Information and Software Technology 1995 **37** (1) 5–13

Towards a benchmark for the evaluation of software testing techniques

James Miller, Marc Roper, Murray Wood and Andrew Brooks

Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, UK

Despite the existence of a great number of software testing techniques we are largely ignorant of their respective powers as software engineering methods. It is argued that more experimental work in software testing is necessary in order to place testing techniques onto a scale of measurement, or classify them in such a way that is useful to the software engineer. Current experimental practices are examined using a parametric framework and are shown to contribute little towards a cohesive and useful body of knowledge. The idea of a benchmark repository of faulty and correct software is explored enabling unification of diverse experimental results. Such a unification should start the process of moving towards an evaluation taxonomy of testing methods.

Keywords: testing, software quality, experimentation, benchmark repository

Introduction

Given two testing methods, *A*, and *B*, how much 'better' is one than the other?

To anyone outside the field of testing, this might seem to be a reasonable request, but it is guaranteed to strike a chord of uncertainty within those involved in it.

Sometimes we are able to create a partial ordering of testing techniques, stating that *A* will achieve a higher level of coverage than *B* (see Ntafos[1] and Rapps and Weyuker[2] for example). It should be noted that such partial orderings are unable to say anything about how much more coverage is achieved by one technique over another, and frequently techniques are placed at the same level because we are unable to distinguish between their relative coverage abilities. Sometimes we are even unable to position *A* and *B* in a partial ordering—that is, we can say nothing about the relative test coverage merits of the technique. Whilst such partial orderings are useful, it must be stressed that they represent theoretical comparisons of the degree of thoroughness with which a testing technique may test a program. They do not necessarily relate to the 'ability' of the technique to find faults within code and consequently are of little use to the practising software engineer.

Even placing two testing techniques within a partial ordering still leaves the tester with a nagging doubt. We *know* what can go wrong—in practically any testing technique there is a wide range of variability. In many techniques there is variability in the analysis of the program under test (for example in branch testing, given two *if-then-else* statements in sequence there are four ways of choosing the minimal number of branches through the sequence) and there is variability in the process of selecting data to test the analysed program (that is, given that the order in which the branches are to be tested has been chosen, there is likely to be a large number of possible sets of test data that will meet this criterion)*.

The consequence of this is that when the techniques are applied in practice a 'strong' technique (one which tends to generate data that will achieve thorough coverage of the program—data flow testing, or LCSAJs for example) may fail to find a fault revealed by a much 'weaker' technique (one which achieves comparatively scant coverage of the code under test—such as statement testing). This point is raised and illustrated by Hamlet[3]. Consider the following example:

```
function multiplier(n,m:integer) : integer;
var result:integer;
begin
  result := 0;
```

*Possible exceptions to this are techniques like weak mutation testing of constant errors in arithmetic relations where the 'off-by-a-constant' error term can be fixed.

SSDI 0950-5849(94)00456-0

```
    while n > 0 do
    begin
      result := result + n;
      n := n - 1
    end;
    multiplier := result
  end;
```

The person applying branch testing to the above function might choose values of (5,3) for n and m to exercise the true and false outcomes of the branch, and even supplement it with the values (0,2) to test the loop zero times, and be content with the results. On the other hand, another person carrying out branch testing, or even applying the weaker technique of statement testing might choose the input (1,2) and, receiving an unexpected result realize that the line:

```
result := result + n;
```

should in fact have been coded as

```
result := result + m;
```

Admittedly this is a trivial example, but the scope for this kind of frustrating experience increases with the size of application under test. In addition to this there are other forms of variability which may occur in experimentation.

If research in software testing is going to have a major impact on the industrial practice of software engineering then it needs to generate some results which the software engineer can use. We have to produce more than tentative statements about partial orderings qualified with disclaimers. Returning to our original question, given two testing methods *A* and *B* we want to know how much 'better' one is than the other. The software engineer needs to know how much effort is involved in carrying out *A* and *B*, how many faults they are likely to catch, of what kind and so on. In other words, the full resource implications of choosing a particular testing technique should be known. (Indeed, it could be argued that testing should do this if only to be classified within software engineering.) Of course, we still need the caveats, but the software engineer is concerned about the average performance of a technique applied on a large scale.

Current experimental practices

This section does not represent a full survey of experimentation in software testing (the benefits of such an activity are questionable, especially as the activities of experimental design, execution, and analysis of results are of variable quality) but instead gives a flavour of current practices. For the interested reader, DeMillo *et al.*[4] provide a short description and a number of references pertaining to experiments performed on individual testing techniques. The papers chosen represent what we regard as a cross-section of the current experimental practice in the area. They span a period of about 10 years and, given the amount of experimental work carried out in this area, are by no means a small sample size. The aim here is not to level criticism at the respective authors but to try to present a cohesive view of experimental evaluation.

We have labelled the experiments after their authors and the papers they are drawn from are as follows: Basili and Selby[5]; Lauterbach and Randell[6]; Howden[7]; Girgis and Woodward[8]; Lesniak-Betley[9]; Solheim and Rowland[10]; Frankl and Weiss[11]; and Foreman and Zweben[12].

For the purposes of comparison we have looked at the parameters of the experiments and these are presented in Tables 1 and 2. The presence of a '?' in Table 1 indicates that the information was unknown or not directly available from the reference.

The presentation of this information in a table is necessarily and intentionally terse and there are some other points which need to be made which embellish some of the experiments. The results of Basili and Selby are fairly cautiously and equivocally stated. In the three-phase study it was found that structural testing was 13.2% less efficient than the other techniques; in the second phase nothing could be concluded, and in the third phase it was found that reading (not strictly a testing technique, admittedly, but nonetheless a commonly applied technique with its own problems of variability) was 16% more efficient than the other techniques and that functional testing was 11.2% more efficient than structural testing. This study also contained detailed subject data (relating to experience, etc.) and applied more statistical techniques, thereby allowing the expression of the results in terms of significance levels.

The study conducted by Solheim and Rowland is interesting as it involves artificially generated and executed software. These are large and complex systems but made slightly simpler in terms of the computations (which are effectively meaningless). The experiments were then carried out on various versions of generated systems. Solheim and Rowland state that they believe that integration testing is effected more by the large-scale structure of the software than the computations taking place in the individual modules. They freely admit that the models have not been validated yet and warn against extrapolating their results to apply to real systems. Their results found that top-down integration testing was most effective in terms of defect correction and exercised the greatest number of modules. In contrast, bottom-up integration testing was least effective and exercised relatively few modules. Top-down and big-bang integration testing produced the most reliable systems (and bottom-up the least reliable). There was no preference for strategies which incorporated spot unit testing, or for those which did not. Furthermore, there was no preference for either phased (one 'level' at a time) or incremental (one module at a time) approaches.

The experiment carried out by Frankl and Weiss was trying to investigate not only if one technique was more effective than another, but also if it is more effective for test sets of the same size. In addition to this they explored the relationship between partial coverage and effectiveness (in other words, if a program has had 50% of its branches covered, then what is the probability that an error will be exposed). It is also worth noting that they did not use the testing strategies to generate the data, but generated random sets of data from various profiles until they met the adequacy criteria (i.e. 100% edge or variable use coverage). They suggest that this approach is more likely to be carried out in practice. Their results showed that all-uses was more

Table 1. Comparison of experimental evaluations

	Basili and Selby	Lauterbach and Randell	Howden	Girgis and Woodward	Lesniak-Betley
Subjects	Students and Professionals	Professionals	?	?	Professionals
No. of Subjects	74	4	?	?	?
Software Type	Text Formatter, Math Plot, Numeric Data, Database maintenance	Military, Scenario Generator	DP, PL/1 update, String Edit (*2), Sort, Stats.	Maths?	?
Language	Simpl-T	?	COBOL, PL/1, Algol, PL/360	Fortran 77	?
Size (LOC)	55, 95, 48, 144	45, 29, 32, 79, 328, 376, 680, 292	450, 175, 25, 21, 13, 28	Small 50?	569
Fault Origin	Natural and seeded	Natural remaining	Natural(5) and seeded	Seeded by tool	Natural
No. Faults	9, 6, 7, 12 (34)	4, 3, 1, 14, 10, 21, 14, 18 (85)	3, 20, 1, 2, 1, 1 (28)	80	38
Fault Classification	Omission/Commission* (Init., Comp., Control, I/F, Data, Cosmetic)	?	?	Computation, Domain	?
Testing Techniques	Reading, Structural (Statement), Functional	Branch, Random, Functional, Review, Error Anomaly, Structure Analysis	Path, Branch, Functional, Symbolic, and Requirements, Design and Anomaly Analysis	Weak Mutation, Dataflow (c, p, all) Control (Nodes, Edges, LCSAJs, DDs)	Statement
Results (% of faults found unless otherwise stated)	Three phases - see discussion in text.	Branch: 25 Random: 19 Func.: 23 Review: 36 EA: 29 SA: 3	Path: 64 Branch: 21 Func.: 36 Symbolic: 61 Req. A.: 25 Design A.: 7 Anomaly A.: 14	Weak M.: 41 Dataflow: 70 Control: 85	Statement 95

Table 2. Comparison of experimental evaluations

	Solheim and Rowland	Frankl and Weiss	Foreman and Zweben
Subjects	None (Randomly generated test cases)	None (Randomly generated test cases)	None (Impact determined by authors)
No. of Subjects	0	0	0
Software Type	Automatically generated and executed, but with a variety of module types	Array manipulation, matrix manipulation, text formatting and string matching	Typesetting (T_EX)
Language	Ada	Pascal	WEB/Pascal
Size (LOC)	24 systems of over 1000 modules, and 15 systems of 1364 modules	9 programs ranging from 22 to 78 LOC	24 000 (approx)
Fault Origin	Randomly seeded	Natural remaining	Natural remaining
No. Faults	Varies with system, (approx in range 150–500)	1 per program	32
Fault Classification	By module type (control, interface, computation, timing deadlock, resource interrupt, re-entrant swapping and scope)	None	None (Knuth—T_EX's author—created one but this was not used)
Testing Techniques	Top Down, Bottom Up Big Bang and Modified Sandwich Integration (some done in phases or incrementally and with or without Spot Unit Testing)	Randomly generated from profiles to meet All-Edges (Branch), All-Uses and Null (i.e. any test data) criteria	Control flow (Branch and Statement), Data flow (6 levels), plus other supplementary techniques
Results (% of faults found unless otherwise stated)	Top Down generally most effective—for details see discussion in text	All-Uses generally most effective—for details see discussion in text	Statement 34, Branch 34, All p-uses 34, All-uses 41, All d-u-paths 50 (see discussion in text)

effective than all-edges for five of the nine programs, and more effective than the null criterion for six of the nine programs. All edges was more effective than the null criterion for five of the nine programs. In addition, all-uses appeared to guarantee detection of the error in four programs. In comparing test sets of a similar size, they found all-uses to be more effective than all-edges for four of the nine programs, more effective than null for four of the nine programs. For *none* of the programs was all-edges more effective than the null criterion. Their paper then goes on to discuss the effectiveness of partial achievement of criteria, but this is beyond the scope of this paper.

The study carried out by Foreman and Zweben did not actually generate any test data but instead determined the

path of execution capable of revealing the fault. They then analysed the test strategies to determine which were *guaranteed* to find the fault (this is more stringent than other approaches which consider a fault to be revealed if it is exposed by any set of test data conforming to some strategy). When doing the analysis they considered the control and data flow techniques from the weakest to the strongest—if the weakest technique would not reveal the fault then they moved on to a stronger technique and so on. If the control and data flow techniques failed then they moved on to the other techniques (such as boundary testing and static data flow analysis). From the family of data flow testing techniques they also considered all definitions (which found 22% of faults). This combined with all p-uses achieves branch coverage (and consequently is capable of revealing exactly the same set of faults as branch coverage).

All experiments were laboratory based except those conducted by Lauterbach and Randell and Lesniak-Betley which were executed in the 'real world'. Lauterbach and Randell also placed an emphasis on timing and stopping rules; it also contains the partial results for the 'CSC' (computer software component) level which have not been included in the table.

Bearing in mind the complex nature of comparisons, what are we able to conclude from Table 1? The answer is, disappointingly little. There are contradictions in the results (see the figures for branch testing and functional testing for Lauterbach and Randell and Howden for example). Even if there were no contradictions, would we be able to use the results to place the techniques on the ordinal scale? Unfortunately not! The disparity in almost every parameter (subjects, numbers, software type, language, testing techniques, etc.) coupled with the lack of information (it is not always clear as to what constitutes 'functional' testing, for example) and variable experimental practices means that no sensible conclusion could be drawn from the studies examined. The diversity of studies fails to build into a cohesive body of research and ends up only being exploratory and issue-raising.

There is also much experimental activity of a more modest nature. Many developers of testing techniques include in their papers a short evaluation section. Although practice varies, this frequently takes the form of comparing their new technique against established or rival techniques by seeding a small program with a few faults and running the program with data generated according to the respective testing technique. Such an activity might provide the developer with an 'initial justification' for their technique, and the inquisitive researcher with an inkling of its fault-finding ability, but it does little to contribute towards the production of better software.

How do we progress in experimentation?

We need to encourage good experimentation to be repeated by researchers throughout the world. Given that the work summarized in this paper represents much of the experimentation carried out to date (and from which we can conclude little), what can be done to increase the amount of successful and productive experimentation? The main problem is variability—of subjects, languages, software type, technique application and data selection—to name but a few sources. This, and other sources of problems in experimental evaluations of testing techniques are raised by Hamlet[13]. The authors have also explored other possible solutions to the problems of experimentation[14] and it is one of these possible solutions that is explored in this section.

As a way of reducing the impact of variability the testing 'community' should agree on building a large suite of software to allow the benchmarking of testing strategies. (This idea is briefly discussed by Osterweil and Clarke[15].) Rather than attempting to describe and artificially manufacture benchmark software to model the various aspects of variability, it is suggested that the collection of a large number of 'real world' programs would effectively model the majority of these cases without the need to produce artificial taxonomic descriptions. Such taxonomic descriptions of the above-mentioned sources of variability are certainly currently unavailable and are themselves the source of a great deal of research within the testing community (for example there exists no agreed classification of fault types despite nearly 20 years of extensive research[16]). The 'performance' of a testing technique against this software would then define its position in an ordering or supply a profile of its cost and performance on a range of software and fault types. The problems of benchmark creation and how it would be used are discussed in the following sections.

Creation of a common benchmark repository

This problem is analogous to Computer Architecture performance or Database and Transaction processing evaluation by benchmarking. Such a repository could be extended over a number of problem domains to provide domain specific results as well as general results and finally generalized into an evaluation taxonomy comparing techniques against characteristics of the software, faults, etc. Unfortunately, creation of a complete and consistent software benchmark is in itself an extremely difficult task. The computer architecture community have been investigating this issue since Curnow and Wichmann's Whetstone benchmark program in 1976[17]. In this period relatively little progress has been made, the field has been plagued with bogus claims, poor definitions and a variety of different measures all claiming to measure performance. For example, Hennessy and Patterson[18] in their book make the following statement:

> Cost/performance fallacies and pitfalls have ensnared many computer architects including ourselves. For this reason, more space is devoted to the warning section in this chapter [Performance and Cost] than in other chapters in this book.

Also organizations currently producing benchmark figures/comparisons tend to make rather bland statements often accompanied with caveats, leaving the reader to draw his or her own conclusions[19].

Recently the community has realized that a more concerted effort is required if the above deficiencies are to be

addressed. The year 1988 saw the formation of the System Performance Evaluation Cooperative (SPEC). SPEC contains many of the leading computer hardware manufactures who have now agreed on a set of programs and inputs that all architectures will be subjected to when providing benchmarking data[20]. Also newly formed is the Perfect Club: this is a collection of universities and companies who attempt to extend existing benchmarking suites by writing new programs in new languages, often utilizing new algorithms to further test current and future architectures[21]. Similarly, in the domain of databases and transaction processing, the Transaction Processing Performance Council (TPC) has been formed with the express aim of defining benchmarks[22].

Although the work in these communities still has some way to go, the authors strongly believe that it shows the way forward to the testing community. The construction and maintenance of a testing benchmark suite is too large and too important an undertaking to be left to a single institution.

Perhaps the Computer Architecture community's main achievement has been to clearly indicate that producing a set of 'meaningful' benchmarks is a highly hazardous undertaking. Many attempts have failed due to the deficiencies described below. As we shall see, these deficiencies have parallels in any testing benchmark.

(1) **Assumptions and approximations.**
Because attempting to calculate performance accurately is a very expensive and difficult process, many practitioners have sought to shorten the evaluation exercise. This involves evaluating a 'pseudo-performance' measure which is easier to calculate. Unfortunately this new measurement is always deficient because of the simplifications and hence it is difficult to generalize or aggregate the results[18]. Similarly, in creating a testing benchmark the donated code is going to be simplified in terms of, for example, its size and its interaction with its environment.

(2) **Effects of different evaluation platforms.**
Often the other components in the system add distorting effects, such as varying amounts of error detection of different compilers and operating systems.

(3) **What exactly are we measuring?**
Are we interested in average behaviour, worst case behaviour or some combination of the two?

(4) **Do we need multiple performance measures?**
Currently the Computer Architecture field is favouring multiple measures but it is still a question of debate. For example, many computer architecture benchmarks of scientific programs on vector machines report results in MFLOPS (millions of floating point operations per second). Berry *et al.*[23] argue that this metric is inappropriate in many of the above circumstances and calls for further investigation into alternative metrics. Extending this notion into the field of testing, we ask if we can capture the performance of a testing strategy by a single measure (such as number of tests generated, reliability of the consequent code, etc.)?

(5) **Differences between theory and reality/ implementation.**
Hennessy and Patterson[18] cite the example where a computer architecture benchmark might be assumed to run twice as fast on a dual processor machine as on a single processor machine. This of course is not the case. In testing terms, as has been pointed out, a major source of variability is in the testing methods themselves. Another way of progressing is to constrain the application of the method so that the production of test data becomes more or less deterministic and repeatable. For example, in branch testing we could require that for each predicate one set of test data falls within a certain tolerance of the boundary defined by the predicate (if possible). If we are unable to do this for a testing method, then the extreme course of action might be to dismiss it as a testing method and categorize it as an *ad hoc* technique (the performance of which cannot be predicted).

(6) **Can synthetic benchmarks predict performance?**
Most benchmark systems are made up of small benchmarks that attempt to exercise one or more specific aspects. Is it realistic to expect any benchmark to cover all facets of a particular feature? The testing analogue assumes that the use of seeded faults can mimic the full range of complexity and stubbornness exhibited by natural faults. Also, do we run the risk of testing strategies evolving to achieve high performance on the synthetic benchmarks, while potentially sacrificing performance in 'real systems'? Furthermore, synthetic benchmarks are expensive to produce, requiring an exact specification. For example, TPC Benchmark A (for database systems) runs to 40 pages of definitions without discussing implementation details[24].

This again causes many researchers to seek simplifications in producing such benchmarks. For example, within the testing community it is possible to envisage T_EX becoming a standard (due to the publications of fault listing, etc.[25], and the fact that it has already been used in some empirical studies[12,26]) but can a single program really capture the myriad of different debugging problems? Also such a system can lead to erroneous results, because all the details of T_EX are widely known, testing methodologies can 'adapt' to suit these specific problems rather than the general case. This may also be the case with other candidate *de facto* benchmarks such as the set of well-known small programs used by Frankl and Weiss[11], a subset of which was used by Mathur and Wong[27].

In the ideal situation all evaluation of testing strategies should be 'blind' (i.e. all testing attempted on completely unknown problems); unfortunately this is extremely expensive and difficult to organize (all problems becoming one use only and all evaluation having to take place concurrently). A more attainable proposition is to have a large number of differing programs to alleviate most possibilities to 'adaption' or specific details of the relevant program spreading to new experimenters.

(7) **Should we exercise censorship in which programs are contributed to the benchmark?**
For example, the SPEC use the following criteria for accepting candidate computer architecture benchmarks[28]:
- Public domain applications are desired.
- The benchmark should run for a minimum of one minute on an eight Dhrystone 1.1 MIP machine.
- The same source code should run without change on all member machines.
- Open debate of strengths and weaknesses of benchmarks.
- Clear documented reasons for rejecting benchmarks.
- Results must be correct and reproducible.

Clearly despite having many excellent features such a list cannot be considered definitive, complete or without debate. Indeed it is difficult to see how any list could ever be constructed to define an absolute set of terms for benchmark selection.

Transferring this analogy to the testing domain, any testing benchmark might be subjected to similar constraints being applied to the size, complexity and other aspects of the software.

Building a large-scale benchmark repository

The ideal repository of software would be a hand-crafted selection which represented the different sources of variability—i.e. it would contain software written in a variety of languages and paradigms, applied to different domains and containing the gamut of fault types and distributions. In the short term, however, achieving such a suite of programs would be difficult to say the least. Apart from the problems described earlier, relating to the lack of agreed fault taxonomy, there would inevitably be conflict about languages, machine environments and problem domains. On top of all that, who would write the software?

The authors believe that the easiest way to start alleviating the above problems is to create a very large repository of 'real-world' software. This, in fact, could be relatively easily gathered. Many projects today are produced under version control packages, such as SCCS[29], RCS[30] or Adele[31]. If programmers can be convinced to update the current version every time a fault is removed, simply by recording the before and after versions, we have the basis of an entry for the repository. This would have to be combined with a functional specification (the importance of which is emphasized by Foreman and Zweben[12]) defining how the software is run and from which test data may potentially be chosen. Of course, these 'donations' will be drawn from different problem domains, be written in different languages, and have different faults.

This approach is directly supported by many of the packages (allocating a new version number every time a new edit is undertaken). What is required is to convince organizations and their staff of the benefits of participating in this project. Indeed, from the donor's point of view there is little in the way of effort involved. Only a small percentage of updates during the normal testing and debugging phase (or operation and maintenance) are required for such a repository to become a useful source of information. Hence with the assistance of people involved in various industrial projects, such a repository could rapidly grow into a major source of information on the performance of various testing techniques on a variety of software and fault types.

One fault or many? In many experimental evaluations of software, faults are seeded in the code one at a time. The argument for doing this is usually that it is the simplest way of telling if the fault has been found. If we observe the results of the test data applied to the faulty and corrected code then if there is a difference we can conclude that the test data has revealed the fault (if the results are the same then either the test data has not revealed the fault, or it has and the output has been 'masked' in some way by some other piece of code). If there are several faults in the code, then if the output between the correct and faulty code differs we can only conclude that *some* of the faults have been revealed—but not how many or which ones. This is of little benefit if we are trying to quantify the performance of a testing technique. The authors recognize that placing a restriction of only one fault on an update is both an unlikely and unrealistic situation, but if the results are to be checked automatically and the behaviour of a technique quantified, then it is a restriction which will have to apply.

Alternatively a project could provide a series of 'incrementally corrected snapshots' of the system. Now each of the faults could be explored (within the benchmark) concurrently, yielding a more realistic test (multiple faults).

With reference to Figure 1, the output from version 1.0 would be compared with that of 1.1 and if there is a difference then the fault is assumed to be found. Version 1.2 has two fewer faults when compared with 1.0 and 1 fault as compared with 1.1, so its output is compared with 1.1 and again if they are different then the fault is assumed to have been found. If the outputs are the same then the test data has not revealed the fault. In either case, the outputs of 1.2 and 1.3 are compared and so on... The problem with this scenario is that if the final system has evolved significantly from the initial system then the two outputs

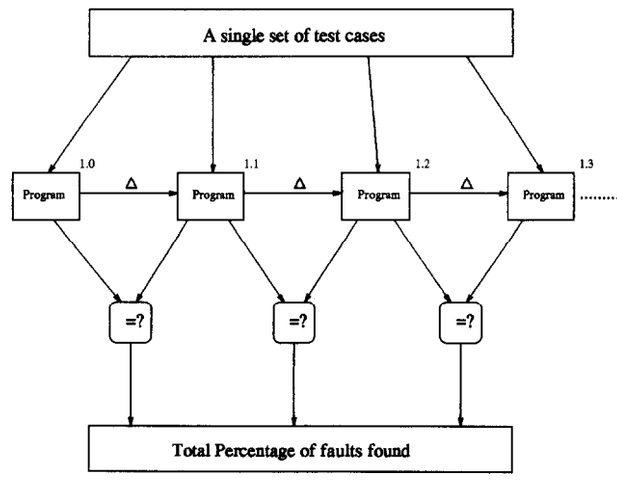

Figure 1 Concurrent evaluation of a single set of test cases against a series of incremental snapshots

may be too different to allow sensible selection of a single set of test cases. Additionally, it is imposing an ordering on which faults are found when.

A further problem is that there could be masking problems with faults disguising that other faults have been revealed and so a constraint has to be placed on the software to ensure that the faults are relatively independent. Of course, it is important and interesting to examine the behaviour of non-independent faults, but this puts a greater onus on the benchmark site to evaluate exactly which (as opposed to 'if any') faults have been revealed.

Does size matter? As in the previous section, we are wandering into the undesirable area of benchmark constraint. However, it must be remembered that people are going to have to test the donor code (see next section) and so the size of contribution has to be large enough to be non-trivial, but not so large as to deter any potential tester from actually trying to test it. Of course, this can cause a problem. It may be that certain kinds of fault only tend to appear in larger pieces of code, or become exposed by strategies which apply at a higher level of abstraction. If this is the case, then we run the risk of only finding certain classes of fault. Whilst the application of detailed unit testing techniques to large pieces of software is highly time consuming, such pieces of software should be held in the repository and used for experimentation with 'higher-level' techniques such as those associated with integration testing.

How is it used?

Experimentation with testing techniques would take place worldwide at any time. An experimenter would voluntarily engage in the process (illustrated in Figure 2) outlined below:

(1) An experimenter would need to obtain the faulty version of the program and the functional specification and to derive test cases for this program using any testing technique desired. There is no restriction on which programs are chosen, or which testing method is used. Furthermore, there is no restriction on the way in which data are generated—the experimenter might choose to do it by reading the code and generating the data by hand, or use the specification to generate the data and execute an instrumented version of the program to determine when some test adequacy criteria had been met.

(2) These test cases would then be returned to the repository, along with an indication of how they were derived.

(3) The repository would then run the test cases through both the faulty and corrected versions and compare their outputs, any difference in outputs signifying the technique had revealed the fault. It is necessary for the group looking after the repository to run the tests to ensure fair play and help to preserve the anonymity of the faults.

(4) This information (the technique used, the data generated, the cost, etc.) would then be noted by the repository and stored in a database to form the basis of subsequent analysis.

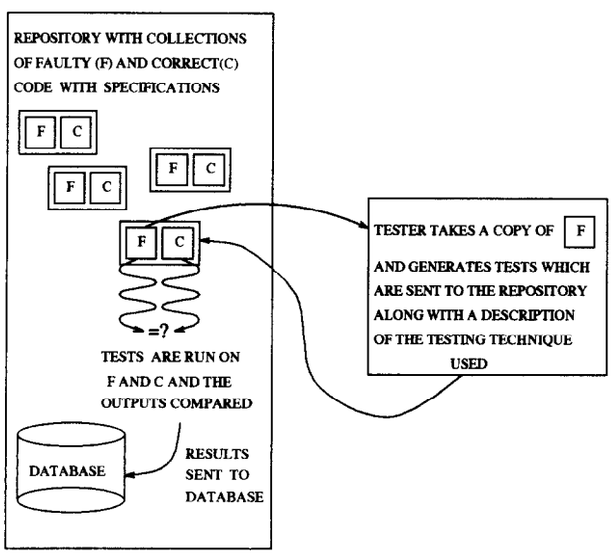

Figure 2 Illustration of the benchmark in use

It should not be assumed that an individual experimenter is expected (or even desired) to attempt the total amount of benchmarking for an individual technique. Instead it is hoped that many researchers within the testing community would participate for a relatively short period on a regular basis. It is only by distributing the effort that such projects will ever become feasible and provide results which are generalizable.

Obviously the repository site must take care in the construction of the benchmark. Indeed the amount of time and resources required to construct and maintain even such a semi-autonomous benchmarking repository demands that it be a community-wide exercise rather than overburdening a single institution.

The above comments do not imply the need for a single centralized repository site. Indeed the repository could be distributed worldwide with institutions making whatever contributions they could afford in terms of time and resources. This would help to spread the load of construction and maintenance. Experimenters would (in general) interact with the repository site via normal remote access procedures ftp, etc.) and hence by having the repository distributed the probability of overloading a single site is greatly reduced. Experimenters have alternatives if their initial site of interactive experiences equipment malfunction.

Results

The results from such a benchmarking study will take time to evolve. As the numbers of people trying out the techniques on different pieces of software increase, a picture will begin to emerge showing how effective particular techniques are in relation to others, what types of faults they tend to find, and what kinds they tend to miss. Furthermore, the effects of languages will become clear, as will a picture of the resource requirements in terms of numbers of test cases per technique. The results will not be a simple 'A is better than B' (although it will be possible to draw such a conclusion) but a profile of the effectiveness

of a technique used in different hands on a variety of software types.

Towards a taxonomy for experimental evaluation of testing techniques

To compare experiments, the experimenters need to have looked at the same things. It would be desirable to take each experiment that has been done and classify it according to a parametric framework, similar to that in Table 1. This would show which experiments were evaluating which techniques and encourage the build-up of a cohesive body of knowledge. It would also give future experimenters an idea of what has been done and which parameters should be considered and help to promote the all-important activity of repeating experiments. No such classification can be done sensibly at the moment since the studies are so diverse they would each occupy a separate component in the framework. This procedure is standard practice in a great number of mature scientific and technology disciplines, such as physics, chemistry, biology, aeronautics, automotive and power generation, where it is more readily recognized that the identification and classification of key ideas and concepts are essential tasks for progress[32].

The end goal of such a schema should be to produce an evaluation taxonomy. Such a taxonomy would contain statements about each testing technique performance against various characteristics. These characteristics would be found during the empirical build-up of knowledge, rather than hypothesized without physical evidence. The process of producing a taxonomy by an empirical approach has been well established in other fields since the 1960s[33–36], but seems unknown within the computer science community. With such a taxonomy in place users can feel confident about selecting a testing method to meet their situation. Also such a taxonomy can be a starting point for resolving other ambiguities within the field such as nomenclature[37].

Conclusions

If we are to place the discipline of software testing on a scientific or engineering platform then not only do we need to carry out large numbers of experiments, but these need to be carried out in a way that is comparable and which builds up into a cohesive body of knowledge. Experimenters cannot work in isolation, but need to consider other experiments, and *repeat* other experiments. This is undoubtedly a massive undertaking which without careful organization and structuring is unlikely to succeed. It is argued (in this article) that the most appropriate framework for conducting successful experimentation (upon testing techniques) is by establishing a benchmark repository. By constructing such a repository not only would experimentation and evaluation be encouraged, but also more importantly the various parameters (sources of variability) of the experiment would be defined before the experiment and consistent with other experiments using the repository. The repository would allow straightforward collection of experimental data and the data would be sufficiently well defined as to allow comparison and accumulation.

Despite being a massive undertaking, the corresponding benefits of producing such information are staggering. How much time is wasted on the application of inappropriate testing techniques? How much software is released every year with large numbers of faults due to insufficient/inappropriate testing? Without such quantitative evaluation information, it is difficult to see how these questions of productivity and quality can be answered.

References

1 Ntafos, S C 'A comparison of some structural testing strategies' *IEEE Trans. on Soft. Eng.* Vol 14 No 6 (June 1988) pp 868–874
2 Rapps, S and Weyuker, E J 'Selecting software test data using data flow information' *IEEE Trans. on Soft. Eng.* SE-11 No 4 (April 1985) pp 367–375
3 Hamlet, D C 'Are we testing for true reliability?' *IEEE Software* Vol 9 No 4 (1992) pp 21–27
4 DeMillo, R A, McCracken, W M, Martin, R J and Passafiume, J F *Software testing and evaluation* Benjamin Cummings (1987)
5 Basili, V R and Selby, R W 'Comparing the effectiveness of software testing strategies' *IEEE Trans. on Soft. Eng.* SE-13 Vol 12 (December 1987) pp 1278–1296
6 Lauterbach, L and Randell, W 'Experimental evaluation of six test techniques' *Proc. Compass 89* ACM Press (1989) pp 36–41
7 Howden, W E 'An evaluation of the effectiveness of symbolic testing' *Software—Practice and Experience* Vol 8 (1987) pp 381–397
8 Girgis, M R and Woodward, M R 'An experimental comparison of the error exposing ability of program testing criteria' *Proc. Workshop on Software Testing* IEEE (July 1986) pp 64–73
9 Lesniak-Betley, A M 'Audit of statement analysis methodology' *Proc. 3rd Ann. Int. Phoenix Conf. on Computers and Communications* (March 1984) pp 174–180
10 Solheim, J A and Rowland, J H 'An empirical study of testing and integration strategies using artificial software systems' *IEEE Trans. on Soft. Eng.* Vol 19 No 10 (1993) pp 941–949
11 Frankl, P G and Weiss, S N 'An experimental comparison of the effectiveness of branch testing and data flow testing' *IEEE Trans. on Soft. Eng.* Vol 19 No 8 (1993) pp 774–787
12 Foreman, L M and Zweben, S H 'A study of the effectiveness of control and data flow testing strategies' *J. Systems and Software* Vol 21 (1993) pp 215–228
13 Hamlet, R 'Special section on software testing (guest editorial)' *Comm. ACM* Vol 31 No 6 (1988) pp 662–667
14 Roper, M, Miller, J, Brooks, A and Wood, M 'Towards the experimental evaluation of software testing techniques' *Technical Report EFOCS-2-93*, Dept. Computer Science, University of Strathclyde, Glasgow, UK (July 1993)
15 Osterweil, L and Clarke, L A 'A proposed testing and analysis research initiative' *IEEE Software* Vol 9 No 5 (1992) pp 89–96
16 Roper, M *Software testing* McGraw-Hill (1994)
17 Curnow, H J and Wichmann, B A 'A synthetic benchmark' *The Computer Journal* Vol 19 No 1 (1976)
18 Hennessy, J L and Patterson, D A *Computer architecture, a quantitative approach* Morgan Kaufmann (1990)
19 MIPS Computer Systems, Inc. *Performance brief CPU benchmark Issue 3.5* (1988)
20 SPEC *SPEC benchmark suite release 1.0* (1989)
21 Berry, M D, Chen, D, Koss, P and Kuck, D 'The Perfect Club benchmarks: effective performance evaluation of supercomputers' *CSRD 827*, University of Illinois, Center for Supercomputing Research (1988)
22 Transaction Processing Performance Council (TPC) *TPC benchmark, a standard* ITOM International Co (1989)
23 Berry, M, Cybenko, G and Larson, J 'Scientific benchmark characterizations' *Parallel Computing* Vol 17 Nos 10–11 (1991) pp 1173–1194
24 Transaction Processing Performance Control (TPC) 'TPC benchmark A' in Gray, J (ed) *The benchmark handbook* (1991)
25 Knuth, D E 'The errors of TeX' *Software—Practice and Experience* Vol 19 No 7 (1989) pp 607–686
26 Horgan, J R and Mathur, A P 'Assessing testing tools in research and education' *IEEE Software* Vol 9 No 3 (1992) pp 61–69
27 Mathur, A P and Wong, W E 'An empirical comparison of data flow and mutuation-based test data adequacy criteria' *Software Testing, Verification and Reliability* Vol 4 No 1 (1994) pp 9–31

28 Dixit, K M 'The SPEC benchmarks' *Parallel Computing* Vol 17 Nos 10–11 (1991) pp 1195–1209
29 Rochkind, M J 'The source control system' *IEEE Trans. on Soft. Eng.* Vol 1 No 4 (1975) pp 364–370
30 Tichy, W F 'RCS—a system for version control' *Software Practice and Experience* Vol 15 (1985) pp 637–654
31 Belkhatir, N and Estublier, J 'Experiences with a database of programs' *ACM SIGPLAN Notices* Vol 22 No 1 (1987) pp 84–91
32 Gamlin, L and Vines, G *The evolution of life* Oxford University Press (1987)
33 Jardine, N and Sibson, R *Mathematical taxonomy* Wiley (1971)
34 Ehrlich, P R and Holm, R W 'Patterns and populations' *Science* Vol 137 (1962) pp 652–657
35 Davis, P H and Heywood, V H *Principles of angiosperm taxonomy* Oliver and Boyd (1963)
36 Sokal, R R and Camin, J H 'The two taxonomies: areas of agreement and conflict' *Systematic Zoology* Vol 14 (1965) pp 176–195
37 Mayr, E *Principles of systematic zoology* McGraw-Hill (1969)