*-way test sets for purely random test. Numeric values were drawn from the whole range with equal probability and replacement. Enumerations and Boolean values were selected as above. The generator described in Wichmann and Hill [329] was used to ensure long sequences.
- For each function generate one or more sets of "base choice" test vectors as defined in Ammann and Offutt [9]. In base choice, a base vector is selected, perhaps based on expected or normal use and additional vectors are generated from this base by changing a single value of one variable in each new vector until all values have been used for all variables.

- For each function, execute each of the valid mutants on each test vector and for each test set recorded the number of mutants that were killed.

5.6.3 Experimental Results

Results are shown in Table 22. The first column gives the function name and the second states the information given in the next four rows as follows. For each function the first row (vectors) is the number of test vectors in the set determined by the size of *t*-way test vectors. The second row (*t*-way) is the number of mutants killed by *t*-way vectors for $t = 2$ to 5. The third row (rand sel) is the number of mutants left alive a using a test set created by random selection and the fourth row (random) is the number alive after applying the randomly generated test sets. The seventh column (Base) gives the number of vectors in the base choice test set and the number of mutants left alive below it (row labelled *t*-way). The final column (Hand) gives the number of vectors in the hand-generated test sets and the number of mutants left alive below it. For each function, the smallest test set that had the best performance is in bold.

Table 25 gives indicative information on the amount of time in seconds that it took to run each set of *t*-way adequate test sets data for each function.

Table 22. Number of mutants killed for each of the sets of test vectors applied.

Function Name	Process	2-way	3-way	4-way	5-way	Base	Hand
_dip_debounce	vectors	19	60	205	634	25	18
	t-way	**9**	9	9	9	28	12
	rand sel	14	9	9	9		
	random	11	10	10	9		
_aip_median_filter	vectors	12	28	54		7	27
	t-way	49	40	40		56	41
	rand sel	46	43	40			
	random	**40**	40	40			
_sdc_fuel_control	vectors	17	57	174	504	17	15
	t-way	101	49	25	22	36	21
	rand sel	126	31	24	22		
	random	84	58	25	**18**		
aip_spike_filter	vectors	16	49	146	400	14	40
	t-way	42	23	23	23	80	18
	rand sel	66	37	32	23		
	random	82	82	66	**16**		
_thc_decide_state	vectors	73	271	972	2883	28	17
	t-way	228	206	100	**57**	313	60
	rand sel	182	146	63	**57**		
	random	348	346	307	232		
_thc_autocal	vectors	20	70	181	377	14	6
	t-way	333	188	**187**	187	270	197
	rand sel	407	299	264	189		
	random	410	335	299	221		
_aip_apply_filters	vectors	34	142	562	1949	23	68
	t-way	47	46	46	46	64	64
	rand sel	**46**	46	46	46		
	random	**46**	46	46	46		
_gov_rpm_err	vectors	17	62	208	662	17	17
	t-way	**443**	443	443	443	444	446
	rand sel	**443**	443	443	443		
	random	465	462	462	460		
_sdc_pre_start	vectors	22	79	228	573	13	14
	t-way	736	**673**	673	673	965	675
	rand sel	700	**673**	673	673		
	random	742	742	742	742		
_gov_gen_ffd_rpm	vectors	21	81	299	1040	29	14
	t-way	701	190	158	**140**	785	152
	rand sel	663	270	148	**140**		
	random	502	265	152	152		

The primary concern of this section of the experimental work was to find which of the techniques was best at killing mutants in the selected functions. One way of considering

this is to look at which technique kills the most mutants for each function. The results are summarised as follows:

- *t*-way test vectors won or drew in six of the ten cases.
- Test vectors generated via random selection (random design) won or drew in half the cases.
- Random data generation won or drew in four of the ten cases but notably only has a single win in the second half of the table.

The selection of "a winner" from these results in this way is arbitrary in that it is the test set that killed the most mutants won regardless of the number of vectors required. Indeed for some code only small numbers of vectors were required. Another way to approach the concept of a winner is to examine the number of cases where a method failed to achieve a result comparable with the hand-generated tests. Here, there is one failure for *t*-way and random selection plus a near miss (_sdc_fuel_control by one) and four failures for random testing.

I also calculated the mean number of vectors required to kill each mutant. Here the number of vectors required to achieve the best result is used and I found that *t*-way requires 2.62 vectors per mutant, random selection required 2.71 vectors per mutant and random required 3.70.

Base choice was never the best performing technique and its performance was comparable with the hand-generated tests in only two cases. These results were surprising given that previous work by Smith *et al.* [286] and Grindal *et al.* [138] found that the technique to perform rather better.

The question is: are the main results for *t*-way and random testing statistically significant, given that the sample is small and the distribution of the data is not known? The Wilcoxon matched pairs signed ranks test[53] was applied to the results for *t*-way and random test sets versus those for the hand-generated vectors. Given the differences in the numbers of mutants generated for each function, the data was normalised to an interval scale by converting the raw score to a percentage value (mutants left alive as a percentage

[53] Calculations were made using the wilcox.exact function in the exactRankTests package of R versions 2.9.1 running under WindowsXP. Note that the wilcox.test functions should not be used if there are ties.

of total mutants). Table 23 shows the statistical values generated for the *t*-way test sets versus the hand generated test sets, the first row being the level of significance for a two-tailed test, and the second and third rows the sum of the positive and negative ranks.

The original hypotheses were:

H1: that 2-way adequate test sets are at least as effective at killing mutants as hand-generated tests.

H2: that *t*-way adequate test sets for a small factor greater than two, are at least as effective at killing mutants as hand-generated tests.

For the purpose of statistical analysis, the first hypothesis can be restated as the null hypothesis $H1_0$: The difference between the members of each pair (x, y) has median value zero and there is no difference between the treatments. The alternative $H1_A$: is that the median value is not zero, i.e., that there is a difference.

Table 23. Two-tailed P values and Wilcoxon values of positive and negative sums for percentage of mutants left alive for *t*-way adequate and hand-generated test sets.

	t =2	t =3	t = 4	t = 5
P	0.01953	0.3223	0.4492	0.5566
W^+	50	38	35.5	21
W^-	5	17	19.5	34

For H1, the null hypothesis $H1_0$ is rejected if the P value is less than $\alpha = 0.05$ (5% level of significance). For H1 this is clearly the case, and thus we can state that there is a significant difference between the median number of mutants left alive between the two samples ($T = 5$, $n = 10$, $P < 0.05$, Wilcoxon signed ranked test for matched pairs, two-tailed). In addition, if we examine the W^+ and W^- rank sum values we see that the *t*-way adequate test has a larger number of mutants left alive.

For H2 the null hypothesis $H2_0$ is the same: the difference between the members of each pair (x, y) has median value zero, and there is no difference between the treatments. However, in this case the P values for $t = 3$, 4 and $t = 5$ exceed 0.05 ($P > 0.05$ in all cases), thus we cannot reject the null hypothesis $H2_0$ and conclude that there is no statistically significant difference between the two sets of test vectors. Further, if we examined the W^+ and W^- rank sum values it can be observed that as we progress from $t = 3$ to $t = 5$, the rank sums progressively change to favour the *t*-way test sets and the rank sums for $t = 5$ suggest

that it may be more effective. However, a post-hoc single-tailed test does not reach a significance of $P < 0.2$. However, the hypothesis should not be rejected out of hand, because the number of samples is low ($n = 10$); instead, more investigation is warranted. The result also suggests that testing for $t = 6$ would be worth while (although computationally expensive) in order to throw further light on the topic.

Table 24. Two-tailed P values and Wilcoxon values of positive and negative sums for percentage mutants left alive for random test sets with the same number of vectors as the assocated *t*-way adiquate test sets and hand-generated test sets.

	t =2 equivalent	**t =3 equivalent**	**t = 4 equivalent**	**t = 5 equivalent**
P	0.01953	0.01953	0.05469	0.3008
W^+	50	50	39	32
W^-	5	5	6	13

If we apply the same tests to the results for random tests show in Table 24, we see a similar pattern, however here we reject the null hypothesis for random test sets the same size as the matched 2-way and 3-way adequate test sets. For the random test set of the same size as the 4-way adequate test set we should accept the null hypothesis as $P > 0.05$. However there is quite a possibility of committing a Type I error[54] given the closeness of computed P value to the selected limit ($P = 0.05469$). This is supported by observing that the Wilcoxon rank sums ($W^+ = 39$, $W^- = 6$) and $n = 9$ (one tie). For the random test set of the same size as the 5-way test sets we cannot reject the null hypothesis and state that at the 5% level of significance random test sets perform as well as the hand generated test sets ($T = 13$, $n = 9$, $P < 0.05$, Wilcoxon signed ranked test for matched pairs, two-tailed). It should be noted however that Wilcoxon rank sums ($W^+ = 32$, $W^- = 13$) still appear to favour the hand-generated test vectors so the evidence here is perhaps not as strong as in the case of the 5-way adequate test sets, and warrants further investigation.

Given these results, the obvious question to ask is, is the ability to kill mutants of the 5-way adequate test sets and the randomly generated test sets of the same size, the same? Applying the Wilcoxon matched pairs signed ranks test we have a null hypothesis H_0 that the performance of the two test sets is the same. Directly comparing the performance of the two test sets gives $P = 0.2188$ and means we cannot reject the null hypothesis ($T = 6$, $n = 7$,

[54] We reject the null hypothesis when it is actually true.

P > 0.05, Wilcoxon signed ranked test for matched pairs, two tailed). However, examination of the Wilcoxon rank sums (W+ = 6, W- = 22) suggest that the 5-way adequate test sets are perhaps more effective, more evidence is however required.

Table 25. Execution times for the *t*-way adequate test sets.

Function Name	Valid Mutants	2-way	3-way	4-way	5-way	Max (hours)
_dip_debounce	81	76	210	743	1649	0.46
_aip_median_filter	217	64	127	248		0.07
_sdc_fuel_control	213	132	362	808	3667	1.02
aip_spike_filter	178	109	433	858	1665	0.46
_thc_decide_state	311	707	2723	8156	43451	12.07
_thc_autocal	386	139	582	2313	4253	1.18
_aip_apply_filters	669	198	420	675	2788	0.77
_gov_rpm_err	783	212	851	3239	8563	2.34
_sdc_pre_start	1237	906	1506	5083	16,231	4.51
_gov_gen_ffd_rpm	1227	972	2612	17,758	33,653	9.35

Another factor that needs to be considered is whether the two sets of vectors kill the same mutants, or whether the set of mutants killed by each are disjoint with only a small intersection. Data presented in Table 26 shows the number of mutants left alive for each the hand-generated tests and the difference between the two (delta). In addition, it also gives the number of unique mutants that were left alive by one set of test vectors but not the other. The data indicates that there was actually a significant overlap. This result is significant because it implies that it is reasonable to assume that each of the two sets of test vectors have a similar ability to detect actual faults. Although the relationship between the ability to kill mutants and detect faults has not been investigated in this work studies such as Andrews *et al.* [11] have indicated that if a test set is good at one task, then it is also good at the other.

Table 26. The number of mutants not killed by the largest *t*-way adequate test seta and hand-generated tests. The final two columns are the number of killed mutants unique to each set.

Function Name	*t*-way Alive	Hand Alive	Delta	*t*-way Unique	Hand Unique
_dip_debounce	9	12	3	1	4
_aip_median_filter	40	41	1	0	1
_sdc_fuel_control	22	21	-1	8	7
aip_spike_filter	23	18	-5	9	4
_thc_decide_state	57	60	3	4	7
_thc_autocal	187	197	10	0	10
_aip_apply_filters	46	64	18	0	18
_gov_rpm_err	443	446	3	3	6
_sdc_pre_start	673	675	2	0	2
_gov_gen_ffd_rpm	140	152	12	1	13

5.6.4 Investigations

There are several interesting features present in Table 22 that deserve comment:

- Why is it so difficult to obtain a good kill rate for the _sdc_pre_start function?
- Is the fault detecting ability of random testing really static for _sdc_pre_start?
- Can the results for _gov_gen_ffd_rpm be improved if we use more random tests?

An examination of live mutant's _sdc_pre_start code reveals that the majority of live mutants were connected with manipulating variables that have Boolean values. As has been noted in other work Ellims, Ince and Petre [114] and in a large amount of research on searched-based test data generation, e.g., Michael *et al.* [232] and Bottaci [39], Boolean data appears to be intrinsically difficult to deal with.

The _sdc_pre_start code was executed with a number of different randomly generated test sets using different seeds for 288, 573, 1200 and 2400 values. Although some of the vector sets showed some improvement, the best result returned was only 717 live mutants and all data sets showed the same flat pattern as shown in Table 22.

Code for _gov_gen_ffd_rpm was run with a test set of 2000 and 5000 vectors taking 12 and 32.4 hours to execute. The test set of size 2000 showed no improvement while the test set of 5000 vectors killed only two additional mutants.

5.7 *t*-way Optimization

5.7.1 Aims

There are two obvious issues with the data presented above. First, that the execution times with *t*-way test generation were long for some functions when compared with the time taken to generate the tests by hand. Timesheet data gave an average of 5.6 hours for AIP functions, 5.7 hours for DIP and 1.9 hours for SDC functions. Second, the number of vectors that would have to be examined to determine whether a test passed or failed is infeasibility large. In practice, a large part of the problem with generating tests by hand is determining whether the output is correct. Given the volume of tests generated automatically, determining whether the code passes or fails places an unacceptable burden on the tester and significantly reduces the utility of any automatic generation technique.

Therefore, this experiment has two aims. First, to investigate the potential of reducing the amount of time required to run all the mutants. Second, to determine if a minimal test set can be extracted from the process to reduce the oracle problem to a manageable level.

5.7.2 Procedure

For this experiment, the test driver was modified to record which vectors killed which mutants for each set of test vectors. After all vectors from a *t*-way test set had been run over all remaining mutants the minimisation routine determined which vector killed the most mutants and this vector is selected to be retained. The mutants that this vector killed were removed from further consideration. This process was repeated for each vector in the test set until there were no vectors remaining that killed more than one mutant.

The run with the next set of vectors excluded from consideration those mutants that were previously killed by all preceding test sets but otherwise the minimisation process was identical. This continued until the final set of vectors was run, when the restriction on not selecting vectors that only kill a single vector was removed.

Other procedures to reduce the number of vectors that need to be considered have been investigated previously. A simple suggestion by Offutt [248] was to ignore the vectors that do not kill any mutants. However, these experiments suggested that savings might not be great, as a large number of vectors kill at least one mutant, which is why I delay selecting

any vectors that kill only a single mutant until the final pass. Offutt *et al.* [255] suggested a mechanism for selecting minimal sets of vectors that again removes mutants as they are killed but runs the set of vectors in different orders.

5.7.3 Experimental Results

The results from experiment 2 are shown in Table 27. This table reports the time to run the largest *t*-way test set (max), the time using the minimisation procedure outlined above (min) and the percentage time saving for the minimisation (gain) for each function tested. Information on vectors given in the table is the number of hand-generated vectors (hand), the size of the largest single *t*-way adequate test set (max) and the size of the optimised test set (min). For reference the *t* value of the test set that first resulted in the maximum number of mutants killed is shown in the second column, which is labelled *t*.

Table 27 shows that, in terms of time saved, the minimisation procedure delivered significant saving for most of the functions, with an average saving of close to 53%. However, it is also clear that for functions that showed no increase in mutants killed at higher values of *t* (e.g. _dip_debounce) the process can be counter-productive. However, that it is not always the case, for example _aip_apply_filters. The benefits where high *t* values did show improvement are more supportive of the idea that the minimisation scheme trialled here is worth while.

Table 27. Summary data for *t*-way minimisation runs.

Function Name	*t*	Time (seconds)			Vectors		
		max	min	gain	hand	max	min
_dip_debounce	2	1649	2029	123 %	18	634	6
_aip_median_filter	3	248	67	27 %	27	54	9
_sdc_fuel_control	5	3667	1144	31 %	15	504	12
aip_spike_filter	2	1665	628	37 %	40	400	9
_thc_decide_state	5	43451	6942	16 %	17	2883	13
_thc_autocal	4	4253	1276	30 %	6	377	13
_aip_apply_filters	2	2788	2029	73 %	68	1949	7
_gov_rpm_err	2	8563	6118	71 %	17	662	4
_sdc_pre_start	2	16231	18212	112 %	14	573	12
_gov_gen_ffd_rpm	5	33653	5767	17 %	14	1040	22

Results for the size of the test sets from the minimisation routine were less ambiguous than those for time minimisation. In eight of the ten cases, the test sets were smaller than the hand-generated test sets. In the remaining two cases, they are not substantially larger in terms of total tests required. Spearmans'rank order correlation was applied to the size of the of the hand-generated and *t*-way adequate test sets the calculated value of r_s is -0.665 with eight degrees of freedom which suggests a modest[55] negative correlation. Testing the significance of this, the null hypothesis H_0 is that there is no correlation. At the 5% confidence interval P is 0.648 which is less than the calculated value so we cannot reject H_0 at the 5% level of confidence.

There is however one down side, as reported in Smith *et al.* [286] to this minimisation process. Namely, vectors that were selected by the minimisation procedure were not very user friendly. That is, it would take a significant effort to understand what is being tested. Here, none of the test cases contained tests that would be obvious to an engineer producing the test cases by hand (the author was the engineer in charge of the Wallace[56] project). Indeed many of the test cases, especially those for the function _aip_apply_filters contained data that, in practice, would not be used and would be disallowed by the tool that vets the engine control unit calibration data. The importance of this is not completely clear; the vectors in the *t*-way adequate test set are clearly capable of killing almost the same set of mutants as the hand-generated test sets (see Table 26) which suggests that they are as adequate for killing mutants. The questions then becomes are mutant adequate test sets good at detecting real faults? That question is however outside the scope of this work.

One possible method of addressing this issue is to use as a first pass to the minimisation process a small set of hand-generated vectors. The feasibility of doing this is considered in the next section.

---

[55] Fowler and Cohen [120] pg. 132.

[56] There was also a project Gromet.

5.8 Minimisation with Hand Vectors

5.8.1 Aims

Given the results from the minimisation experiments in the previous section, an obvious question to ask is whether there is any benefit to be gained by including a small number of user generated tests in the process.

Because there is a set of user derived tests available, the aim of this experiment was to simulate the situation where the user creates a small number of tests to target the main paths though the code. This approach potentially has a number of advantages compared with taking the purists view that the creation of tests sets is an either/or situation with regards to the use of automatically generated data.

It should be noted that this procedure is most probably not equivalent to the "seeding" suggested by Cohen *et al.* [67] and Czerwonka [87]. The reason for this is that the process does not take into consideration the tuples that are potentially covered by the hand-generated test. Indeed this is not actually possible given the disjoint nature of the set of values from which that the two sets of vectors were drawn.

5.8.2 Procedure

The driver program was further modified so that it would take as parameters the number of values to be selected at random without replacement from the test set and the seed to use for the random number generator. An example of this procedure is shown in Figure 15 where the compiled code containing the hand-generated test is passed the parameter `r8` which specifies that 8 vectors are to be selected and the parameter `i1` specifies that seed 1 is to be used[57].

```
dec_hand.out r8 i1 g p >res_r8_1_dec_hand.txt
dec_2way.out n g p >res_r8_1_dec_2way.txt
dec_3way.out n g p >res_r8_1_dec_3way.txt
dec_4way.out n g p >res_r8_1_dec_4way.txt
dec_5way.out n >res_r8_1_dec_5way.txt
```

Fig. 15. Example application of the optimization process.

[57] Seeds are all large seven-digit prime numbers taken from a table of first 10,000 prime values.

The other parameters shown in Figure 15 are defined in Table 28.

Table 28. Parameters that can be passed to the mutation driver program.

Parameter	Usage
d	write child debug file, used to enable the output of debugging information from the child process spawned for each mutant.
g	generate a no-run list, creates a list of mutants that can be excluded for the next iteration as vectors have been found that kill them.
i	initialise/select random seed to be used
m	use a no-run list, read in the set of mutants that are not to be run during this execution.
p	perform partial minimisation, select only vectors that kill more than one vector
r	enable random selection of test vectors with subset sized N
s	skip mutation execution, also used for debugging. Allows the driver to be run without spawning any child processes.

Each of the larger functions in the industrial code based was then run though the minimisation process five times with different seeds using values for the `r` parameter of 2, 4, 6 and 8 (i.e. for a total of 20 runs). Data was collected on the number of mutants not killed, the total execution time, and size of the resulting optimised vector set. The mean values for each set of runs with equal `r` values were then calculated.

5.8.3 Experimental Results

Experimental results are summarised in Table 29. As previously the first column is the name of the function being tested. The second column lists the measures used to compare the results from this experiment with the results from the optimisation runs presented in Table 27 (vectors and time in seconds) and the live mutants from Table 22 (alive). Data from these tables are replicated in the final column, which is labelled "best to date". The remaining columns show the number of vectors drawn from the hand-generated test sets. The data given being the mean of the five runs.

Table 29 shows that adding small numbers of hand-generated vectors has a small effect on the size of the resulting test data sets (number of vectors) and a slightly better effect on the execution times.

The effects however are not always consistent, for example for `_gov_rpm_err` the number of vectors in the resulting test set was larger than for the best case without the hand-generated vectors. However, the improvement in time was a little more consistent; with four of the six functions showing a clear improvement which is probably because the hand-generated phase of the optimisation process is very short and results in all subsequent phases being more efficient. There was also a small reduction in the number of mutants left alive in some cases, but this effect was relatively small.

The statistic significance of these results was investigated using Wilcoxon matched pairs signed ranks test for N = 8 and the data for the best performance from the optimisation process (best to date). The null hypothesis is the same as previously, i.e., H_0: the difference between the members of each pair (x, y) has median value zero and there is no difference between the treatments. For execution times at the 5% confidence level we cannot reject the null hypothesis (T = 1, n = 6, P = 0.0625, Wilcoxon signed ranked test for matched pairs, two-tailed) and conclude that there is no difference between the two samples. For the size of the final set of test vectors at the 5% level of confidence (T = 4, n = 5, P = 0.5, Wilcoxon signed ranked test for matched pairs, two-tailed) we cannot reject the null hypothesis and conclude there is no difference.

These results suggest that there was no negative effect of adding a small number of hand-generated tests to the *t*-way adequate test sets and that this procedure may be advantageous in that the execution time has the potential be considerably reduced. We should also consider the amount of effort required to check the reduced test sets manually for correctness. If we assume that manually generated tests will be simpler to verify than automatically generated tests then the total effort of checking the vectors has potentially been reduced.

Table 29. Summary of results for adding *N* (2, 4, 6, 8) hand-generated tests randomly drawn *without* replacement from the set of hand-generated tests associated with each of the subject functions.

Function Name	Measure	Hand generated vectors added				Best to date
		N = 2	N = 4	N= 6	N = 8	
_dip_debounce		*not run*	*not run*	*not run*	*not run*	
_aip_median_filter		*not run*	*not run*	*not run*	*not run*	
_sdc_fuel_control		*not run*	*not run*	*not run*	*not run*	
aip_spike_filter		*not run*	*not run*	*not run*	*not run*	
_thc_decide_state	alive	56	56	55	55	57
	time (s)	4803	4660	3489	2826	6942
	vectors	9	9	10	11	13
_thc_autocal	alive	187	187	187	187	187
	time (s)	2356	2076	1578	1420	1276
	vectors	15	16	15	15	13
_aip_apply_filters	alive	46	46	46	46	46
	time (s)	1613	2130	2157	1706	2029
	vectors	6	7	7	7	7
_gov_rpm_err	alive	443	442	438	440	443
	time (s)	3685	5504	3280	4217	6188
	vectors	5	5	6	6	4
_sdc_pre_start	alive	673	673	673	673	673
	time (s)	7211	7492	8201	7484	18212
	vectors	12	12	9	9	12
_gov_gen_ffd_rpm	alive	143	140	140	140	140
	time (s)	3439	2258	3796	2534	5767
	vectors	17	17	17	15	22

6. Conclusions

6.1 Introduction

This chapter is divided into sections as follows:

- Section 6.2 gives a brief summary of the main experimental results.
- Section 6.3 reviews the original hypothesis that was investigated and the conclusions that can be reached given the experimental results.
- Section 6.4 discusses results from random testing.
- Section 6.5 briefly discusses the effect of combining human generated and automatically generated test vectors.
- Section 6.6 looks at the threats to the validity of the experiments that were conducted.
- Section 6.7 looks at specific weaknesses in the work that was undertaken and suggest ways of avoiding those in future.

6.2 Summary

The original research question being investigated was to determine whether automatic test data generation as currently documented in the literature is competitive in terms of error detection capabilities with test sets developed by traditional human methods when applied to unit testing. Specifically, the tests were designed to test the following hypotheses:

- that 2-way adequate test sets are at least as effective at killing mutants as hand-generated tests;
- that *t*-way adequate test sets for small factors (t = 3, 4, 5) are at least as effective at killing mutants as hand-generated tests;
- that it is possible to construct a minimised test set from a *t*-way adequate test set that is small enough to allow the correctness of results to be checked manually.

The main sets of experiments that were conducted are as follows:

- experiments that tested a set of sort functions using 2-way, random and hand-generated tests.
- experiments that tested small set of functions drawn from the code for an engine control system using 2-way, random and hand-generated tests (Table 19 and Table 22).

- extending the second set of experiments to higher factors to determine if that resulted in an improvement in coverage (Table 22);
- experiments that used data on which test vectors kill which mutants to minimise the size of the test sets to allowing them to be reviewed manually (Table 25);
- experiments that added small numbers of hand-generated tests as the first stage in the optimisation process (Table 27).

Allied with these experiments were a number of additional investigations such as the timing experiments described in section 5.4.1.5, the examination of the techniques ability to detect real faults that is described in section 5.5.2.2 and work with more complete data models in section 5.5.2.3.

Overall, the main planned set of experiments were aimed at the obvious interpretation of the primary research question. That is, whether automatically generated test sets can discover as many faults as hand-generated test sets. The ad-hoc investigations had the same purpose.

6.3 Conclusions

The results of these experiments have been surprising. At the start of this study, it was suspected that 2-way techniques would offer an effective way of testing critical software given the volume of studies that supported this view. However, the results show that although 2-way adequate test sets may be effective in some situations, for example the field studies reported in chapter 3, they have generally fallen short of expectations. Nevertheless, the research conducted here supports the view that test sets that are adequate to higher factors show significant promise. Likewise, the process of optimising versus mutation adequacy appears to be very effective at reducing the number of vectors that need to be examined by hand. The next three sections address these points more formally.

6.3.1 Hypothesis One

Hypothesis one was "that 2-way adequate test sets are at least as effective at killing mutants as hand-generated tests".

The results given in **Table 19** and **Table 22** indicate that 2-way (pairwise) combinatorial techniques that use simple selection criteria for selection of data points are not adequate

with respect to hand-generated tests for the more complex functions where complexity is indicated by the number of mutants generated, the nesting level of the code, or number of condition statements. This is supported by the statistical analysis given in section 5.6.3 on the data in Table 22.

This is counter to the conventional view put forward in a number of field studies such as Dalal *et al.* [93], [92], Burroughs *et al.* [51], Huller [173], in various results reported by Cohen *et al.* [68], [70], [67] and work by Burr and Young [50]. However, the results presented in this thesis give some support to the observations of Smith *et al.* [287], [286] who found that 2-way test vectors were less effective than base choice [9] for certain classes of faults.

Of the empirical studies, only the work by Grindal *et al.* [138] [139] strongly supported the effectiveness of 2-way adequate test tests. In common with Smith *et al.* [287], [286], Grindal *et al.* found that base choice techniques were effective, another result that is not supported by the work presented here, which found base choice to be possibly the worst of the techniques examined.

The reason for the disparity between the results reported in this thesis on 2-way testing and earlier work is not clear although it is possible that the earlier work was biased because it included comparisons with testing that was originally poorly performed. For example, comparisons with the field studies by Dalal *et al.* [93], [92]. Latter field studies by Smith *et al.* [287], [286] could also have had a bias in that real faults were being examined and this may have skewed the distribution of faults that could be detected.

The empirical study by Grindal *et al.* [138] [139], however, is difficult to explain on the same grounds. One possibility is that the code was not complex enough for the purpose having been created originally to test manual fault detection techniques by Kamsties and Lott [186]. A similar effect was observed with studies performed here that used different sort functions. In this case it was found that the it was virtually impossible to differentiate between the different test data generation techniques. Quite what this says about a large number of empirical test studies that have made use of sort functions is uncertain. However, it suggests that simple functions of this type are *not* good subjects for testing research.

6.3.2 Hypothesis Two

Hypothesis two was "that *t*-way adequate test sets for a small factor greater than two, are at least as effective at killing mutants as hand-generated tests".

The results presented in Table 22 show that test sets that involve higher values of *t*-way adequate tests are as effective as hand-generated tests at killing mutants. Again, this is supported by the statistical analysis. However, this statement holds only in terms of being able to distinguish mutated from original code. This work has not assessed the ability of distinguish "real" faults from code mutants to any great extent. However, as noted previously, results from both Offutt [247] and Andrews, Briand and Labiche [11] strongly suggest that a test set adequate for killing mutants for one will be effective at finding real faults.

The results here strongly support the body of work by Dunietz *et al.* [106]; Kobayashi *et al.* [197]; Kuhn, Wallace, Reilly and Gallo [313], [202], [203] and [201] and contradicts the results given in study by Schroeder, Bolaki and Gopu [277].

Why this contradiction should have arisen is not known. However, one possible reason is that Schroeder *et al.* [277] performed the testing from the external interface to the program, which might have made it more difficult to target effective combinations at the points where they would be most effective.

Other possible explanations include:

- that the data structures or the structure of the data in Schroeder *et al.* is more complex than in the code examined in this study (see section 6.8 for a discussion of arrays);
- at $t = 4$ the factor applied was not sufficient to trigger the faults.

6.3.3 Hypothesis Three

Hypothesis three was "that it is possible to construct a minimised test set from a *t*-way adequate test set that is small enough to allow the correctness of results to be checked manually".

Results from the minimisation experiments given in Table 27 are highly encouraging to say the least. For the code examined, all but two of the resulting sets of test vectors are actually smaller than the vector sets for that hand-generated code. For the remaining two sets the vectors are not overly large being just over half the size again in one test set and

just of twice the size in the other set. However, vector sets of 22 and 13 vectors are not excessive even compared with some of the existing hand-generated vector sets in the project. In addition, each of these larger vector sets outperforms the hand-generated tests slightly.

Unfortunately, there is little work that can be directly compared with the study here. Only one paper Offutt, Pan and Voas [255] directly addresses the issue of minimising mutation adequate test sets. Moreover, the reductions in test set size observed by these researchers was only on the order of 33% though it is assumed from a much smaller base figure. A reduction of this order would be almost useless when we consider factors larger than three as it would still leave us with several hundred or thousand test cases that need to be considered. From this perspective, the effectiveness of the rather simplistic minimisation procedure developed here is at first sight astonishing.

In addition, the batched procedure used here has at its disposal a much larger amount of information than the procedure suggested by Offutt, Pan and Voas [255], which removes mutants as they are killed and is therefore much less likely to be able to identify and select the best test cases than the batched procedure. That being said, the minimisation procedure suggested here is only locally optimal.

6.4 Threats to Validity

One threat to the external validity of the results reported here is that code being tested may not be representative of other code although other researchers Gotlieb [135], Offut *et al.* [252] have used a variant of aip_median_filter and Dillon and Meudec [104] used the function itself. This is a general problem in testing research and code from different application domains is likely to have different properties. The code used here is thought to be representative of fixed point integer code widely used in the real-time embedded applications domain.

A novel threat is that because the complete software development process was strongly controlled as detailed in Ellims, Bridges and Ince [112], the code used in this thesis may actually be easier to detect faults in than more typical code. Thus the results presented here are possibly optimistic. The only way to test this possibility would be to repeat the

experiments with other code sets. However, often these rarely have the necessary hand-generated test vectors available for comparison. Another threat is that code mutation may not be representative of real faults. Results in both Offutt [247] and Andrews, Briand and Labiche [11] strongly suggest that test sets that are adequate for mutation will also be effective for real faults.

The final threat to validity that needs to be considered is that of equivalent mutants. In the work presented here, these have mainly been ignored. The effect on the results, however, is expected to be minimal because equivalent mutants affect all of the techniques being used to the same degree. Thus, although the results are internally consistent and direct comparison between test sets is possible, making comparisons with other work is more difficult.

The major threat to internal validity comes from the way in which the data points that were used in the *t*-way and random selection data sets were limited to minimum, median and maximum values. This is a simplistic approach. However, it should tend to bias the results against success, resulting in a false negative. Moreover, the data selection process does follow examples in books such as Copeland [81], which will possibly provide the primary source of information on combinatorial techniques for practitioners.

The tool used to insert faults into the code may also present a risk to internal validity because although it avoids the bias associated with hand-seeded faults, it is a relatively simple tool and is not capable of introducing mutants over multiple lines. Analysis of the results however suggests that the majority of effective operators have been implemented; this analysis is provided in Appendix A.

6.5 Observations

6.5.1 Random Testing

Random testing as a data generation technique can be surprisingly effective. However, it does not appear to be reliable in the sense that, although it may often provide good results, it cannot be counted on to provide good results always. Of the ten functions tested to high factors, random testing performed best in three of the first five least complex functions, but

only performed best in two of the second five functions and in both cases was tied with another technique that performed equally as well.

In addition, for two of the most complex functions _sdc_pre_start and _gov_gen_ffd_rpm - the use of very large numbers of random tests did not show any real improvement in the performance. This result matches observations on random testing made by a number of other authors such as Michael *et al.* [232], Frankl *et al.* [121] and Reid [269], [270] that indicate that when random testing works, it works well, but when it does not work, the results can be spectacularly bad.

6.5.2 Combined Human/Machine Vectors

Some initial work has been done (section 5.8) using small numbers of hand-generated vectors as the first step in the optimisation process. Initial results suggest that although the number of mutants killed is only minimally affected, additional savings could be made in execution time. As processor time is cheap when compared with human effort, this additional step may not ultimately prove to be effective.

However, the above conclusion does not take into account the fact that a small number of human derived tests may be simpler to validate than the automated tests. For example, a carefully selected set of tests that have easy to determine correct results may be of more use and perhaps more effective than automatically generated tests that required significant human input to determine their correctness. The limited work done here does not discount this possibility, which is worth further investigation.

6.6 Discussion of Tools

6.6.1 Hand Generated Tests

The original hand-generated tests were embedded in spreadsheets. The main reason for this was that it was found that a major issue with developing unit tests by hand was that it took so much effort to compute the expected results.

To speed up the process spreadsheets were used to allow the user to calculate expected results that could then be extracted as a comma separated file and pasted into a C program to run the tests on the function. As much of the code for this is very similar, this approach

eventually resulted in the development of the Test Harness Generator (THG) as reported by Ellims and Parkins [116]. THG automates much of the process of taking a set of input data and expected results and producing a complete test harness for a function in C that can be compiled. Use of a spreadsheet even allows some opportunities for automatically generating input data.

For the research conducted here, this has several downsides. First, the sheets used for THG are quite complex containing information on the function called, the functions to be called (if any), values for `#define` names used in the sheet, and so on. Second, the sheets contain large numbers of values defined in terms of previous values.

The net effect of this is that extracting the values is not straightforward and is time consuming when done by hand. In hindsight, the THG Visual Basic program should have been modified to extract the data. Furthermore, to produce a practical test method, some integration within the tool chain is required.

6.6.2 *t*-way Generation Tools

During the course of the work undertaken here two different tools have been used to generate *t*-way adequate test sets - an implementation of the AETG algorithm as presented in Cohen *et al.* [67] and the other, `jenny` [182], a tool taken from the internet. Each of these tools has its own advantages and disadvantages.

As stated in section 5.3.1, the implementation of the AETG algorithm was far too slow to be of practical use for more than three factors. This was this tools major drawback. However, as implemented the tool has two advantages over the `jenny` tool.

The first advantage is in data input, the AETG implementation takes a list of variables and the values that they can take on as shown in Figure 16. Here, the first line specifies the seed for the random number generator as defined by Wichmann and Hill [329], the second tells the tool how many parameters to expect (18) and how many values to use for each of the variables. The following lines specify the values to be used for those variables. Some data lines specify Boolean values (e.g. line 3 with 666, 667) which are replaced manually using a text editor, and some specify actual values (e.g. line 7 with 0 5139062 and 10278125), which can be used directly.

In contrast, the `jenny` tool takes the generation specification as a set of program parameters on the command line. This corresponds directly with the second line in Figure 16. As `jenny` has no information on the values each variable should take on it assigns alphanumeric strings to each output in the form a1, a2, ... a*n* for the first input; b1, b2 ... b*n* for the second and so on. This means that a translation step between the representation used in `jenny` and the values required by the test harness is required.

The second advantage for the AETG implementation is that the output is in the correct format to be directly included into the mutant driver program. Output from the `jenny` tool however has to be post processed; however using an adapted input stage from the AETG implementation to convert its output into the same format as the output from my AETG implementation reduces the size of the problem.

```
1
18  2  2  2  3  3  3  4  3  2  3  3  2  2  3  3  3  3  3
666 667
666 667
766 767
0 1280 2560
0 5139062 10278125
820000 4111250 8222500
881 882 883 884
0 4400 8800
555 555
0 1020 2040
0 1020 2400
555 555
666 667
62 311 622
0 1280 2560
0  127 255
0 1200 2400
0 1280 2560
```

Fig. 16. Input format for the AETG based tool for the `_gov_ffd_rpm` function.

The AETG implementation also has one further advantage. The hand-generated test vectors included data for values that were defined as inputs but that were not directly used by the code under test. For example, where code access a specific array element, the hand-generated test initializes the target element and those either side to known values. This allows off by one errors to be more easily detected. The AETG implementation can take this into account but the `jenny` tool cannot. With `jenny`, initialization data such as this has to be inserted by hand.

The `jenny` tool is however not without its own advantages. The clearest of these is the speed with which it can generate the vectors sets. Also there is a slight advantage in the number of vectors generated though this is not large.

There may also be an advantage in the way in which the `jenny` tool allocates values to variables that have already been completely removed from the set of *t*-way tuples yet to be covered. The AETG implementation uses the first value defined for each variable in this case whereas the `jenny` tool uses a series of values which seems to give a slightly better mutant kill rate. The fact that the AETG algorithm as presented in Cohen *et al.* [67] does not define what should happen has been previously noted by Cohen *et al.* [76] but has not been investigated further as part of the work presented here.

6.6.3 Csaw

The Csaw manual [111] gives an outline of the process for using the tool as follows:

- Put the function or functions of interest into the required format.
- Add the names of global variables etc. using the special operators provided.
- Compile the code.
- Remove those mutants that do not compile.

While this appears reasonably straight forward, in practice it can be quite time consuming. For example, the process of preparing the functions while conceptually simple is subject to human error, which can affect the quality of the mutants. Also if a statement that is spread over two lines is not modified so that it is on a single line, the statement deletion operator will not function correctly. Although a minor point, this does affect both the integrity of the mutation process and which mutations compile.

Removal of the mutations that do not compile is also very time consuming. In the initial work these had to be removed by hand and a dummy function left in their place, but for the larger functions this was impractical and the mutation tool `line.c` was modified to take a list of mutations to be removed automatically. However, selecting the mutations to be removed remains a manual operation that requires a scan of the error file that is produced by the `gcc` compiler. Ideally, extraction of failed function identifiers should be automated but the multi-line nature of the error messages in the output file makes this awkward.

6.6.4 Process Integration

Given the points above it will be obvious that integration within and between the tools is fairly minimal. Although this did not directly affect the quality of results for individual functions it did affect the number of functions that could be examined.

In conclusion, therefore, to make the technique explored in this work a practical proposition for a production environment requires two main improvements. First, the test data generation tool chain needs to be improved so that no or only minimal work is required to convert the data specification into a usable set of test vectors. Second, that the mutation process needs to be more fully automated to remove at least some of the work that currently has to be performed by hand.

6.7 Summary

In summary what has been shown here is as follows:

- That 2-way test sets for unit testing do not appear to be competitive with high quality hand-generated test sets in terms of effectiveness.
- That *t*-way adequate test sets of factor 5 (or above) appear to be as effective as hand-generated test sets.
- That it is possible to minimise the test sets created using *t*-way adequate techniques such that, there is no loss of detection ability and the minimised test set is small enough to be validated by hand.

Several other interesting points have been raised by the experiments reported in this thesis. These include:

- That the base choice method formalised by Ammann and Offutt [9] is not competitive with either *t*-way adequate test sets, hand-generated test set or even with random testing.
- That random testing can be very effective on functions that have low "complexity", for example, it performed well on sorting functions and in the first half of the functions drawn from an industrial project.

This last point is potentially significant for a large amount of empirical work on the effectiveness of software testing techniques. As noted in section 2.2.5 high complexity does not seem to be a feature of many functions used in testing research.

7. Future Work

7.1 Introduction

The experimental work described in this thesis was focused on determining two things. First, it asked whether *t*-way adequate test sets are "reliable" in the sense that they do at least as well as hand-generated test sets. Second, it set out to show that there is a workable procedure for reducing the number of test vectors that need to be considered by hand. That is, to make the "oracle problem" tractable.

To address the first point, several experiments were performed on industrial code that compared the ability of *t*-way adequate test sets up to factor five to kill code mutants with the ability of hand-generated test sets and of a small number of automatically generated test sets to do the same thing. These experiments showed that *t*-way adequate test sets were comparable in performance with the hand-generated test sets and appeared to show an advantage over the other techniques with which it was compared.

To address the second point, a "brute force" optimisation process used in batches ($t = 2$, 3, 4 and 5) was applied that recorded which vectors killed which mutants and successively selected the best. This approach conclusively showed that a small set of test vectors can be extracted from a much larger set of vectors with *no* loss in the ability to distinguish mutants from the original code.

The work is, however, incomplete in a number of ways and the purpose of this chapter is to explore possible routes by which the work could be expanded and extended in order to increase both confidence in the reliability of the results and to increase the utility of the results.

7.2 Code Variety

The code selection criteria are given in section 5.2.4.1 and this can lead to several possible sources of bias, some of which are given in section 6.6. Although it is believed that the code used is representative of high quality, real-time embedded code, there are a small number of issues associated with the sample as follows:

- Only a small percentage of the actual code that makes up the project has been examined. Moreover, no single module has been examined in detail and there is a bias towards selecting functions with more complex logic.
- The code used was taken from a single project.
- The code was selected to avoid dealing with complex data structures.
- The test has been applied to single functions, i.e., unit testing.

To address the first and last points, one possibility is to test a complete code module, starting with unit tests of the same form that have been used in the empirical work reported here before moving on to testing combinations of functions and then attempting to build up to a complete module test. This last point may seem unnecessary, however as pointed out in section 3.4.2.5 this is the procedure used by the majority of empirical work done in combinatorial testing to date. Given the difference in results reported by Kuhn and Okun [201] and Schroeder, Bolaki and Gopu [277], this may throw some light on what the possible issues that arise when the technique is applied to complete programs, or large sections of programs. What effect this has on the ability of a vector set to kill mutants is poorly understood but would be of considerable interest to practitioners to be able to reliably test larger conglomerations or units[58].

To address the second point, there is only one possible solution, which is to use code from another project. However, this is not a trivial matter because to be able to make comparisons with the work performed here; unit test sets constructed by hand are required. To be able to satisfy this requirement the unit tests for the Boar project reported in Ellims, Bridges and Ince [112] have been extracted from the project archive and these may provide an interesting comparison. Unit tests for the Boar project were outsourced and it is known that there is a significant difference between Wallace and Boar in what activity in the unit test process (test design versus test run) errors were revealed.

In addition to this, a copy of the TCAS program investigated in Kuhn and Okun [201] and used by Andrews, Briand and Labiche [11] has recently been obtained which this

[58] The author was involved in extended discussions on this topic when defining "software unit" for the draft of the ISO 26262 standard.

should allow a direct connection between the work reported here and the work performed by those authors to be established.

7.3 Structured Data

The Wallace system has functions within it with higher complexity than those that were used in this study. However, the inputs to these functions are large arrays of one or more dimensions and it is not obvious how *best* to deal effectively with these data structures. For example, does one treat the array (or other structure) as a collection of individual variables or as a complete unit where one selects between various predetermined options?

The approach taken in this work to data is the former. As typified by the work done with the sort functions in section 5.4.1, each array element has been dealt with as if it were an isolated variable

However, this approach would probably not work in situations in which the elements of an array have internal structure. For example, an array with two or three axes may represent an n-dimensional surface that needs to be smooth. Thus setting each element without regard to its neighbours and in turn to their nearest neighbours may not produce realistic, or more importantly, useful results.

The other situation present in the Wallace code base is where arrays are used to schedule events based on either a time frame or on an engine position as measured by the engine-timing wheel. In this case, the points where actions take place are directly dependent on the previous elements. As before, this situations could be problematic if the elements of the array are treated as separate items. In this case, however, there is a possible solution. Because the arrays are filled dynamically at run time, it is possible the problem could be abstracted one level. That is, instead of setting the array elements directly, the code used to set the elements could be called (after verification) and t-way adequate test set used to drive the generation process.

The discussion above is, of course purely speculation. Whether or not using unstructured data is actually an issue for single executions of a function as used in unit testing remains to be investigated.

7.4 Data models

As noted in section 5.5.2.3, the data selection model, i.e., minimum, middle and maximum value, that is used is in the main body of this work is possibly too simplistic. The data given in Table 21 indicates that there can be an advantage in using more complete data models. The data for the three functions in Table 20 (taken the initial experiments with 2-way adequate test sets with the AETG based tool) are shown with the data from Table 22, which used the `jenny` tool to generate the *t*-way adequate test sets in Table 30.

In Table 30, the first column contains the function names, the second column contains the data for hand-generated tests and the third contains the results for 2-way adequate tests generated by the AETG based tool. The fourth column shows the best results from Table 21 for the improved data selection process. The last four columns contain the data for the *t*-way testing using the `jenny` tool.

Table 30. Combined data from Table 20 and Table 22, the first row for each function is the number of surviving mutants and the second is the number of vectors.

	Data from Table 20			*t*-way Data From Table 22			
	Hand	AETG	Best	2-way	3-way	4-way	5-way
_sdc_fuel_control	21	107	26	101	49	25	22
	15	15	23	17	57	174	504
_sdc_pre_start	675	746		736	673		
	14	16		22	79		
_gov_gen_ffd_rpm	152	729	562	701	190	158	140
	14	18	27	21	81	299	1040

The application of a more sophisticated data selection process to the _sdc_fuel_control and _gov_gen_ffd_rpm functions showed a large improvement over both 2-way adequate test sets. For the function _sdc_fuel_control, the improvement was not matched by the simplistic data selection scheme until the 4-way test set was applied. Note that no improvement was observed for the function _sdc_pre_start. However, in isolation this does not invalidate the results because, as noted previously this function seems to be extraordinarily difficult to test.

This strongly suggests that the full set of experiments in section 5.6 should be repeated using more complete data models as outlined in section 5.5.2.3 because it is possible that the kill rates could be markedly improved. This ties in with the observations that the

production of good data models is not a trivial task and that it requires expert input as made by Dalal *et al.* [93], [92] and Bell and Vouk [32].

One final observation needs to be made here. It would also be an advantage to the tester if the Csaw tool were modified so that it could automatically collect data on which mutants could not be killed. This information would provide more concrete feedback to the tool set user on how value selections could be improved and might remove the need for some trial and error.

7.5 Oracles

The ability of the minimisation technique trailed in this thesis to select good test sets is dependent on the ability of the oracle code to detect differences in the output of the mutated and un-mutated programs. In the experiments conducted as part of this research, it seems to be a reasonable assumption that the oracles for the majority of the functions are at least adequate, given the reduction achieved in the size of the final test sets.

It is, however, a reasonable question to ask, at what point does the oracle become ineffective? To recap, the oracles consist of two parts: the un-mutated function and its associated outputs, and a function used to compare those outputs with the outputs of the mutated code. In many instances, these comparison functions compare not only the expected outputs but also the inputs to check they have not been inadvertently modified. In some cases (for example where the output is an array), they also compare values to each side of the element that is supposed to be modified.

One way to address this question might be to systematically degrade the existing comparison functions so that the comparisons that are not strictly necessary, such as checking the inputs remain unmodified, are removed. This approach should provide information on the true robustness of the optimisation techniques explored in this thesis.

It should also be possible to determine how robust the technique is in relation to errors in the un-mutated function. In the work undertaken to date, I have assumed that the actual target function is correct. This is not unreasonable given that the target functions are taken from a production system that has been in use for nearly a decade with no errors reported from the field. In addition, the target functions were developed within a quality control system that is capable of dealing with SIL 3 level software. This, however, might not be

the norm. If the purpose of testing is to discover errors in the software then at some point un-mutated code will be used that contains errors. Thus, to test the reliability of the techniques explored here, the obvious experiment would be to use mutated code as the oracle function, re-generate the mutants, and re-run the optimisation procedure. The expected result is that the final set of vectors selected should be similar to, if not the same as the optimised test sets that were obtained from the original experiments.

There is also the issue of how to improve the comparison between the un-mutated code and the mutated code. For example, in section 5.4.1 it was found that both the shellsort and quicksort functions produced identical results for the un-mutated and mutated code, even when the mutated code appears to be functionally incorrect. An initial attempt to differentiate the assumed correct code from the mutated code was made using execution time, but this was ineffective.

The obvious alternative to measuring execution time would be to internally record the number of times that the loops are iterated and compare those values for the mutated and un-mutated code. It should be a reasonably simple matter to add the necessary code by hand or to have the Csaw tool add the necessary code. However, this is perhaps a partial solution, a more complete solution is to record the complete execution of the function being tested be employing the techniques that are used with watchdog processors for control flow checking. For example, it should be possible to add assigned node signatures and compare the final signatures for both the mutated and un-mutated functions, Mahmood and McCluskey [218] give a brief survey of relevant techniques.

7.6 Optimisation

The optimisation technique trailed here is a general purpose technique and as applied only has only a minor dependency on the *t*-way adequate test sets in that it is applied sequentially to each larger factor.

Thus, the optimisation process could also be applied to the data sets generated using random selection. It would be interesting to compare the size of the optimised test sets obtained by random selection with the optimised test sets derived for *t*-way adequate tests. In general, this comparison can be used as a comparative measure of the effectiveness of

the two techniques. I would expect stronger techniques to produce smaller optimised sets of vectors than a weaker technique.

Given that the technique can be applied to random testing, there is no reason why the technique cannot also be applied to techniques such as Malaiya's [219] anti-random testing or to boundary following techniques as suggested by Hoffman *et al.* [164].

8. References

[1] Abdellatif-Kaddour, O., Thvenod-Fosse, P., and Waeselynck, H., "Property-oriented testing: a strategy for exploring dangerous scenarios," in *Proceedings of the 2003 ACM symposium on Applied computing*. Melbourne, Florida: ACM Press, 2003.

[2] Abrial, J. R., Borger, E., and Langmaack, H. E., *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, vol. 1165. Berlin: Springer, 1996.

[3] Ackermann, J., "Robust control prevents car skidding," *Control Systems Magazine*, vol. 17, pp. 23-31, 1997.

[4] ADAC, "Die ADAC Pannenstatistik," in *Motorwelt*, vol. 5, 1989.

[5] Agrawal, H., DeMillo, R. A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E. W., Martin, R. J., Mathur, A. P., and Spafford, E., "Design of Mutant Operators for the C Programming Language," Department of Computer Science, Purdue University, W. Lafayette SERC–TR–41–P, April 12, 2006 2006.

[6] Alander, Mantere, and Moghadampour, "Testing software response times using a genetic algorithm," in *Proceedings of 3rd Nordic Workshop on Genetic algorithms and their applications (3NWGA)*, 1997.

[7] Alander, Mantere, and Turunen, "Genetic algorithm based software testing," in *Proceedings of International Conference (ICANNGA97)*. Norwich (UK): Springer-Verlag, 1997.

[8] Ammann, P. and Offutt, J., *Introduction to Software Testing*. Cambridge: Cambridge University Press, 2008.

[9] Ammann, P. E. and Offutt, J., "Using Formal Methods to Derive Test Frames in Category-Partition Testing," presented at Proc. of 9th Annual Conf. on Computer Assurance (COMPASS'94), Gaitersburg, Maryland, USA, 1994.

[10] Andrews, D. M. and Benson, J. P., "An automated program testing methodology and its implementation," in *Proceedings of the 5th international conference on Software engineering*. San Diego, California, United States: IEEE Press, 1981.

[11] Andrews, J. H., Briand, L. C., and Labiche, Y., "Is Mutation an Appropriate Tool for Test Experiments?," presented at Proc. of the 27th Int'l Conf. on Software Engineering, St. Louis, MO, USA, 2005.

[12] Annon, "NIST/SEMATECH e-Handbook of Statistical Methods," 2008.

[13] Anon, *lEEE Std 610.121990: IEEE Standard Glossary of Software Engineering Terminology*. New York: IEEE, 1990.

[14] Anon, "DO-178B Software Considerations in Airborne Systems and equipment certification," RTCA, Washington 1992.

[15] Anon, *Quality systems -- Model for quality assurance in design, development, production, installation and servicing*: ISO, 1994.

[16] Anon, "Functional Safety of Electrical/Electronic/Programmable electronic safety-related systems, Part 1: General Requirements, BS EN 61508-1:2002," British Standards, 2002.

[17] Anon, *ISO/IEC 15504:2004 Information technology - Process assessment* ISO, 2004.

[18] Anon, "MISRA-C:2004 Guidelines for the use of the C language in critical systems.," MIRA Limited, Nuneaton 2004.

[19] Anon, "Numerical Algorithms Group," vol. 2006, 2006.

[20] Apt, K. R., *Principles of constraint programming*. Cambridge: Cambridge university Press, 2003.

[21] Ashlock, D., "Finding designs with genetic algorithms," in *Computational and Constructive Design Theory*, Wallis, W. D., Ed. Bristol: Kluwer Academic Publishers, 1996, pp. 49-65.

[22] Baresel, A., Binkley, D., Harman, M., and Korel, B., "Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. Boston, Massachusetts, USA: ACM Press, 2004.

[23] Baresel, A., Pohlheim, H., and Sadeghipour, S., "Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms," in *Proceedings of Genetic and Evolutionary Computation Conf. (GECCO'03)*. Chicago, USA.: Springer, 2003.

[24] Baresel, A. and Sthamer, H., "Evolutionary testing of flag conditions," in *Proceedings of Genetic and Evolutionary Computation Conf. (GECCO'03)*. Chicago, USA.: Springer, 2003.

[25] Baresel, A., Sthamer, H., and Schmidt, M., "Fitness Function Design To Improve Evolutionary Structural Testing," in *Proceedings of the Genetic and Evolutionary Computation Conference*: Morgan Kaufmann Publishers Inc., 2002.

[26] Barnes, J., *High integrity Ada: the SPARK approach*. Harlow: Addison Wesley Longman, 1997.

[27] Bartak, R., "On-Line Guide to Constraint Programming," 1998.

[28] Baudry, B., Fleurey, F., Jzquel, J.-M., and Le Traon, Y., "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*: IEEE Computer Society, 2002.

[29] Baudry, B., Hanh, V. L., Jezequel, J. M., and Le Traon, Y., "Building Trust into OO Components Using a Genetic Analogy," in *Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*: IEEE Computer Society, 2000.

[30] Baudry, B., Hanh, V. L., and Le Traon, Y., "Testing-for-Trust: The Genetic Selection Model Applied to Component Qualification," in *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*: IEEE Computer Society, 2000.

[31] Beizer, B., *Software testing techniques (2nd ed.)*: Van Nostrand Reinhold, 1990.

[32] Bell, K. Z. and Vouk, M. A., "Effectiveness of Stochastically Generated Dependencies in Pairwise Testing," in *Supplementary Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*. Saint Malo, France: ACM Press, 2004.

[33] Bertolino, A. and Marre, M., "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 885-899, 1994.

[34] Bhansali, P. V., "The MCDC paradox," *SIGSOFT Softw. Eng. Notes*, vol. 32, pp. 1-4, 2008.

[35] Bicevskis, J., Borzovs, J., Staujums, U., Zarins, A., and Miller, E., "SMOTL - a system to construct samples for data processing program debugging," *IEEE Transactions on Software Engineering*, vol. 5, 1979.

[36] Bird, D. L. and Munoz, C. U., "Automatic generation of random self-checking test cases," *IBM Syst. J.*, vol. 22, pp. 229-245, 1983.

[37] Blanco, R., Tuya, J., Tuya, J., and Diaz, A., "A scatter search approach for automated branch coverage in software testing," in *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP'03)*: IEEE Computer Society, 2003.

[38] Boland, P. J., Singh, H., and Cukic, B., "Comparing Partition and Random Testing via Majorization and Schur Functions," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 88-94, 2003.

[39] Bottaci, L., "Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithms," presented at Proc. of the Genetic and Evolutionary Computation Conference, 2002.

[40] Bottaci, L., "Predicate expression cost functions to guide evolutionary search for test data," in *Proceedings of Genetic and Evolutionary Computation Conf. (GECCO'03)*. Chicago, USA.: Springer, 2003.

[41] Boyer, R. S., Elspas, B., and Levitt, K. N., "SELECT - a formal system for testing and debugging programs by symbolic execution," *SIGPLAN Not.*, vol. 10, pp. 234-245, 1975.

[42] Brownlie, R., Prowse, J., and Phadke, M. S., "Robust Testing of AT&T PMX/StarMAIL Using Oats," *AT&T Technical Journal*, vol. 71, pp. 41-47, 1992.

[43] Bryce, R. C. and Colbourn, C. J., "Test prioritization for pairwise interaction coverage," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1-7, 2005.

[44] Bryce, R. C. and Colbourn, C. J., "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, pp. 960-970, 2006.

[45] Bryce, R. C. and Colbourn, C. J., "One-Test-at-a-Time heuristic Search for Interaction Test suites," presented at Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07), London, England, 2007.

[46] Bryce, R. C., Colbourn, C. J., and Cohen, M. B., "A framework of greedy methods for constructing interaction test suites," in *Proceedings of the 27th international conference on Software engineering*. St. Louis, MO, USA: ACM Press, 2005.

[47] Buehler, O. and Sthamer, H., "Evolutionary functional testing of a vehicle brake assistant system," in *The Six Metaheuristics International Conference Vienna (MIC2005)*: Morgan Kaufmann Publishers Inc., 2005.

[48] Buehler, O. and Wegener, J., "Evolutionary functional testing of an automated parking system," in *Proceeding of the Int'l Conf. on Computers, Communications & Control Technologies (CCCT'03) and 9th Annual Int'l Conf. on Information Systems Analysis and Synthesis (ISAS'03)*. Florida USA: IEEE, 2003.

[49] Bueno, P. M. S. and Jino, M., "Automatic test data generation for program paths using genetic algorithms," *International Journal of Software Engineering and Knowledge Engineering*, vol. 12, 2002.

[50] Burr, K. and Young, W., "Combinatorial test techniques: table-based automation, test generation and code coverage," in *Intl. Conf. on Software Testing, Analysis, and Review (STAR)*. San Diego, CA, 1998.

[51] Burroughs, K., Jain, A., and Erickson, R. L., "Improved quality of protocol testing through techniques of experimental design," in *IEEE International Conference on Communications (SUPERCOMM/ICC'94)*. New Orleans, LA, USA: IEEE, 1994.

[52] Cameron, P. J., *Combinatorics: Topics, Techniques, Algorithms*. Cambridge: Cambridge University Press, 1994.

[53] Chan, K. P., Chen, T. Y., and Towey, D., "Restricted Random Testing," in *Proceedings of the 7th International Conference on Software Quality*: Springer-Verlag, 2002.

[54] Chen, T. Y., Cheung, S. C., and Yiu, S. M., "Metamorphic testing: a new approach for generating next test cases," Department of Computer Science, Hong Kong University of Science and Technology HKUST-CS98-01, 1998.

[55] Chen, T. Y., Tse, T. H., and Yu, Y. T., "Proportional sampling strategy: a compendium and some insights," *J. Syst. Softw.*, vol. 58, pp. 65-81, 2001.

[56] Chen, T. Y. and Yu, Y. T., "On the Relationship Between Partition and Random Testing," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 977-980, 1994.

[57] Chen, T. Y. and Yu, Y. T., "On the Expected Number of Failures Detected by Subdomain Testing and Random Testing," *IEEE Trans. Softw. Eng.*, vol. 22, pp. 109-119, 1996.

[58] Chen, T. Y. and Yu, Y. T., "On the Test Allocations for the Best Lower Bound Performance of Partition Testing," in *Proceedings of the Australian Software Engineering Conference*, 1998.

[59] Cheon, Y. and Kim, M., "A fitness function for modular evolutionary testing of object-oriented programs," Dept. of Computer Science, University of Texas at El Paso TR #05-35, 2005.

[60] Chilenski, J. J. and Miller, S. P., "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Eng. Jnrl.*, pp. 193-200, 1994.

[61] Claessen, K. and Hughes, J., "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*: ACM Press, 2000.

[62] Clarke, J., Dolado, J. J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., and Shepperd, M., "Comparing the effectiveness of software testing strategies," *IEE Proceedings- Software*, vol. 150, pp. 161- 175, 2003.

[63] Clarke, L. A., "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, pp. 215-222, 1976.

[64] Clarke, L. A., Richardson, D. J., and Zeil, S. J., "TEAM: a support environment for testing, evaluation, and analysis," in *Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. Boston, Massachusetts, United States: ACM Press, 1988.

[65] Clocksin, W. F. and Mellish, C. S., *Programming in Prolog*. Berlin: Springer-Verlag, 1994.

[66] Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G., "Method and system for automatically generating efficient test cases for systems having interacting elements," Office, U. S. P., Ed., 1996.

[67] Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C., "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 437-444, 1997.

[68] Cohen, D. M., Dalal, S. R., Kajla, A., and Patton, G. C., "The Automatic Efficient Test Generator (AETG) System," presented at Proc. 5th International Symposium on Software Reliability Engineering, Monterey, CA, USA, 1994.

[69] Cohen, D. M., Dalal, S. R., Parelius, J., and Patton, G. C., "Efficacy of AETG Test Cases as measured by code coverage, ," vol. 2008, 1995, pp. Technical Memorandum.

[70] Cohen, D. M., Dalal, S. R., Parelius, J., and Patton, G. C., "The Combinatorial Design Approach to Automatic Test Generation," *IEEE Softw.*, vol. 13, pp. 83-88, 1996.

[71] Cohen, J., "Constraint logic programming languages," *Commun. ACM*, vol. 33, pp. 52-68, 1990.

[72] Cohen, M. B., Colbourn, C. J., and Ling, A. C. H., "Augmenting Simulated Annealing to Build Interaction Test Suites," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*: IEEE Computer Society, 2003.

[73] Cohen, M. B., Dwyer, M. B., and Shi, J., "Exploiting Constraint Solving History to Construct Interaction Test Suites," presented at Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART-MUTATION 2007), Windsor, England, 2007.

[74] Cohen, M. B., Dwyer, M. B., and Shi, J., "Interaction testing of highly-configurable systems in the presence of constraints," presented at Proceedings of the 2007 international Symposium on Software Testing and Analysis (ISSTA '07), London, United Kingdom, 2007.

[75] Cohen, M. B., Dwyer, M. B., and Shi, J., "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach," *IEEE Transactions on Software Engineering*, vol. 34, pp. 633-650, 2008.

[76] Cohen, M. B., Gibbons, P. B., Mugridge, W. B., and Colbourn, C. J., "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering*. Portland, Oregon: IEEE Computer Society, 2003.

[77] Cohen, M. B., Gibbons, P. B., Mugridge, W. B., Colbourn, C. J., and Collofello, J. S., "Variable Strength Interaction Testing of Components," in *Proceedings of the 27th Annual International Conference on Computer Software and Applications*: IEEE Computer Society, 2003.

[78] Colbourn, C. J., Chen, Y., and Tsai, W., "Progressive Ranking and Composition of Web Services Using Covering Arrays," presented at Proceedings of the 10th IEEE international Workshop on Object-Oriented Real-Time Dependable Systems, Washington, DC, 2005.

[79] Colbourn, C. J., Cohen, M. B., and Turban, R. C., "A deterministic density algorithm for pairwise interaction coverage," in *Proc. of the IASTED Intl. Conference on Software Engineering*, 2004.

[80] Cooper, D. W., "Adaptive testing," in *Proceedings of the 2nd international conference on Software engineering*. San Francisco, California, United States: IEEE Computer Society Press, 1976.

[81] Copeland, L., *A Practitioner's Guide to Software Test Design*. Boston: Artech House Publishers, 2004.

[82] Coward, D. and Ince, D. C., *The symbolic execution of software; THE sym-bol SYSTEM*. lONDON: cHAPMAN & hALL, 1995.

[83] Coward, P. D., "A review of software testing," *Inf. Softw. Technol.*, vol. 30, pp. 189-198, 1988.

[84] Coward, P. D., "Symbolic execution and testing," *Inf. Softw. Technol.*, vol. 33, pp. 53-64, 1991.

[85] Cuéllar, J. R. and Wildgruber, I., "The real-time behavior of the steam-boiler," in *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, vol. 1165, Abrial, J. R., E. Borger, and H. Langmaack, Eds. Berlin: Springer, 1996, pp. 184-202.

[86] Cytron, R., Ferrante, J., Brosen, B. K., Wegman, M. N., and Zadeck, F. K., "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451-490, 1991.

[87] Czerwonka, J., "Pairwise testing in real world," presented at Pacific Northwest Software Quality Conference, 2006.

[88] Dadeau, F., Ledrun, Y., and Du Bousquet, L., "Directed Random Reduction of Combinatorial Test Suites," presented at Proc. Second Int'l Workshop on Random Testing (RT'07), Atlanta, GA (USA), 2007.

[89] Daich, G. T., "New spreadsheet tool helps determine minimal set test parameter combinations," *STSC CrossTalk*, 2003.

[90] Daich, G. T., "Testing combinations of parameters made easy," in *Proceedings of AUTOTESTCON 2003. IEEE Systems Readiness Technology Conference*. Boston, Massachusetts, USA: IEEE, 2003.

[91] Dalal, S., Jain, A., Karunanithi, N., Leaton, J., and Lott, C., "Model-based Testing of a Highly Programmable System," presented at Proc. of the Ninth International Symposium on Software Reliability Engineering, 1998.

[92] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M., "Model-based Testing in Practice," presented at Proc. of the 21st Int'l Conf. on Software Engineering, Los Angeles, California, United States, 1999.

[93] Dalal, S. R. and Mallows, C. L., "Factor-Covering Designs for Testing Software," *Technometrics*, vol. 40, pp. 234-243, 1998.

[94] Daley, N., Hoffman, D., and Strooper, P., "A framework for table driven testing of Java classes," *Softw. Pract. Exper.*, vol. 32, pp. 465-493, 2002.

[95] Daran, M. and Thevenod-Fosse, P., "Software Error Analysis: a Real Case Study Involving Real Faults and Mutations," *SIGSOFT Softw. Eng. Notes*, vol. 21, pp. 158-171, 1996.

[96] Darringer, J. A. and King, J. C., "Applications of symbolic execution to program testing," *IEEE Computer*, pp. 51-59, 1978.

[97] Deason, W. H., Brown, D. B., Chang, K. H., and J. H. Cross, I., "A Rule-Based Software Test Data Generator," *IEEE Transactions on Knowledge and Data Engineering*, vol. 3, pp. 108-117, 1991.

[98] DeMillo, R. A., Lipton, R. J., and Sayward, F. G., "Hints on Test Data Selection: Help for the Practising Programmer," *Computer*, pp. 34-41, 1978.

[99] DeMillo, R. A. and Offutt, A. J., "Constraint-Based Automatic Test Data Generation," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 900-910, 1991.

[100] DeMillo, R. A. and Offutt, A. J., "Experimental results from an automatic test case generator," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, pp. 109-127, 1993.

[101] Diamond, W. J., *Practical Experiment Design For Engineers and Scientists*. New York: John Wiley & Sons, 2001.

[102] Diaz, E., Blanco, R., and Tuya, J., "Applying tabu and scatter search to automated software test case generation," in *Proc. Sixth Metaheuristic Int'l Conf.* Vienna, 2005.

[103] Diaz, E., Tuya, J., and Blanco, R., "Automated software testing using a metaheuristic technique based on tabu search," in *Proceedings of 18 IEEE Int'l Conf. on Automated Software Eng. (ASE'03)*. Boston, Massachusetts, USA: IEEE, 2003.

[104] Dillon, E. and Meudec, C., "Automatic Test Data Generation from Embedded C Code," presented at Computer Safety, Reliability, and Security 23rd Int'l Conf. (SAFECOMP 2004), Potsdam, 2004.

[105] Dorigo, M. and Gambardella, L. M., "Ant colony system: a cooperative learning approach to the travelling salesman problem," *IEEE Trans. on Evolutionary Computation*, vol. 1, pp. 53-66, 1997.

[106] Dunietz, I. S., Ehrlich, W. K., Szablak, B. D., Mallows, C. L., and Iannino, A., "Applying Design of Experiments to Software Testing: Experience Report," presented at Proc.of the 19th Int'l Conf. on Software Eng., Boston, Massachusetts, United States, 1997.

[107] Duran, J. and Ntafos, S., "An Evaluation of Random Testing," *IEEE Trans. Softw. Eng.*, vol. 10, pp. 438-444, 1984.

[108] Edvardsson, J., "A Survey on Automatic Test Data Generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*. Linkoping, 1999.

[109] Eiben, A. E. and Smith, J. E., *Introduction to evolutionary computing*. Berlin: Springer-Verla, 2003.

[110] Ellims, M., "On wheels, nuts and software," in *Proceedings of the 9th Australian workshop on Safety critical systems and software - Volume 47*. Brisbane, Australia: Australian Computer Society, Inc., 2004.

[111] Ellims, M., "The Csaw Mutation Tool Users Guide," Department of Computer Science, Open University 2007.

[112] Ellims, M., Bridges, J., and Ince, D. C., "The Economics of Unit Testing," *Empirical Softw. Eng.*, vol. 11, pp. 5-31, 2006.

[113] Ellims, M., Ince, D., and Petre, M., "AETG vs. Man: an Assessment of the Effectiveness of Combinatorial Test Data Generation," Department of Computer Science, Open University 2007/08, 2007.

[114] Ellims, M., Ince, D., and Petre, M., "The Csaw C Mutation Tool: Initial Results," presented at Mutation 2007, Windsor, UK, 2007.

[115] Ellims, M. and Jackson, K., "ISO 9001: Making the Right Mistakes," in *SAE World Congress*. Detroit, USA: SAE, 1999.
[116] Ellims, M. and Parkins, R. P., "Unit Testing Techniques and Tool Support," in *SAE World Congress*. Detroit, USA: SAE, 1999.
[117] Ellims, M. and Zurlo, J. R., "Feedforward Engine Control Governing System," Office, U. s. P., Ed. USA, 2003.
[118] Ferguson, R. and Korel, B., "The chaining approach for software test data generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, pp. 63-86, 1996.
[119] Fetzer, J. H., "Program Verification: the Very Idea," *Comm. of the ACM*, vol. 31, pp. 1048-1063.
[120] Fowler, J. and Cohen, L., *Practical Statistics for Field Biology*. Chichester: John Wiley & Sons 1990.
[121] Frankl, Weiss, S. N., and Hu, C., "All-uses vs. mutation testing: an experimental comparison of effectiveness," *J. Syst. Softw.*, vol. 38, pp. 235-253, 1997.
[122] Frankl, P. G. and Deng, Y., "Comparison of delivered reliability of branch, data flow and operational testing: A case study," *SIGSOFT Softw. Eng. Notes*, vol. 25, pp. 124-134, 2000.
[123] Frankl, P. G., Hamlet, R. G., Littlewood, B., and Strigini, L., "Evaluating Testing Methods by Delivered Reliability," *IEEE Trans. Softw. Eng.*, vol. 24, pp. 586-601, 1998.
[124] Frankl, P. G. and Iakounenko, O., "Further empirical studies of test effectiveness," *SIGSOFT Softw. Eng. Notes*, vol. 23, pp. 153-162, 1998.
[125] Frankl, P. G. and Weiss, S. N., "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Trans. Softw. Eng.*, vol. 19, pp. 774-787, 1993.
[126] Frankl, P. G. and Weyuker, E. J., "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Softw. Eng.*, vol. 14, pp. 1483-1498, 1988.
[127] Freedman, R. S., "Testability of Software Components," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 553-564, 1991.
[128] Gallagher, M. J. and Narasimhan, V. L., "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 473-484, 1997.
[129] Girgis, M. R., "An experimental evaluation of a symbolic execution system," *Softw. Eng. J.*, vol. 7, pp. 285-290, 1992.
[130] Glass, H. and Cooper, L., "Sequential Search: A Method for Solving Constrained Optimization Problems," *J. ACM*, vol. 12, pp. 71-82, 1965.
[131] Godefroid, P., Klarlund, N., and Sen, K., "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. Chicago, IL, USA: ACM Press, 2005.
[132] Goldberg, A., Wang, T. C., and Zimmerman, D., "Applications of feasible path analysis to program testing," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. Seattle, Washington, United States: ACM Press, 1994.
[133] Goldberg, D., "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, pp. 5-48, 1991.
[134] Goodenough, J. B. and Gerhart, S. L., "Toward a theory of test data selection," *IEEE Trans. Softw. Eng.*, vol. 1, pp. 156-173, 1975.

[135] Gotlieb, A., "Exploiting Symmetries to Test Programs," presented at Proceedings of the 14th International Symposium on Software Reliability Engineering, 2003.

[136] Gotlieb, A., Botella, B., and Rueher, M., "Automatic test data generation using constraint solving techniques," *SIGSOFT Softw. Eng. Notes*, vol. 23, pp. 53-62, 1998.

[137] Gouraud, S. D., Denise, A., Gaudel, M. C., and Marre, B., "A New Way of Automating Statistical Testing Methods," in *Proceedings of the 16th IEEE international conference on Automated software engineering*: IEEE Computer Society, 2001.

[138] Grindal, M., Lindström, B., Offutt, A. J., and Andler., S. F., "An Evaluation of Combination Strategies for Test Case Selection," Department of Computer Science, University of Skövde HS-IDA-TR-03-001, 2003 2003.

[139] Grindal, M., Lindström, B., Offutt, J., and Andler, S. F., "An evaluation of combination strategies for test case selection," *Empir Software Eng*, vol. 11, pp. 583-611, 2006.

[140] Grindal, M., Offutt, J., and Andler, S. F., "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15, pp. 167-199, 2005.

[141] Grochtmann and Wegener, "Evolutionary testing of temporal correctness," in *Quality Week Europe 98*. Boston, Massachusetts, USA, 1998.

[142] Gross, H. G., Jones, B., and Eyres, D., "Evolutionary algorithms for the verification of execution time bounds for real-time software," in *IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems*. London, UK, 1999.

[143] Gross, H. G. and Mayer, N., "Evolutionary Testing In Component-based Real-time System Construction," in *Proceedings of the Genetic and Evolutionary Computation Conference*: Morgan Kaufmann Publishers Inc., 2002.

[144] Gross, H. G. and Mayer, N., "Search-based Execution-Time Verification in Object-Oriented and Component-Based Real-Time System Development," in *Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*: IEEE, 2003.

[145] Gupta, N., Mathur, A. P., and Soffa, M. L., "Automated test data generation using an iterative relaxation method," in *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*. Lake Buena Vista, Florida, United States: ACM Press, 1998.

[146] Gupta, N., Mathur, A. P., and Soffa, M. L., "UNA Based Iterative Test Data Generation and its Evaluation," in *Proceedings of the 14th IEEE international conference on Automated software engineering*: IEEE Computer Society, 1999.

[147] Gupta, N., Mathur, A. P., and Soffa, M. L., "Generating Test Data for Branch Coverage," in *Proceedings of the 15th IEEE international conference on Automated software engineering (ASE 2000)*: IEEE Computer Society, 2000.

[148] Gutjahr, W. J., "Partition Testing vs. Random Testing: The Influence of Uncertainty," *IEEE Trans. Softw. Eng.*, vol. 25, pp. 661-674, 1999.

[149] Hamlet, D., "Are We Testing for True Reliability?," *IEEE Softw.*, vol. 9, pp. 21-27, 1992.

[150] Hamlet, D., "Random testing," in *Encyclopedia of Software Engineering*, Marciniak, J., Ed. Bristol: Wiley, 1994, pp. 970-978.

[151] Hamlet, D., "Implementing prototype testing tools," *Softw. Pract. Exper.*, vol. 25, pp. 347-371, 1995.
[152] Hamlet, D. and Taylor, R., "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 1402-1411, 1990.
[153] Hamlet, R., "Introduction to special section on software testing," *Commun. ACM*, vol. 31, pp. 662-667, 1988.
[154] Hamlet, R. G., "Testing Programs with the Aid of a Compiler," *IEEE Trans. Softw. Eng.*, vol. 3, pp. 279-290, 1977.
[155] Harman, M., Fox, C., Hierons, R., Lin, H., Danicic, S., and Wegener, J., "VADA: A Transformation-Based System for Variable Dependence Analysis," in *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*: IEEE Computer Society, 2002.
[156] Harman, M., Lin, H., Hierons, R., Wegener, J., Sthamer, H., Andr, Baresel, and Marc, R., "Testability Transformation," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 3-16, 2004.
[157] Harman, M., Lin, H., Hierons, R. M., Baresel, A., and Sthamer, H., "Improving Evolutionary Testing By Flag Removal," in *Proceedings of the Genetic and Evolutionary Computation Conference*: Morgan Kaufmann Publishers Inc., 2002.
[158] Harrel, J. M., "Orthogonal array testing strategy (OATS) technique," 2004.
[159] Hentenryck, P., Simonis, H., and Dincbas, M., "Constraint satisfaction using constraint logic programming," *Artif. Intell.*, vol. 58, pp. 113-159, 1992.
[160] Hnich, B., Prestwich, S. D., and Selensky, "Constraint-Based Approaches to the Covering Test Problem," presented at CSCPL'04, 2005.
[161] Hnich, B., Prestwich, S. D., Selensky, E., and Smith, B. M., "Constrant Models for the Covering Test Problem," 2006.
[162] Hoare, C. A. R., "How Did Software Get So Reliable Without Proof?," in *Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*: Springer-Verlag, 1996.
[163] Hoffman, D. and Brealey, C., "Module test case generation," *SIGSOFT Softw. Eng. Notes*, vol. 14, pp. 97-102, 1989.
[164] Hoffman, D., Strooper, P., and White, L., "Boundary values and automated component testing," *Software Testing, Verification and Reliability*, vol. 9, pp. 3-26, 1999.
[165] Hoffman, D. M. and Strooper, P., "Automated Module Testing in Prolog," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 934-943, 1991.
[166] Hoskins, D., Turban, R. C., and Colbourn, C. J., "Experimental designs in software engineering: d-optimal designs and covering arrays," in *Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research.* Newport Beach, CA, USA: ACM Press, 2004.
[167] Hoskins, D. S., Colbourn, C. J., and Montgomery, D. C., "Software performance testing using covering arrays: efficient screening designs with categorical factors," presented at Proceedings of the 5th international Workshop on Software and Performance (WOSP '05), Palma, Illes Balears, Spain, 2005.
[168] Howden, W. E., "Methodoly for the generation of program test data," *IEEE Trans. Comput.*, vol. 24, pp. 554-559, 1975.

[169] Howden, W. E., "Reliability of the path analysis testing strategy," *IEEE Trans. Softw. Eng.*, vol. 2, pp. 208-215, 1976.
[170] Howden, W. E., "DISSECT - A symbolic evaluation and program testing system," *IEEE Trans. Softw. Eng.*, vol. 4, pp. 70-73, 1978.
[171] Howden, W. E., "An evaluation of the effectiveness of symbolic testing," *Softw. Pract. Exper.*, vol. 8, pp. 381-397, 1978.
[172] Howden, W. E., "Functional program Testing," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 162-169, 1980.
[173] Huller, J., "Reducing time to market with combinatorial design method testing," in *Proceedings of 10th Annual International Council on Systems Engineering (INCOSE'00)*. Minneapolis, MN, USA, 2000.
[174] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th international conference on Software engineering*. Sorrento, Italy: IEEE Computer Society Press, 1994.
[175] Ince, D. C., "The automatic generation of test data," *Computer Journal*, vol. 30, pp. 63-69, 1987.
[176] Ince, D. C., "Software Testing," in *Software Engineers Reference Book*: Butterworth-Heinemann, 1991, pp. 19/1-19/15.
[177] Ince, D. C. and Hekmatpour, S., "An empirical investigation of random testing," *The Computer Journal*, vol. 29, pp. 380, 1986.
[178] Irvine, S. A., Pavlinic, T., Trigg, L., Cleary, J. G., Inglis, S., and Utting, M., "Jumble Java Byte Code to Measure the Effectivness of Unit Tests," presented at Mutation 2007, Winsor, UK, 2007.
[179] Isermann, R., Schwarz, R., and Stolzl, S., "Fault-tolerant drive-by-wire systems," *Control Systems Magazine*, vol. 22, pp. 64-81, 2002.
[180] Jarvis, R. A., "Adaptive global search in a time variant environment using a probabilistic automaton with pattern recognition supervision," *IEEE Trans. Systems Science and Cybernetics*, vol. 6, pp. 209-217, 1970.
[181] Jasper, R., Brennan, M., Williamson, K., Currier, B., and Zimmerman, D., "Test data generation and feasible path analysis," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. Seattle, Washington, United States: ACM Press, 1994.
[182] jenny, "http://www.burtleburtle.net/bob/math."
[183] Jones, B. F., Eyres, D. E., and Sthamer, H. H., "A strategy for using genetic algorithms to automate branch and fault-based testing," *Computer Journal*, vol. 41, pp. 98-107, 1998.
[184] Jones, B. F., Sthamer, H. H., and Eyres, D. E., "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, pp. 299-306, 1996.
[185] Jones, J. A. and Harrold, M. J., "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 195-209, 2003.
[186] Kamsties, E. and Lott, C. M., "An Empirical Evaluation of Three Defect-Detection Techniques," in *Proceedings of the 5th European Software Engineering Conference*: Springer-Verlag, 1995.

[187] Kaner, C., Bach, J., and Pettichord, B., *Lessons learned in software testing: a Context driven approach*. New York: John Wiley & Sons, 2002.

[188] Kemmerer, R. A. and Eckmann, S. T., "UNISEX: A UNIX-based Symbolic Executor for Pascal," *Software - Practice and Experience*, vol. 15, pp. 439-458, 1985.

[189] Keppel, G. and Saufley, W. H., *Introduction to Design and Analysis: A Students Handbook*. San Francisco: W.H Freeman and Company, 1980.

[190] Kimberley, W., "Building the BMW 7 Series," in *Automotive Design and Production*: Gardner Publications, 2005.

[191] King, J. C., "A new approach to program testing," *SIGPLAN Not.*, vol. 10, pp. 228-233, 1975.

[192] King, J. C., "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385-394, 1976.

[193] King, K. N. and Offutt, A. J., "A Fortran language system for mutation-based software testing," *Softw. Pract. Exper.*, vol. 21, pp. 685-718, 1991.

[194] Kneuper, R., "Limits of formal methods " *Formal Aspects of Computing*, vol. 9, pp. 379-394, 1997.

[195] Knuth, D., *The Art of Computer Programming: Sorthing and Searching*, vol. 3, 2nd ed. Reading: Addison-Wesley, 1998.

[196] Kobayashi, N., Tsuchiya, T., and Kikuno, T., "A new method for constructing pair-wise covering designs for software testing," *Inf. Process. Lett.*, vol. 81, pp. 85-91, 2002.

[197] Kobayashi, N., Tsuchiya, T., and Kikuno, T., "Non-Specification-Based Approaches to Logic Testing for Software," *Information and Software Technology*, vol. 44, pp. 113-121, 2002.

[198] Kopetz, H., "Automotive Electronics," presented at 11th Euromicro Conference on Real-Time Systems, 1999.

[199] Korel, B., "Automated Software Test Data Generation," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 870-879, 1990.

[200] Korel, B. and Al-Yami, A. M., "Assertion-oriented automated test data generation," in *Proceedings of the 18th international conference on Software engineering*. Berlin, Germany: IEEE Computer Society, 1996.

[201] Kuhn, D. R. and Okun, V., "Pseudo-Exhaustive Testing for Software," presented at 30th Annual IEEE/NASA Soft. Eng. Workshop (SEW'06), 2006.

[202] Kuhn, D. R. and Reilly, M. J., "An Investigation of the Applicability of Design of Experiments to Software Testing," presented at Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02), 2002.

[203] Kuhn, D. R., Wallace, D. R., and Gallo, A. M., "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. Softw. Eng.*, vol. 30, pp. 418-421, 2004.

[204] Kuhn, R., Lei, Y., and Kacker, R., "Practical Combinatorial Testing: Beyond Pairwise " in *IT Professional*, vol. 10, 2008, pp. 19-23.

[205] Laitenberger, O., "Studying the Effects of Code Inspection and Structural Testing on Software Quality," in *Proceedings of the Ninth International Symposium on Software Reliability Engineering*: IEEE Computer Society, 1998.

[206] Lammermann, F., Baresel, A., and Wegener, J., "Evaluating Evolutionary Testability with Software-Measurements," in *Genetic and Evolutionary Computation – GECCO 2004: Genetic and Evolutionary Computation Conference*. Seattle,WA, USA: Springer, 2004.

[207] Lamont, M., "http://linux.wku.edu/~lamonml/ algor/sort/," 2006.

[208] Lapierre, S., Merlo, E., Savard, G., Antoniol, G., Fiutem, R., and Tonella, P., "Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees," in *Proceedings of the IEEE International Conference on Software Maintenance*. Oxford, UK: IEEE Computer Society, 1999.

[209] Lee, G., Morris, J., Parker, K., Bundell, G. A., and Lam, P., "Using symbolic execution to guide test generation," *Software Testing, Verification and Reliability*, vol. 15, pp. 41-61, 2005.

[210] Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., and Lawrence, J., "IPOG/IPOG-D: efficient test generation for multi-waycombinatorial testing," *Softw. Test. Verif. Reliab.*, 2007.

[211] Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., and Lawrence, J., "IPOG: A General Strategy for T-Way Software Testing," presented at 14th Annual IEEE Int'l Conf. and Workshops on the Engineering of Computer-Based Systems (ECBS'07), 2007.

[212] Lei, Y. and Tai, K. C., "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," in *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*: IEEE Computer Society, 1998.

[213] Leveson, N. G., *Safeware: system safety and computers*. New York: ACM Press, 1995.

[214] Lin, J.-C. and Yeh, P.-L., "Automatic test data generation for path testing using GAs," *Inf. Sci.*, vol. 131, pp. 47-64, 2001.

[215] Lindquist, T. E. and Jenkins, J. R., "Test-Case Generation with IOGen," *IEEE Softw.*, vol. 5, pp. 72-79, 1988.

[216] Liu, X., Liu, H., Wang, B., Chen, P., and Cai, X., "A unified fitness function calculation rule for flag conditions to improve evolutionary testing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. Long Beach, CA, USA: ACM Press, 2005.

[217] London, R. L., "Software Reliability Through Proving Programs Correct," presented at Proc. IEEE Int. Symp. Fault-Tolerant Computing, 1971.

[218] Mahmood, A. and McCluskey, E. J., "Concurrent Error Detection Using Watchdog Processors-A Survey," *IEEE Trans. Comput.*, vol. 37, pp. 160-174, 1988.

[219] Malaiya, Y. K., "Antirandom testing: getting the most out of black-box testing," presented at Proceedings., Sixth International Symposium on Software Reliability Engineering, 1995, Toulouse, France, 1995.

[220] Mandl, R., "Orthogonal LatinSsquares: an Application of Experiment Design to Compiler Testing," *Commun. ACM*, vol. 28, pp. 1054-1058, 1985.

[221] McDermid, J., Galloway, A., Burton, S., Clark, J., Toyn, I., Nigel Tracey, N., and Valentine, S., "Towards Industrially Applicable Formal Methods: Three Small Steps, and One Giant Leap," in *Proceedings of the Second IEEE International Conference on Formal Engineering Methods*: IEEE Computer Society, 1998.

[222] McDermid, J. A. and Kelly, T. P., "Safety Critical Systems: Achievement and Prediction," presented at 2nd SEAS DTC Technical Conference, Edinburgh, 2006.

[223] McDermid, J. A. and Rook, P., "Software development process models," in *Software Engineers Reference Book*, J.A., M., Ed.: Butterworth Heinemann, 1991.

[224] McMinn, P., "Search-based software test data generation: a survey," *Softw. Test. Verif. Reliab.*, vol. 14, pp. 105-156, 2004.

[225] McMinn, P., Binkley, D., and Harmann, M., "Testability transformation for efficient automated test data search in the presence of nesting," in *Proceedings of the Third UK Testing Workshop (UKTest 2005)*. Sheffield, UK, 2005.

[226] McMinn, P. and Holcombe, M., "The state problem for evolutionary testing," in *Proc. Genetic and Evolutionary Computation Conf. 2003 (GECCO'03)*: Springer, 2003.

[227] McMinn, P. and Holcombe, M., "Hybridizing evolutionary testing with the chaining approach," in *Genetic and Evolutionary Computation – GECCO 2004: Genetic and Evolutionary Computation Conference*. Seattle, WA, USA: Springer, 2004.

[228] McMinn, P. and Holcombe, M., "Evolutionary testing of state-based programs," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*. Washington DC, USA: ACM Press, 2005.

[229] Memon, A., Porter, A., Yilmaz, C., Nagarajan, A., Schmidt, D., and Natarajan, B., "Skoll: Distributed Continuous Quality Assurance," presented at ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, 2004.

[230] Meudec, C., "ATGen: automatic test data generation using constraint logic programming and symbolic execution," *Software Testing, Verification and Reliability*, vol. 11, pp. 81-96, 2001.

[231] Michael, C. and McGraw, G., "Automated Software Test Data Generation for Complex Programs," in *Proceedings of the 13th IEEE international conference on Automated software engineering*: IEEE Computer Society, 1998.

[232] Michael, C. C., McGraw, G., and Schatz, M. A., "Generating Software Test Data by Evolution," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 1085-1110, 2001.

[233] Michael, C. C., McGraw, G. E., Schatz, M. A., and Walton, C. C., "Genetic algorithms for dynamic test data generation," in *Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*: IEEE Computer Society, 1997.

[234] Michalewicz, Z. and Fogel, D. B., *How to solve it: modern heuristics*. Berlin: Springer-Verlag, 2002.

[235] Miller, W. and Spooner, D., "Automatic generation of floating-point test data," *IEEE Trans. Softw. Eng.*, vol. 2, pp. 223- 226, 1976.

[236] Mueller, F. and Wegener, J., "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," in *Fourth IEEE Real Time Technology and Applications Symp*: IEEE, 1998.

[237] Myers, G. J., "A controlled experiment in program testing and code walkthroughs/inspections," *Commun. ACM*, vol. 21, pp. 760-768, 1978.

[238] Myers, G. J., *Art of Software Testing*: John Wiley & Sons, 1979.

[239] Nair, V. N., James, D. A., Ehrlich, W. K., and Zevallos, J., "A Statistical Assessment of some Software Testing Strategies and Application of Experimental Design Techniques," *Statistica Sinica.*, vol. 8, pp. 165-184, 1998.

[240] Naur, P., "Programming by action clusters," *BIT*, vol. 9, pp. 250-258, 1969.
[241] Neumann, P. G., *Computer Related Risks*. New York: The ACM Press, 2003.
[242] Nikolik, B. and Hamlet, D., "Solving constraints involving indexed variables," Portland State University, Portland 1997.
[243] Nilsson, R. and Henriksson, D., "Test case generation for flexible real-time control systems," in *Proceedings of the Tenth IEEE Int'l conf. on Emerging Technologies and Factory Automation*. Catania, Italy, 2005.
[244] Ntafos, S. C., "On Comparisons of Random, Partition, and Proportional Partition Testing," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 949-960, 2001.
[245] Nurmela, K. J. and Ostergard, P. R. J., "Constructing covering designs by simulated annealing," Helsinki University of Technology Digital Systems Laboratory Series B No. 10, 1993.
[246] Offutt, A. J., "The coupling effect: fact or fiction.," *SIGSOFT Softw. Eng. Notes*, vol. 14, pp. 131-140, 1989.
[247] Offutt, A. J., "Investigations of the Software Testing Coupling Effect," *ACM Trans. on Soft. Eng. and Methodology*, vol. 1, pp. 5-20, 1992.
[248] Offutt, A. J., "A Practical System for Mutation Testing: Help for the Common Programmer," presented at Proc. of the IEEE Int'l Test Conference on TEST: The Next 25 Years, 1994.
[249] Offutt, A. J., "An Integrated automatic test data generation system," *Journal of Systems Integration*, vol. 1, pp. 391 - 409, 2003.
[250] Offutt, A. J., Jin, Z., and Pan, J., "The dynamic domain reduction procedure for test data generation," *Softw. Pract. Exper.*, vol. 29, pp. 167-193, 1999.
[251] Offutt, A. J. and King, K. N., "A Fortran 77 interpreter for mutation analysis," in *Papers of the Symposium on Interpreters and interpretive techniques*. St. Paul, Minnesota, United States: ACM Press, 1987.
[252] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C., "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, pp. 99-118, 1996.
[253] Offutt, A. J. and Voas, J. M., "Subsumption of Condition Coverage Techniques by Mutation Testing," Dept. of Information and Software Systems Engineering , George Mason Univ., Fairfax, Va. ISSE-TR-96-1OO, 1996.
[254] Offutt, J. and Seaman, E. J., "An Integrated automatic test data generation system," in *Proceedings of the Fifth Annual Conference on Systems Integrity, Software Safety and Process Security (COMPASS '90)*. Gaithersburg, MD, USA, 1990.
[255] Offutt, J. A., Pan, J., and Voas, J. M., "Procedures for Reducing the Size of Coverage Based Test Sets," presented at Twelfth Int. Conf. on Testing Computer Software., Washington D.C., 1995.
[256] Ostrand, T. J. and Balcer, M. J., "The category-partition method for specifying and generating functional tests," *Commun. ACM*, vol. 31, pp. 676-686, 1988.
[257] Ould, M. A., "Testing: a challenge to method and tool developers," *Softw. Eng. J.*, vol. 6, pp. 59-64, 1991.
[258] Pan, J., Koopman, P., and Siewiorek, D., "A dimensionality model approach to testing and improving software robustness," in *Proceedings of the IEEE Systems Readiness Technology Conference, AUTOTESTCON '99*. San Antonio, TX, USA: IEEE, 1999.

[259] Parasoft, " C++Test version 2.2," 2004.
[260] Pargas, R. P., Harrold, M. J., and Peck, R. R., "Test-data generation using genetic algorithms," *Software Testing, Verification and Reliability*, vol. 9, pp. 263-282, 1999.
[261] Park, S. K. and Miller, K. W., "Random number generators: good ones are hard to find," *Commun. ACM*, vol. 31, pp. 1192-1201, 1988.
[262] Pasquini, A., Crespo, A. N., and Matrella, P., "Sensitivity of reliability-growth models to operational profile errors vs. testing accuracy," *IEEE Transactions on Reliability*, vol. 29, pp. 531-540, 1996.
[263] Perkinson, W. B., "A methodology for designing and executing ISDN feature tests, using automated test systems," in *Proceedings of IEEE Global Telecommunications Conference, 1992 (GLOBECOM '92)*. Orlando, FL, USA: IEEE, 1992.
[264] Peterson, I., *Fatal defect: chasing killer computer bugs*. New York: Vintage Books, 1996.
[265] Piwowarski, P., Ohba, M., and Caruso, J., "Coverage measurement experience during function test," in *Proceedings of the 15th international conference on Software Engineering*. Baltimore, Maryland, United States: IEEE Computer Society Press, 1993.
[266] Pohlheim, Conrad, and Griep, "Evolutionary safety testing of embedded control software by automatically generating compact test data sequences," in *SAE World Congress 2005*: SAE, 2005.
[267] Press, W. H., Teukolsky, A. A., Vetterling, W. T., and Flannery, B. P., *Numerical recipes in C : the art of scientific computing*. Cambridge: Cambridge University Press, 1992.
[268] Ramamoorthy, C. V., Ho, S. F., and Chen, W. T., "On the automated generation of program test data," *IEEE Trans. Softw. Eng.*, vol. 2, pp. 293-300, 1976.
[269] Reid, S. C., "An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing," in *Proceedings of the 4th International Symposium on Software Metrics*: IEEE Computer Society, 1997.
[270] Reid, S. C., "Module testing techniques - which are the most effective? Results of a retrospective analysis," in *Proc. Eurostar97: 5th European Conf. on Software Testing*, 1997.
[271] Richardson, J. A. and Kuester, J. L., "Algorithm 454: the complex method for constrained optimization [E4]," *Commun. ACM*, vol. 16, pp. 487-489, 1973.
[272] Roper, M., "Computer aided software testing using genetic algorithms," in *Tenth Int'l Quality Week Conf.* San Francisco, 1997.
[273] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C., "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," in *Proceedings of the International Conference on Software Maintenance*: IEEE Computer Society, 1998.
[274] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J., "Test Case Prioritization: An Empirical Study," in *Proceedings of the IEEE International Conference on Software Maintenance*: IEEE Computer Society, 1999.
[275] Rothermel, G., Untch, R. J., and Chu, C., "Prioritizing Test Cases For Regression Testing," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 929-948, 2001.

[276] Saikkonen, R., "Linux I/O port programming mini-HOWTO v3.0," vol. 2007, 2000.

[277] Schroeder, P. J., Bolaki, P., and Gopu, V., "Comparing the Fault Detection Effectiveness of N-way and Random Test Suites," presented at ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering, 2004.

[278] Schultz, A. C., Grefenstette, J. J., and De Jong, K. A., "Test and Evaluation by Genetic Algorithms," *IEEE Expert: Intelligent Systems and Their Applications*, vol. 8, pp. 9-14, 1993.

[279] Schultz, A. C., Grefenstette, J. J., and De Jong, K. A., "Applying genetic algorithms to the testing of intelligent controllers," in *Proceedings of Applying Machine Learning in Practice (IMLC-95)*, 1995.

[280] Sherwood, G., "Effective Testing of Factor Combinations," presented at Third Int'l Conf. Software Testing, Analysis and Review, Washington, DC, 1994.

[281] Sherwood, G. B., "Improving test case selection with constrained arrays," AT&T Report Number e.g. SCE-04-15, 1990.

[282] Sherwood, G. B., "Improving test case selection with constrained arrays II," AT&T Report Number e.g. SCE-04-15, 1990.

[283] Sherwood, G. B., "Tutorial – Constraints Example," vol. 2006: Testcover.com 2005.

[284] Shiba, T., Tsuchiya, T., and Kikuno, T., "Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing," presented at Proc. 28th Int'l Computer Software and Applicatios Conf (COMPSAC'04), 2004.

[285] Shooman, M. L., "Avionics Software Problem Occurrence Rates," presented at Proc.7th International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, 1996.

[286] Smith, B., Feather, M. S., and Muscettola, N., "Challenges and Methods in Testing the Remote Agent Planner," presented at Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO., 2000.

[287] Smith, B., Millar, W., Tung, Y. W., Nayak, P., Gamble, E., and Clark, M., "Validation and Verification of the Remote Agent for Spacecraft Autonomy," in *Proceedings of the 1999 IEEE Aerospace Conference*, 1999.

[288] Sneed, H. M., "Data Coverage Measurement in Program Testing," in *Proc. of IEEE Workshop on Software Testing*. Banff, Canada: IEEE, 1986.

[289] Stacy, W. and MacMillan, J., "Cognitive bias in software engineering," *Commun. ACM*, vol. 38, pp. 57-63, 1995.

[290] Stardom, J., "Metaheuristics and the search for covering and packing arrays," in *Dept. Mathematics*, vol. MSc: Simson Fraser University, 2001.

[291] Stevens, B. and Mendelsohn, E., "Efficient software testing protocols," in *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*. Toronto, Ontario, Canada: IBM Press, 1998.

[292] Tai, K. C. and Lie, Y., "A Test Generation Strategy for Pairwise Testing," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 109-111, 2002.

[293] Teasley, B., Leventhal, L. M., and Rohlman, D. S., "Positive Test Bias in Software Testing By Professionals: What's Right and What's Wrong," presented at Emperical Studies of Programmers: Fifth Workshop, Palo Alto, CA, 1993.

[294] Thevenod-Fosse, P., "From random testing of hardware to statistical testing of software," in *Proceedings of 5th Annual European Computer Conference (CompEuro'91) 'Advanced Computer Technology, Reliable Systems and Applications'*. Montreal, Que., Canada, 1991.

[295] Thevenod-Fosse, P. and Waeselynck, H., "STATEMATE applied to statistical software testing," *SIGSOFT Softw. Eng. Notes*, vol. 18, pp. 99-109, 1993.

[296] Thevenod-Fosse, P., Waeselynck, H., and Crouzet, Y., "An experimental study on software structural testing: deterministic versus random input generation," in *Digest of Papers Twenty-First International Symposium on Fault-Tolerant Computing (FTCS-21)*. Boston, Massachusetts, USA, 1991.

[297] Tichy, W. F., "Should Computer Scientists Experiment More?," *Computer*, vol. 31, pp. 32-40, 1998.

[298] Tichy, W. F., Lukowicz, P., Prechelt, L., and Heinz, E. A., "Experimental evaluation in computer science: a quantitative study," *J. Syst. Softw.*, vol. 28, pp. 9-18, 1995.

[299] Tracey, N., Clark, J., and Mander, K., "Automated program flaw finding using simulated annealing," in *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. Clearwater Beach, Florida, United States: ACM Press, 1998.

[300] Tracey, N., Clark, J., and Mander, K., "The way forward for unifying dynamic test-case generation: The optimisation-based approach," in *International Workshop on Dependable Computing and Its Applications (DCIA)*, 1998.

[301] Tracey, N., Clark, J., Mander, K., and McDermid, J., "An Automated Framework for Structural Test-Data Generation," in *Proceedings of the 13th IEEE international conference on Automated software engineering*: IEEE Computer Society, 1998.

[302] Tracey, N., Clark, J., Mander, K., and McDermid, J., "Automated test-data generation for exception conditions," *Softw. Pract. Exper.*, vol. 30, pp. 61-79, 2000.

[303] Tracey, N., Clark, J., McDermid, J., and Mander, K., "A search-based automated test-data generation framework for safety-critical systems," in *Systems engineering for business process change: new directions*: Springer-Verlag New York, Inc., 2002, pp. 174-213.

[304] Tsoukalas, M. Z., Duran, J. W., and Ntafos, S. C., "On some reliability estimation problems in random and partition testing," *IEEE Trans. Softw. Eng.*, vol. 19, pp. 687-697, 1993.

[305] Tung, Y.-W. and Aldiwan, W. S., "Automating test case generation for the new generation mission software system," in *Aerospace Conference Proceedings*. Big Sky, MT, USA: IEEE, 2000.

[306] Untch, R. H., Offutt, A. J., and Harrold, M. J., "Mutation analysis using mutant schemata," presented at Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '93), Cambridge, Massachusetts, 1993.

[307] Valentine, R., "Cruise Control," in *Automotive Electronics Handbook, Second Edition*, R.K., J., Ed. New York: McGraw-Hill, 1999.

[308] Vinter, J., Hannius, O., Norlander, T., Peter Folkesson, P., and Karlsson, J., "Experimental Dependability Evaluation of a Fail-Bounded Jet Engine Control

System for Unmanned Aerial Vehicles," presented at International Conference on Dependable Systems and Networks (DSN'05), 2005.

[309] Voas, J. M. and McGraw, G., *Software Fault Injection: Inoculating programs Against Errors*. New York: John Wiley & Sons, 1998.

[310] Voges, U., Gmeiner, L., and von Mayrhauser, A., "SADAT - An Automated Testing Tool," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 286-290, 1980.

[311] von Mayrhauser, A., Bai, A., Chen, T., Anderson, C., and Hajjar, A., "Fast Antirandom (FAR) Test Generation," in *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*: IEEE Computer Society, 1998.

[312] Vouk, M., McAllister, D. F., and Tai, K. C., "An experimental evaluation of the effectiveness of random testing of fault- tolerant software," in *Association for Computing Machinery and IEEE, Workshop on Software Testing*. Banff, Canada: ACM Press, 1986.

[313] Wallace, D. R. and Kuhn, D. R., "Failure Modes in medical device software: an analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering*, vol. 8, pp. 351-371, 2001.

[314] Wallace, M. G., Novello, S., and Schimpf, J., "ECLiPSe : A Platform for Constraint Logic Programming," *ICL Systems Journal*, vol. 12, 1997.

[315] Watkins, A., Berndt, D., Aebischer, K., Fisher, J., and Johnson, L., "Breeding Software Test Cases for Complex Systems," in *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9*: IEEE Computer Society, 2004.

[316] Watkins, A. and Hufnagel, E. M., "Evolutionary test data generation: a comparison of fitness functions," *Softw. Pract. Exper.*, vol. 36, pp. 95-116, 2005.

[317] Watkins, A. L., "The automatic generation of test data using genetic algorithms," in *Proc 4th Software Quality Conf.* Dundee, UK, 1995.

[318] Wegener, J., Baresel, A., and Sthamer, H., "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, pp. 841-854, 2001.

[319] Wegener, J. and Bühler, O., "Evaluation of Different Fitness Functions for the Evolutionary Testing of an Autonomous Parking System," in *Genetic and Evolutionary Computation – GECCO 2004: Genetic and Evolutionary Computation Conference*: Springer, 2004.

[320] Wegener, J. and Grochtmann, M., "Testing temporal correctness of real-time systems by means of genetic algorithms," in *Proceedings of the 10th International Software Quality Week*, 1997.

[321] Wegener, J., Sthamer, H., Jones, B. F., and Eyres, D. E., "Testing real-time systems using genetic algorithms," *Software Quality Control*, vol. 6, pp. 127-135, 1997.

[322] Weyuker, E., Goradia, T., and Singh, A., "Automatically Generating Test Data from a Boolean Specification," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 353-363, 1994.

[323] Weyuker, E. J., "On testing non-testable programs," *The Computer Journal*, vol. 25, pp. 465-470, 1982.

[324] Weyuker, E. J., "The evaluation of program-based software test data adequacy criteria," *Commun. ACM*, vol. 31, pp. 668-675, 1988.

[325] Weyuker, E. J. and Jeng, B., "Analyzing Partition Testing Strategies," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 703-711, 1991.

[326] White, L. J. and Cohen, E. I., "A domain strategy for computer program testing," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 247-257, 1980.

[327] White, L. J. and Wiszniewski, B., "Complexity of testing iterated borders for structural programs," in *Proc. of the 2nd IEEE Workshop on Software Testing, Verification and Analysis*. Banff, Canada: IEEE, 1988.

[328] White, L. J. and Wiszniewski, B., "Path testing of computer programs with loops using a tool for simple loop patterns," *Softw. Pract. Exper.*, vol. 21, pp. 1075-1102, 1991.

[329] Wichmann, B. A. and Hill, I. D., "Generating Good Pseudo-Random Numbers," *Computational Statistics & Data Analysis*, vol. 51, pp. 1614-1622, 2006.

[330] Williams, A. W., "Determination of Test Configurations for Pair-Wise Interaction Coverage," in *Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques*: Kluwer, B.V., 2000.

[331] Williams, A. W. and Probert, R. L., "A practical strategy for testing pair-wise coverage of network interfaces," in *Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*: IEEE Computer Society, 1996.

[332] Williams, A. W. and Probert, R. L., "A Measure for Component Interaction Test Coverage," in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*: IEEE Computer Society, 2001.

[333] Williams, A. W. and Probert, R. L., "Formulation of the Interaction Test Coverage Problem as an Integer Program," in *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*: Kluwer, B.V., 2002.

[334] Wong, E. E., Horgan, J. R., London, S., and Mather, A. P., "Effect of test set size and block coverage on the fault detection effectiveness," in *Proceedings., 5th International Symposium on Software Reliability Engineering*. Monterey, CA, USA: IEEE, 1994.

[335] Wong, W. E., Horgan, J. R., London, S., and Mathur, A. P., "Effect of Test Set Minimization on Fault Detection Effectiveness," *Software Practice and Experience*, vol. 28, pp. 347-369, 1998.

[336] Woodward, M. R., Hedley, D., and Hennel, M. A., "Experience with Path Analysis and Testing of Programs," *IEEE Trans. Softw. Eng.*, vol. 6, pp. 278-28, 1980.

[337] Wu, S. H., Malaiya, Y. K., and Jayasumana, A. P., "Antirandom vs. pseudorandom testing," in *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD '98)*. Austin, TX, USA, 1998.

[338] Xie, T., Marinov, D., Schulte, W., and Notkin, D., "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution," in *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*. Edinburgh, UK: Springer, 2005.

[339] Xu, B., Xu, L., Nie, C., Chu, W., and Chang, C. H., "Applying combinatorial method to test browser compatibility," in *Proceedings. Fifth International Symposium on Multimedia Software Engineering*. Boston, Massachusetts, USA: IEEE, 2003.

[340] Yan, J. and Zhang, J., "Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing," presented at Proc. 30th Annual Int'l Computer Software and Applications Conf. (COMPSAC'06), 2006.

[341] Yilmaz, C., Cohen, M. B., and Porter, A., "Covering arrays for efficient fault characterization in complex configuration spaces," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. Boston, Massachusetts, USA: ACM Press, 2004.

[342] Yin, H., "Antirandom Test Patterns Generation Tool," Computer Science Department, Colorado State University CS-98-101, 1996.

[343] Yin, H., Lebne-Dengel, Z., and Malaiya, Y. K., "Automatic Test Generation using Checkpoint Encoding and Antirandom Testing," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*: IEEE Computer Society, 1997.

[344] Yu-Wen, T. and Aldiwan, W. S., "Automating test case generation for the new generation mission software system," presented at Proc. IEEE Aerospace Conf., 2000, 2000.

[345] Zhan, Y. and Clark, J. A., "Search-Based Mutation Testing for Simulink Models," presented at Proc. of the 2005 Conference on Genetic and Evolutionary Computation, Washington DC, USA, 2005.

9. Appendix A – The Csaw Mutation Tool

9.1 Introduction

This appendix provides additional detail on the mutation tool, Csaw, that was written to support the work reported in this thesis. Section 8.2 describes the mutations that are provided by the tool. Section 8.3 compares the mutation operators provided by the Csaw tool with those commonly used for FORTRAN and with an idealised set of mutation operators for the C programming language.

9.2 Tool Capabilities

The capabilities of the Csaw tool are described in the following sections.

9.2.1 Operator Mutations

The tool can swap one operator for another e.g. '+' for '*', and so on. It supports this for arithmetic and logical operators, variable types etc. The substitutions that can be applied are defined in tables and hence can be easily altered. Some operators are not substituted e.g. '.' and '->' are not currently defined in the tables because mutants involving these operators often result in code that cannot be compiled. Thus, the mutants are by definition dead.

9.2.2 Variable Substitution

The tool will swap one variable name for another variable name e.g. if the variables `i`, `j` and `k` are defined in a function then all instances of `i` will be swapped with `j` and `k`, all instances of `j` with `i` and `k`, and so on. Variable substitution is done on scalar and array types independently but other uses for names such as names of structures and members within a structure are not distinguished.

9.2.3 Constant Substitution

All textual constants (e.g. from #define) are swapped for all other such defined constants that the tool finds in the text of the function being mutated. The tool considers a constant any text string that is not recognised as either a keyword or a variable.

9.2.4 Decimal Constants

The tool will offset decimal constant values by plus or minus one. For example, a constant of "10" will be converted to "both "9" and "11". Floating point, double precision and hexadecimal constant mutations have not currently been implemented. However, "holes" have been left in the tool for them.

9.2.5 Array Index Mutation

Array indexes that use variables are mutated by appending either "+1" or "-1", so `array[i]` is converted to `array[i-1]` and `array[i+1]` to produce off by one errors. Note that variable substitution also affects array indexing.

9.2.6 Statement Removal

Each statement that ends in a '`;`' is deleted. However for this to work correctly a statement must be on a single line because the Csaw tool operates on one line at a time.

9.2.7 Type Mutations

Unlike any other system the author knows of, this tool will mutate the type specifier of a variable. That is, it will swap "`unsigned int`" for "`int`", or "`double`" for "`float`" etc. This capability has been introduced because the primary target for the tool in this work is integer based real-time embedded code. Because of the limited amount of memory that this type of system often has, it is common to use integers of the smallest possible size e.g. using a `char` or `unsigned char` as an integer variable that only takes on a small number of values. Like operator mutations, type mutations are defined in an extendable table. As far as the author is aware, no other mutation tool has this capability.

9.3 Comparisons

9.3.1 FORTRAN Operators

The original set of FORTRAN mutation operators was developed by King and Offutt [193] for the Mothra mutation system for FORTRAN. The set of mutation operators for this system is given in Table 31. Operators fall into three classes as follows; Replacement of Operand (RO) modifiers, Expression Modifiers (EM), and Statement Modifiers.

As one of the first mutation systems introduced, the operators have been extensively studied and their properties better documented than other systems. For example, empirical work has been performed to determine which operators are most effective in the sense that test cases that kill mutants based on a given operator are also observed to kill mutants based on other operators. The largest such study by Offutt *et al.* [252] identified five operators that were deemed "necessary" as follows: absolute value insertion (ABS); arithmetic operator replacement (AOR); logical connector replacement (LCR); relational operator replacement (ROR); and unary operator insertion (UOI). Of these AOR, LCR and ROR are implemented for operators and UOI is partly implemented.

Work has also been done on the theoretical properties of mutation operators compared with other common criteria for measuring the effectiveness of a test set such as statement and decision coverage [238]. Offutt *et al.* [253] have examined the operators that are necessary to achieve various levels of coverage and found that they matched the necessary operators as shown in Table 31.

Table 31. Summary of mutation operators for the FORTRAN programming language.

Operator		Class	Needed	Csaw	Subsumes (2)
AAR	array reference for array reference replacement	RO		yes	
ABS	absolute value insertion	EM	yes	no	
ACR	array reference for constant replacement	RO		no	
AOR	arithmetic operator replacement	EM	yes	yes	
ASR	array reference for scalar variable replacement	RO		no	
CAR	constant for array reference replacement	RO		no	
CNR	comparable array name replacement	RO		yes	
CRP	constant replacement	RO		yes	
CSR	constant for scalar variable replacement	RO		no	
DER	DO statement end replacement	SM		no	
DSA	DATA statement alterations	SM		n/a	
GLR	GOTO label replacement	SM		no	
LCR	logical connector replacement	EM	yes	yes	decision coverage decision/condition
ROR	relational operator replacement	EM	yes	yes	condition coverage decision/condition
RSR	RETURN statement replacement	SM		no	
SAN	statement analysis (replacement by TRAP)	SM		no	statement coverage
SAR	scalar variable for array reference replacement	RO		no	
SCR	scalar for constant replacement	RO		no	
SDL	statement deletion	SM		yes	all defs
SRC	source constant replacement	RO		yes	
SVR	scalar variable replacement	OR		yes	
UOI	unary operator insertion	EM	yes	no	

(1) Replaces each condition in each decision with TRUE or FALSE

(2) Subsumes is on weak mutation not strong mutation

In addition to the coverage criteria defined by Myers, Offutt *et al.* [253] also examined the all-defs requirement as defined by Frankl and Weyuker [126] and concluded that this was achieved by the SDL operator. However, it is interesting to note that the all-uses and all def-uses cases were not similarly examined, which leaves one to assume that these are not subsumed by the mutation.

A comparison of the operators provided by Csaw with the FORTRAN necessary operators shows a few weaknesses. The absolute value insertion (ABS) operator, for example, is not implemented at all in Csaw. Given the way in which Csaw operates this is not a trivial exercise because Csaw does not parse the code, it cannot distinguish an lvalue from an rvalue.

The arithmetic operator replacement (AOR), however, is probably implemented completely in that Csaw does attempt to replace each operator with the set of all other operators that are valid at the same point. Similarly, the logical connector replacement (LCR) replaces the operators with a full set of other valid operators. However, the correspondence with the Mothra operators is not exact. Offutt *et al.* [253] state that "the logical connector mutation operator (LCR), among other modifications, replaces each decision in a program by TRUE and FALSE". Again, Csaw does not do this as it does not know what constitutes a decision.

The relational operator replacement (ROR) is also fully implemented, given the caveat that it may not completely duplicate the FORTRAN operators as wholesale replacement of clauses, as in the case of the LCR operator, is again not possible.

The statement analysis (SAN) operator has not been implemented but this could be implemented using the same mechanism as the statement deletion operator (SDL), which is fully implemented within the bounds of what Csaw can do.

The unary operator insertion (UOI) is likewise not implemented although experiments have shown that doing so may be feasible a part of the variable name replacement code. Other missing operators are discussed more completely below.

9.3.2 Ideal C Mutation Operators

Agrawal *et al.* [5] produced a technical report on mutant operators for the C language. A summary of those operators and how they compare with the operators that Csaw uses is given in Table 32.

Table 32. Summary of C mutation operators and comparison with Csaw mutation tool. Notes on equivalent Mothra mutation operates are included in the usage column.

Area	Operator	Usage	Csaw
Statement	STRP	trap on statement execution, replaces each statement with code to cause a termination	no
	STRI	trap on if condition, replaces branch predicate with code to cause termination	no
	SSDL	statement deletion	implemented
	SRSR	`return` statement replacement	no
	SGLR	`goto` label replacement	no
	SCRB	`continue` replacement by `break`	no
	SBRC	`break` replacement by `continue`	no
	SBRn	`break` to n^{th} enclosing level	no
	SCRn	`continue` to n^{th} enclosing level	no
	SWDD	while replaced by do-while	no
	SDWD	do-while replaced by while	no
	SMTT	multiple trip trap, used to ensure that a loop is executed more than once.	no
	SMTC	multiple trip continue, ensure that if a loop executes n iteration on the n+1 it will not execute the body.	no
	SSOM	sequence operator mutation, used to modify effect of the comma operator	no
	SMVB	move closing brace up or down one line	no
	SSWM	switch statement mutation, cause execution to halt if a case is selected	no
Operators	Obom	binary operator mutation (OAAN, Mothra AOR) (OBBN, Mothra LCR) (ORRN, Mothra ROR)	implemented
	OUOR	unary operator mutation	implemented
	OLNG (UOI)	logical negation	indirectly
	OCNG	logical context negation	partly
	OBNG	bitwise negation	indirectly
	OIPM	indirect operator precedence mutation	no

Area	Operator	Usage	Csaw
	OCOR	cast operator replacement	indirectly
Variables	Varr	mutate array references in expressions	implemented
	Vprr	mutate pointer references in expressions	partial
	Vsrr	scalar variable reference replacement	implemented
	Vtrr	mutate structure references	partial
	VASM	mutate subscripts in array references (multi dimensional arrays)	no
	VSCR	mutate components of structure	indirectly
	VDTR	variable domain traps, TRAP on negative, zero and positive (Mothra ABS)	no
	VTWD	twiddle mutations, mirror off by one errors e.g. +/- 1 (Mothra UOI)	partial
Constants	CRCR	required constant replacement	no
	CCCR	constant for constant replacement	partial
	CCSR	constant for scalar replacement	no

The utility of some of the suggested operators is lower than for others. For example, the SRSR operator (return statement replacement) is most effective if there are multiple return statements as variable replacement operators will generally modify a statement of the form `return (xyz)`. Likewise, the `goto` replacement label operator SGLR is only useful if the `goto` is actually used. In the code used for this study, neither of these conditions were met. That is, the functions used had a single `return` and the `goto` operator was not used.

Some operators are impossible for the Csaw tool set to implement. For example, the SMVB operator normally operates over multiple lines so Csaw, which operates one line at a time, cannot deal with this operator.

As shown in Table 32, some rules are "indirectly" implanted. This means that although the exact mutation mechanism as suggested in [5] is not used, Csaw achieves a similar effect via brute force. However, it should be noted that Csaw's version might not have exactly the same properties. For example, the Varr operator is type aware, i.e., it does not substitute integer arrays for pointer arrays but Csaw will happily make this type of substitution but because the compiler will catch many of these instances the net effect will be almost the same. Another example is structure component replacement; Csaw achieves

nearly the same end as the ideal operator by using every possible variable it knows about. The vast majority will not compile but this will still effectively sift them out and allows the legal ones to pass though.

9.3.3 The Adequacy of Csaw

Given that the Csaw implementation of ideal mutation operators is incomplete, what effect will this have on the results of the work presented?

Some operators will have no or minimal effect because the constructs that they mutate are not presenting in the subject code. The best two examples of this are the mutation operators for return statements and the goto operator. All the code examined in the work here has a single return at the end of a function. Likewise none of the code examined uses the goto operator so the absence of the operator will have no effect.

The absence of the FORTRAN statement analysis operator (SAN and STRP C operator) is unfortunate but again should have minimal effects because the subject industrial code is known to have test sets that achieve full statement and branch coverage. Therefore, the effect of this operators absence should not unduly affect the results because we are performing a direct comparison between the effectiveness of two different test sets. If one set is known *a priori* to meet a criterion then a test set that is less effective should show obvious differences in the mutation kill rate.

The same argument applies to the other operators that have been omitted or only partly implemented. Thus, although Csaw may be considered to be flawed because it does not implement all operators and does not ensure that coverage criteria such as statement and branch coverage are meet, it is adequate for the purposes of this study.

In addition, having an incomplete mutation system does not seem to preclude it being useful in practice. The Jumble mutation system for Java (Irvine *et al.* [178]) for example excludes a large number of the possible operations from within some of the operator classes[59] to achieve a significant speed increase over what would otherwise be possible. This was been done so that regular builds of complete systems and the associated unit testing of mutants can be performed.

---

[59] This point was made during presentations at the Mutation 2007 workshop.