Efficient Software Verification: Statistical Testing Using Automated Search

Simon Poulding and John A. Clark

Abstract—Statistical testing has been shown to be more efficient at detecting faults in software than other methods of dynamic testing such as random and structural testing. Test data are generated by sampling from a probability distribution chosen so that each element of the software's structure is exercised with a high probability. However, deriving a suitable distribution is difficult for all but the simplest of programs. This paper demonstrates that automated search is a practical method of finding near-optimal probability distributions for real-world programs, and that test sets generated from these distributions continue to show superior efficiency in detecting faults in the software.

Index Terms—Software/program verification, testing strategies, test coverage of code, optimization.

1 Introduction

Statistical testing generates test data by sampling from a probability distribution defined over the software's input domain [1], [2], [3]. The distribution is chosen carefully so that it satisfies an adequacy criterion based on the testing objective, typically expressed in terms of functional or structural properties of the software. This paper considers structural statistical testing, which uses criteria based on the program's control flow graph. The criteria ensure that each structural element—such as a statement, branch, or path—is exercised as frequently as possible by inputs sampled from the distribution, on the assumption that this will improve the fault-detecting efficiency.

Thévenod-Fosse and Waeselynck demonstrated that test sets generated by statistical testing can be more efficient at detecting faults than test sets produced by uniform random and deterministic structural testing [1], [4]. However, the derivation of effective probability distributions can be challenging for software that is larger and more complex than "toy" examples. In [4], probability distributions for programs of real-world size were derived manually, using data returned by executing instrumented code, but it was necessary to use a relatively weak adequacy criterion for this approach to be feasible.

Search-Based Software Engineering (SBSE) reformulates software engineering tasks as optimization problems and uses efficient modern algorithms, such as metaheuristic search and operational research techniques, to find solutions [5], [6]. This has proven an effective method of solving a range of software engineering tasks, with applications including: requirements engineering [7], project planning [8], software task allocation [9], code refactoring [10], protocol synthesis [11], search-based test data generation [12], and the design of resource-constrained algorithms [13]. As the performance and affordability of computing resources continues to improve, SBSE becomes a practical method of solving increasingly complex software engineering problems that are often intractable by other means.

This paper proposes using a search-based approach to derive probability distributions for statistical testing. We believe this to be a novel application of SBSE. It is inspired by the manual process described by Thévenod-Fosse and Waeselynck, but uses automated search to enable a more scalable technique without the need to compromise on the strength of the adequacy criterion.

One of the current themes of SBSE research is practicality for the software engineer—the end-user of these applications—particularly when SBSE techniques are applied to problems of realistic scale [14]. Therefore, we show that not only is it *feasible* to use automated search to derive effective probability distributions, but also that it is *practical* to do so when testing software components of real-world size. In addition, we use mutation analysis to provide empirical evidence that probability distributions derived using automated search continue to demonstrate the superior efficiency of statistical testing compared to both uniform random and deterministic structural testing. We show that adding an explicit "diversity" objective to the search can be beneficial in maintaining this efficiency.

The paper is organized as follows: The next two sections place this paper in the context of related work on statistical testing and search-based test data generation. Section 4 defines the hypotheses tested by the experimental work. Section 5 describes the implementation of automated search used to test the hypotheses, and Section 6 discusses the experimental design. The results are analyzed in Section 7. Conclusions are followed by a discussion of future work in Section 8.

2 Statistical Testing

This section describes related work on statistical testing and compares the technique to two other widely used dynamic testing techniques: random testing and structural

• *The authors are with the Department of Computer Science, University of York, Heslington, York YO10 5DD, UK.*
E-mail: {smp, jac}@cs.york.ac.uk.

Manuscript received 1 Sept. 2008; revised 2 Mar. 2009; accepted 3 June 2009; published online 27 Jan. 2010.
Recommended for acceptance by M. Harman and A. Mansouri.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2008-09-0275.
Digital Object Identifier no. 10.1109/TSE.2010.24.

```
1    /* 1 <= a <= 50, 1 <= b <= 20 */
2    int simpleFunc(int a, int b) {
3       int r;
4       if (a<=5) {
5          if (b>=18)
6             r = abs(b-19);
7          else
8             r = b;
9       } else {
10         if (b<=3)
11            r = abs(b-2);
12         else
13            r = 10+b;
14      }
15      return r;
16   }
```

Fig. 1. The implementation of *simpleFunc*.

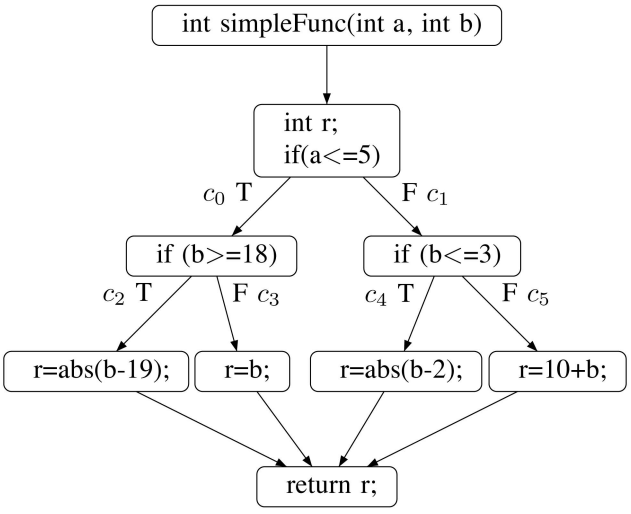

Fig. 2. Control flow graph of *simpleFunc*.

testing. A simple function is introduced that serves as an example throughout this paper.

2.1 Random Testing

Random testing generates test data by random sampling from the input domain according to a probability distribution [15]. No information is required as to the internal structure of the software under test (SUT), and in this situation, the choice of distribution is arbitrary. Using the operational profile—the distribution of input vectors that would occur when the system is in use—is a common choice which has the advantage of producing a reliability estimate in addition to detecting faults. However, it is difficult to derive the profile for a lower level component from the operational profile of the system as a whole, and the operational profile may not be available at the required stage of the development life cycle [16]. For these reasons, a uniform distribution is often used for random testing, and we make this assumption in this paper.

One advantage of random testing is that any vector in the input domain could be used as a test input and this enables the detection of faults that can be overlooked by a more systematic derivation of test cases (see, e.g., [16]). However, large test sets may be required for effective fault detection, and this can make random testing cost-inefficient despite the relative ease of generating test data.

Consider, for example, the function *simpleFunc*, as shown in Fig. 1. If we assume that the function arguments are the only inputs tested, then the input domain is given as follows:

$$D = \{(a, b) \in \mathbb{Z}^2 : 1 \leq a \leq 50, 1 \leq b \leq 20\}. \quad (1)$$

For random testing, the argument a would be treated independently from b and the value of a would be sampled from a uniform distribution over the interval $[1, 50]$. Under such a distribution, the false branch of the `if` statement at line 4 will be taken nine times more frequently than the true branch. This imbalance means that an unnecessarily large test size would be required for at least one input vector, on average, to execute—and therefore potentially detect faults in—statements in the true branch. Such a test set would only be efficient if the false branch were nine times more likely to have faults in it than the true branch.

In the absence of information about the distribution of faults and the structure of the software, it is impossible to know the optimal distribution for random testing. However, there is no reason why an arbitrary choice, such as the operational profile or a uniform distribution, need be close to optimal, and this accounts for the potential inefficiency of random testing in detecting faults.

2.2 Structural Testing

In contrast to random testing, structural testing uses information about the SUT implementation to select input vectors for test cases. Zhu et al. provide a comprehensive survey of structural testing in [17]. A set of coverage elements, C, is defined based on the program's structure. *Branch coverage* defines the elements to be the edges in the control flow graph of the implemented SUT, corresponding to branches arising from control statements. For *path coverage*, the elements are paths through the flow graph, corresponding to execution paths through the SUT. A number of other coverage criteria have been proposed, including those that combine control flow information with knowledge of when data variables are defined and used by the program.

For each coverage element $c \in C$, there is a subdomain, S_c, of the input domain consisting of vectors that exercise the element. As an example, consider again *simpleFunc*. Its control flow graph is shown in Fig. 2. The edge labeled c_2 corresponds to the predicate (`b >= 18`) at line 5 of the function evaluating to true. To exercise this branch also requires the predicate (`a <= 5`) at line 4 to be true, so the subdomain for this branch coverage element is:

$$S_{c_2} = \{(a, b) \in \mathbb{Z}^2 : 1 \leq a \leq 5, 18 \leq b \leq 20\}. \quad (2)$$

An adequacy criterion is defined for test sets used for structural testing specifying that they must exercise each coverage element at least once. In other words, the test set must contain at least one vector from each subdomain S_c.

If it is assumed that satisfying this adequacy criterion is sufficient for test sets to effectively detect faults, then efficiency can be improved by finding *small* test sets that satisfy the criterion. One approach is to consider each coverage element, c, in turn and find an input vector in its subdomain S_c. An alternative is to initially create a large test

set that satisfies the adequacy criterion—for example, by combining existing test sets—and then derive a smaller subset that still meets the criterion [18], [19].

However, this assumption does not necessarily hold: Test sets that satisfy the adequacy criteria need not be effective at detecting faults. It is possible for a set of input vectors that exercises all of the coverage elements to nevertheless contain none of the input vectors that would detect a particular fault. For example, the following set satisfies the adequacy criterion for branch coverage of *simpleFunc*, as well as the stronger path coverage criterion:

$$T = \{(9, 2), (4, 6), (5, 19), (7, 18)\}. \tag{3}$$

Assume that line 6 of the function is incorrectly implemented as `r = b-19;`, i.e., omitting the `abs()` function call. Any input vector in the following subset of the input domain would detect the presence of this fault:

$$D_{f_1} = \{(a, b) \in \mathbb{Z}^2 : 1 \leq a \leq 5, b = 18\}. \tag{4}$$

Although T satisfies the adequacy criterion, it contains none of the fault-detecting input vectors in D_{f_1}.

Whether structural testing generates test sets that are efficient at detecting faults depends on the nature of the software and the adequacy criterion chosen. Theoretical and empirical evidence suggests that structural testing can be no better at detecting faults than random testing in many circumstances, and therefore often less efficient when the overall cost of testing is considered [15], [20]. Nevertheless, the use of structural testing is a mandatory part of many formal software certification processes [21], [22].

2.3 Statistical Testing

Statistical testing overcomes some of the shortcomings of random testing and structural testing by combining approaches from both of these techniques [1], [2], [4], [23], [24], [25]. Statistical testing may also be used for functional coverage [3], but here we consider statistical testing applied to coverage of the SUT's structure.

Statistical testing, like random testing, generates test sets by sampling input vectors according to a probability distribution. The difference is that the probability distribution must satisfy an adequacy criterion defined in terms of the structure of the SUT, in a similar way to the adequacy criteria used by structural testing. Structural testing defines an adequacy criterion for the test set itself, but the adequacy criterion for statistical testing is defined for the probability distribution from which test sets are generated.

As in structural testing, the adequacy criterion considers a set, C, of coverage elements. For a given probability distribution, Φ, over the input domain, a coverage element, $c \in C$, has a particular probability of being exercised by an input vector sampled from Φ. We denote the probability of c being exercised as p_c. We denote the lowest probability p_c across all elements of C as $\hat{p}$ and will refer to this as the *probability lower bound*. The adequacy criterion that Φ must satisfy is that $\hat{p}$ must be greater than or equal to a given target value: This ensures that there is a "good" chance that any particular coverage element c is exercised by a single-input vector sampled randomly from Φ. The target value for $\hat{p}$ will depend on the nature of the SUT and type of coverage elements chosen. We will refer to distributions that have the highest possible value of $\hat{p}$ for a given SUT and coverage type as *optimal*.

Statistical testing may be considered to be an extension to random testing where structural information is used to derive a probability distribution which is more "balanced" in terms of coverage of structural elements of the SUT. The intention is that such a distribution will be more efficient: A test set will detect more faults than a test set of the same size generated from a uniform distribution.

Alternatively, statistical testing may be viewed as a form of structural testing where random sampling from a probability distribution is the means used to generate multiple input vectors that exercise each coverage element. Multiple input vectors should detect more faults than a test set consisting of one input vector for each coverage element. We borrow the terminology used in [1] to describe the latter, more traditional form of structural testing as "deterministic."

As an example, consider the following probability distribution over the input domain of *simpleFunc*:

$$f(a, b) = \begin{cases} 2.94 \times 10^{-3}, & \text{if } 1 \leq a \leq 5, 1 \leq b \leq 17, \\ 1.67 \times 10^{-2}, & \text{if } 1 \leq a \leq 5, 18 \leq b \leq 20, \\ 1.85 \times 10^{-3}, & \text{if } 6 \leq a \leq 50, 1 \leq b \leq 3, \\ 3.27 \times 10^{-4}, & \text{if } 6 \leq a \leq 50, 4 \leq b \leq 20. \end{cases} \tag{5}$$

The distribution is constructed as follows: Each separate case in the equation refers to a subdomain that exercises one of the four, mutually exclusive, innermost branches of *simpleFunc*'s control flow graph, labeled c_2 to c_5 in Fig. 2. For example, subdomain $D_{c_3} = \{(a, b) : 1 \leq a \leq 5, 1 \leq b \leq 17\}$, the first case in (5), consists of the input vectors that exercise branch c_3. Probabilities are assigned to each *individual* vector in a subdomain in inverse proportion to the subdomain's cardinality in order that the probability of selecting *any* input vector from that domain is 0.25. Considering D_{c_3} again, this subdomain has a cardinality of $5 \times 17 = 85$, so the probability assigned to each vector in D_{c_3} is $0.25/85 \approx 2.94 \times 10^{-3}$.

This construction ensures that each of the branches c_2 to c_5 has the same probability, 0.25, of being exercised by a single-input vector sampled from the distribution defined by (5). It follows that the branches c_0 and c_1 each have a probability 0.5 of being exercised. As every branch has a probability of 0.25, or more, of being exercised by an input vector sampled from the distribution, the distribution has a probability lower bound, $\hat{p}$, of 0.25. (Since one, and only one, of the branches c_2 to c_5 is exercised by every possible input vector to *simpleFunc*, 0.25 is also the highest possible value for the lower bound when inducing branch coverage of *simpleFunc*, and so this distribution may be considered optimal.)

For many coverage criteria, an input vector may exercise more than one coverage element and the control structure and data dependencies of the SUT result in complicated dependencies between the coverage element probabilities. For this reason, the construction of a suitable probability distribution using analytical means, such as that used to derive the distribution of (5) for *simpleFunc*, is often infeasible for nontrivial programs.

In practice, a test engineer using statistical testing may select the size of test sets generated from the probability

distribution so that there is a high likelihood that all coverage elements are exercised at least m times by the test set, where m is a choice made by the engineer. The dependencies between coverage elements discussed above typically mean that there is not a simple relationship between test size, the probability of exercising all coverage elements at least m times, and the probability lower bound, $\hat{p}$. However, it is reasonable to expect that the higher the value of the lower bound $\hat{p}$, the smaller the test sets that are required to exercise each coverage element at least m times. If the purpose is to derive efficient test sets—ones that detect the most faults using the fewest test cases—this suggests that setting the probability lower bound as high as possible is a sensible objective.

Thévenod-Fosse and Waeselynck provide convincing empirical evidence of the superior efficacy of test sets generated using structural statistical testing, compared to test sets of the same size generated using uniform random testing and deterministic structural testing [1], [4]. The efficacy was measured by introducing mutation faults into the SUT and assessing how many faults are detected by test sets generated by each of the techniques. (We use a similar method to assess fault-detecting ability in the experimental work of this paper, and this is described in detail in Section 6.2.)

In these papers, Thévenod-Fosse and Waeselynck constructed probability distributions for the statistical testing in two different ways. For simple software, it was possible to derive an optimal distribution by a static analysis of the code [3], similar in technique to how the probability distribution of (5) was constructed for *simpleFunc* above. For more complicated software, a suitable distribution was found empirically as follows [4]: A number of input vectors were sampled from an initial probability distribution (such as a uniform distribution) and used to execute an instrumented version of the software in order to identify the coverage elements exercised. The input vectors that exercised the least-exercised coverage element were reviewed manually and used to intuitively "refine" the probability distribution with the intention of increasing the likelihood of exercising the least-exercised element (increasing the probability lower bound using the terminology introduced above). This trial-and-error process was repeated until the desired probability lower bound was achieved. However, to ensure that this manually intensive approach was feasible, a weak coverage criterion was used and the refinement halted once the coverage probabilities were "deemed sufficiently high."

The manual empirical search described by Thévenod-Fosse and Waeselynck provides the inspiration for the use of automated search proposed by this paper. A practical application of automated search to this problem would not only be more cost-effective, but would also facilitate the use of statistical testing on more complicated software components than would be possible otherwise.

3 Search-Based Test Data Generation

The use of search in generating test data is one of the most active areas of SBSE research [10]. McMinn [12] provides a detailed survey of techniques that have been described for both structural and functional testing, as well as for testing nonfunctional properties such as execution time.

For techniques that generate test data for structural testing, the objective of the search is typically an input vector that exercises a specific coverage element. As for other SBSE applications, two of the distinguishing features are the optimization method used and the associated fitness function. The fitness function assesses how close a candidate input vector is to exercising the desired coverage element, and this information is used to guide the optimization algorithm on an efficient trajectory to this objective.

The fitness is often calculated using the data returned by executing an instrumented version of the SUT with the candidate input vector. One measure of fitness, the *approach level*, compares the executed path through the SUT to a path that exercises the desired element. The more nodes the paths have in common, i.e., the closer the input vector comes to reaching the desired coverage element before following a divergent path, the better the fitness. Pargas et al. [26] use the approach level as the fitness and apply a genetic algorithm as the optimization method.

An alternative fitness measure considers the branch at which the executed path diverges from the desired path. The fitness quantifies how close the predicate for this branch came to returning the Boolean value that would have taken the desired branch. Korel [27] uses this fitness measure, the *branch distance*, and sequential search as the optimization method, while Tracey et al. [28] apply simulated annealing to a refined form of the branch distance metric. Later work by Tracey [29] and Wegener et al. [30] shows that a combination of both approach level and branch distance is the most effective.

In this paper, we take a similar approach by using information provided by an instrumented version of SUT to calculate the fitness. However, a key difference, and a novel aspect of this work, is that the objective of the search is not an input vector exercising a specific coverage element, but instead a probability distribution over the entire input domain that satisfies the adequacy criterion. For this reason, a different type of fitness function is required, as well as a method of representing probability distributions. We propose forms for both in Section 5.

4 Experimental Hypotheses

The objectives of the experimental work described in the remainder of this paper are:

- to demonstrate that automated search is a practical method of deriving probability distributions for statistical testing,
- to verify that test sets generated from these distributions maintain the superior efficiency over uniform random testing and deterministic structural testing that has been demonstrated in previous work on statistical testing.

These objectives are formalized as the following hypotheses:

Hypothesis 1. *Automated search is able to derive input probability distributions with near-optimal probability lower bounds for a range of software, including SUTs significantly*

more complex than "toy" problems. It is possible to find such solutions in a practical time, using computing power typical of a desktop PC.

Hypothesis 2. *Test sets generated by statistical testing using automated search detect more faults than sets of the same size derived using random testing with a uniform distribution* (uniform random testing).

Hypothesis 3. *Test sets generated by statistical testing using automated search that exercise each coverage element multiple times detect more faults than test sets typical of traditional structural testing that exercise each element once* (deterministic structural testing).

Hypothesis 4. *Test sets generated from probability distributions that demonstrate high lower bounds for the coverage element probabilities detect more faults than test sets of the same size generated from distributions with lesser lower bounds. In other words, it is beneficial to search for distributions with near-optimal lower bounds.*

Hypothesis 5. *Test sets generated from probability distributions found by automated search with a diversity constraint are more effective at detecting faults than those derived without such a constraint.*

The term *diversity* is used here to describe probability distributions that sample many different input vectors from the subdomains associated with each coverage element. For example, the following distribution for *simpleFunc* has the same optimal lower bound as that defined by (5):

$$f(a,b) = \begin{cases} 0.25, & \text{if } a = 3, b = 9, \\ 0.25, & \text{if } a = 3, b = 19, \\ 0.25, & \text{if } a = 28, b = 2, \\ 0.25, & \text{if } a = 28, b = 12, \\ 0, & \text{otherwise.} \end{cases} \quad (6)$$

However, a test set generated from this distribution would contain only four different input vectors, and it is reasonable to expect that this lack of diversity would lead to a reduced ability to detect faults in the SUT compared to the more diverse distribution defined by (5).

The notion of diversity is implicit in existing work on statistical testing, but Hypothesis 5 proposes that it is beneficial to make this an explicit objective when using automated search. The hypothesis was formulated as a result of experimental work on the first four hypotheses. Some distributions found using automated search that had near-optimal lower bounds nevertheless demonstrated poor fault-detecting ability. A manual inspection of the test cases generated by these distributions suggested a lack of diversity might be the cause.

5 Implementation

In order to test the experimental hypotheses, an automated search method was implemented. The following aspects of the implementation are described in this section:

- the representation of a probability distribution over the software's input domain,
- how the fitness of a candidate solution is evaluated,
- the search method used to locate the fittest solutions,
- brief details of the software engineering.

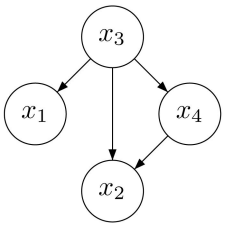

Fig. 3. Example BN for an SUT with four input arguments.

5.1 Representation

The components of the input vector are represented internally by real numbers from the unit interval. We denote the internal components as x_i, where $0 \leq x_i < 1$. This normalization enables the internal representation to be independent of the actual data types and domains of the input arguments. The mapping from the normalized value, x_i, to a realized input argument is straightforward for integer, floating-point, and Boolean data types. For example, if x_1 is the internal value of the argument a of *simpleFunc*, then a sensible mapping is:

$$a = 1 + \lfloor 50x_1 \rfloor. \quad (7)$$

We believe this mechanism can be extended to other data types by defining a mapping from the unit interval, $0 \leq x_i < 1$, to the domain of the data type. For nonnumeric and compound data types—such as strings, pointers, or object data types—the construction of a suitable mapping will be nontrivial and may need to incorporate context-specific information such as object constructor methods or valid pointer values. One possibility is to include the generation of suitable mappings as part of the search itself: The search method would locate both a probability distribution over the unit interval *and* a mapping from the unit interval to the domain of the data type. However, for the empirical work described in this paper, we consider only scalar numeric data types where the mapping can be predefined, as in the example above.

In general, there will be dependencies between the components of the input vector. The representation of the probability distribution must therefore be capable of describing these dependencies as a joint multivariate probability distribution. It would be insufficient to represent only the marginal distributions of each input vector component: This would enforce independence between the components and prevent the derivation of the most optimal probability distributions for statistical testing.

For this reason, a Bayesian network (BN) is used to describe the probability distribution. A BN consists of a directed acyclic graph in which each node represents a component of the input vector, and a directed edge indicates that the child node's probability distribution is conditionally dependent on the parents node's value.

Fig. 3 is an example of a BN describing a probability distribution over the input domain of an SUT with four arguments. The directed edges indicate that the distribution of argument x_1 is conditional on the value taken by its

parent, x_3. Similarly, x_4 is conditional on x_3, while x_2 is conditional on the values taken by both its parents, x_3 and x_4.

A BN typically describes a probability distribution over a discrete set of values. Since the internal component representations, x_i, take real—and therefore nondiscrete—values, the unit interval for each component is discretized by partitioning it into contiguous intervals or "bins." The number of bins and the bin boundaries are variable and are not the same for each node. As a result of this discretization, the nodes in the BN define conditional probability distributions in terms of the bins rather than the actual values.

As an example, consider again the function *simpleFunc*. The two arguments, a and b, are represented internally using real-valued components x_1 and x_2.

Assume the BN for a distribution indicates that the probability distribution of x_2 is conditional on x_1. Assume also that x_1 is discretized by two bins, say $[0, 0.12), [0.12, 1)$, and x_2 by three bins, $[0, 0.17), [0.17, 0.88), [0.88, 1)$. At node x_1, a single probability distribution defines the chances of selecting bin 1 or bin 2. However, at node x_2, two conditional probability distributions are defined: The first specifies the chance of selecting bin 1, 2, or 3 for x_2 if x_1 takes a value from bin 1 of node 1; the second specifies a different distribution that is used if x_1 takes a value from bin 2.

A random vector is sampled from the BN by traversing the graph from parent nodes to child nodes. At each child node, one bin is randomly selected according to the conditional probability distribution defined by the bins chosen at its parent nodes. To convert bin choices to component values, a value is sampled from the bin's interval according to a uniform probability distribution. (This is an arbitrary choice, and we suspect other distributions may be more effective.)

5.2 Fitness Evaluation

The fitness is a measure of how close a candidate probability distribution is to satisfying both a coverage constraint—the statistical testing adequacy criterion—and a diversity constraint. These constraints are expressed using target values: t_{cov} for the lower bound of the coverage element probabilities and t_{div} for the diversity.

To calculate the fitness, K input vectors are sampled from the probability distribution. An instrumented version of the SUT is executed using the sampled input vectors. The instrumentation identifies which coverage elements are exercised during each execution.

To assess the coverage, the indicator variable $e_{c;i}$ is set to 1 if the coverage element $c \in C$ is exercised one *or more* times when the SUT is executed using the ith input vector from the sample, and set to 0 otherwise. An estimate of the lower bound of the coverage element probabilities is then calculated as:

$$g_{\text{cov}} = \frac{1}{K} \min_{c \in C} \left(\sum_{i=1}^{K} e_{c;i} \right). \tag{8}$$

To assess the diversity, the same sample of input vectors is considered. For each coverage element c, h_c denotes the number of times that the most frequent input vector that exercises the element occurs in the sample. For example, if branch c_2 of *simpleFunc* is exercised by input vectors $\{(1, 19), (3, 20), (1, 18), (1, 20), (3, 20)\}$, then h_{c_2} is 2 since the most frequent input vector, $(3, 20)$, occurs twice. The diversity is estimated as follows:

$$g_{\text{div}} = 1 - \frac{1}{|C|} \sum_{c \in C} \left(\frac{h_c}{\sum_{i=1}^{K} e_{c;i}} \right). \tag{9}$$

The term $h_c / \sum_{i=1}^{K} e_{c;i}$ estimates the diversity for a single-coverage element, with the denominator adjusting for the number of times the element was exercised by the sample of input vectors. If a single input vector tends to dominate—i.e., the sample of input vectors that exercises the coverage element is not diverse—then this term will be close to 1. The average of this value is taken over all of the coverage elements, and then subtracted from 1 so that more diverse probability distributions have a value of g_{div} that is numerically higher. This is for consistency with g_{cov}, which also takes higher values for distributions with better coverage.

An overall fitness is formed by a weighted sum of the distance of the coverage and diversity values from their respective targets:

$$\begin{aligned} f = {} & w_{\text{cov}} \max(t_{\text{cov}} - g_{\text{cov}}, 0) \\ & + w_{\text{div}} \max(t_{\text{div}} - g_{\text{div}}, 0), \end{aligned} \tag{10}$$

where w_{cov} and w_{div} are weightings that adjust the relative contributions of each constraint in guiding the search's trajectory. The weightings are parameters to the search algorithm: As for the other algorithm parameters, suitable values must be chosen that ensure an effective algorithm, by means of preliminary experimentation, for example.

The search method attempts to minimize the fitness with the objective of finding a distribution with a fitness of 0. This value indicates that both constraints have been satisfied.

5.3 Search Method

Stochastic hill climbing is used as the search method. The method is similar to the random mutation hill climbing algorithm described by Forrest and Mitchell in [31]. Preliminary experimentation with a number of local search methods indicated that stochastic hill climbing is sufficiently effective, in terms of performance and quality, in finding probability distribution for statistical testing, with the benefit of relative simplicity in terms of both implementation and the number of search parameters.

We explicitly made no attempt to find the *most* effective search method for the SUTs under investigation nor to find the *optimal* search parameter settings. Since one of the experimental objectives is to assess the practicality of the technique, extensive optimization of the search method and parameters for each SUT would not have been realistic.

The initial candidate solution is a BN, where all variables are independent and the marginal distribution of each variable is the uniform distribution over its domain. This choice of a constant, rather randomly-chosen, initial distribution facilitated the experimentation by controlling one source of stochastic variance.

At each iteration of the algorithm, a small sample, size λ, of the neighbors to the candidate solution is chosen at

random. Neighbors are defined as BNs formed by making one, and only one, of the following mutations:

- adding an edge between two nodes (as long as the graph does not become cyclic),
- removing an edge,
- multiplying or dividing the length of a bin interval by a fixed value,
- splitting a bin into two equally sized bins,
- joining two adjacent bins,
- multiplying or dividing the conditional probability of a bin by a fixed value.

These six mutations modify all of the nonconstant properties of the representation: the directed edges between nodes, the number and length of bins, and the probabilities assigned to bins can all be mutated by these operators, but the number of nodes—which is determined by the number of input arguments to the SUT—remains constant. The mutations are also reversible: Any single mutation has a corresponding mutation that reverses it. Nevertheless, the choice of these particular mutations is arbitrary, and it is possible that other mutation operators would result in a more effective algorithm.

All of the neighbors are equally likely to be sampled, apart from those formed by mutations that increase the number of bins in a node or increase the number of edges in the network graph. These neighbors have a smaller chance of being sampled than the others, the difference being controlled by a parameter $\rho < 1$. The decreased probability of these mutations encourages parsimony in the representation, and this was found to improve performance, to avoid unnecessarily large data structures, and to produce more compact solutions that are easier to interpret.

If the fittest of the sampled neighbors is fitter than the current candidate solution, it becomes the candidate solution at the next iteration. The search is terminated when a candidate solution is found with a fitness of 0, indicating that both the coverage and diversity constraints have been met, or if no solution has been found after a (large) number of iterations.

A particular consideration for the search method is that the fitness is noisy since both g_{cov} and g_{div} are estimates calculated from a finite sample from the candidate probability distribution.

The effect of noise is controlled in two ways. First, a relatively large sample size, K, is used so that the noise is reduced, and it does not overwhelm the search. Second, a further sample of input vectors is taken for the candidate solution at each iteration. This sample is combined with the existing sample, and the fitness is evaluated over the larger accumulated sample. Consider the case where a solution survives to the next iteration because its initial fitness estimate was much better than its "true" value owing to the noise. At subsequent iterations, the larger size of the accumulated sample enables increasingly better estimates of the "true" fitness, and the solution continues to survive only if it really has the best fitness.

5.4 Software Engineering Details

The search method is implemented using C++. The GNU Scientific Library [32] is called in order to generate pseudorandom numbers, using the Mersenne Twister algorithm.

For the experiments in this paper, the SUT functions were instrumented manually and linked directly with the search algorithm to form a single executable. Although the facility was created to call instrumented SUTs in the form of stand-alone executables, linking the function with the search algorithm—where this is possible—avoids the significant overhead of calling a separate SUT executable.

TABLE 1
SUT Characteristics

Characteristic	simpleFunc	bestMove	nsichneu
Lines of Code	15	89	1967
No. Input Arguments	2	2	11
Input Domain Cardinality	1000	2.621×10^5	3.395×10^6
Number of Loops	0	7	1
Coverage Elements	6	42	490
Mutants Tested	178	1923	6699

6 Experimental Design

To test the hypotheses proposed in Section 4, four sets of experiments were performed. This section describes the SUTs used for the experiments, the mechanism by which the fault-detecting ability of test sets was assessed and the design of each experiment.

6.1 Software Under Test

In order to demonstrate applicability of the proposed technique to different types of software, each experiment was performed on *simpleFunc* and two larger SUTs based on real-world examples.[1] Apart from their realistic size, the criteria for choosing the two real-world SUTs were that adequate probability distributions would be difficult to derive by either manual or analytical methods, and that they have contrasting characteristics.

Relevant characteristics of the SUTs are shown in Table 1, and the two real-world examples are described in more detail below. The table shows the number of elements, using a branch coverage criterion, and this coverage criterion was used for all experiments. The diversity metric proposed in Section 5.2 assumes a discrete input domain, and so, to permit experimentation on diversity, SUTs were chosen whose arguments were integer data types.

6.1.1 bestMove

This C function determines the best move for the current player in a tic-tac-toe (noughts-and-crosses) game. The two input arguments are the current board position of each player: When considered as a binary string, the set bits indicate locations occupied by the player. The function is adapted from a demo applet on Sun's Java Website [33]. Apart from conversion to C syntax, the major change was to increase complexity by adding code that validates the input arguments.

1. The source code of all three SUTs in available from: http://www.cs.york.ac.uk/~smp/supplemental.

Finding near-optimal probability distributions for this SUT is particularly difficult. For many coverage elements, there are only a few input vectors that exercise the element—e.g., code reached only when the player has a winning position—and the vectors belong to disconnected regions in the input domain.

6.1.2 nsichneu

This function consists of automatically generated C code that implements a Petri net. It forms part of a widely used suite of benchmark programs used in worst-case execution time research [34]. The complexity of this example arises from the large number (more than 250) of `if` statements and the data-flow between them. The function was modified to restore the number of iterations of the main loop to the original (larger) value in order to reduce the amount of unreachable code.

Not only does this function have a large number of coverage elements, the code itself takes significantly longer to execute than the other two SUTs, making fitness evaluation particularly time consuming.

6.2 Assessing Fault-Detecting Ability

The fault-detecting ability of a test set was assessed by its ability to detect mutant versions of the SUT.

The mutants were generated using the Proteum/IM tool [35]. Sets of mutants were created by making single-point mutations, using each of the large number of mutation operators built in to the tool. Since mutants were being used only to compare test sets, there was no need to identify and remove equivalent mutants. For *nsichneu*, a random selection consisting of 10 percent of all possible mutants were used so that the mutant set was of a practical size; for the other two SUTs, all of the possible mutants were used (see Table 1).

The *mutation score*—the proportion of mutants "killed" by a test set—is used as the measure of its ability to detect faults. A mutant was considered as "killed" if for one or more test cases, a mutant produced a different output or the process was terminated by a different operating signal, from the unmutated SUT. The testing process was configured so that mutant executables were terminated by the operating system if they used more than 1 s of CPU time. This limit is significantly longer than the CPU time required by the unmutated versions of the SUTs, and so identifies and terminates any mutant that entered an infinite loop.

Test sets were generated by random sampling of input vectors from the probability distribution *without replacement*: If an input vector was already in the test set, it was rejected and a further input vector was sampled. We believe this is consistent with how test sets would be generated in practice: Using more than one test case with the same input vector usually offers little benefit in terms of the number of faults detected. (Although not described here, initial experimentation showed that using a *with replacement* mechanism produced results that were qualitatively no different.)

6.3 Experiment A

Thirty-two runs of the search algorithm were performed and the proportion of searches that found distributions satisfying the coverage constraint—the statistical testing adequacy criterion—were measured. A diversity constraint was not applied in this experiment.

TABLE 2
Search Parameters

Parameter	simpleFunc	bestMove	nsichneu
Common			
No. of neighbors, λ	4	4	4
Bin length mutation ratio	10	10	10
Bin prob. mutation ratio	10	10	10
Parsimony parameter, ρ	0.8	0.8	0.8
Evaluation sample size, K	200	1000	1000
Experiment A			
Coverage target, t_{cov}	0.24	0.14	0.035
Maximum iterations	4000	60000	4000
Experiment D			
Coverage target, t_{cov}	0.24	0.10	0.025
Coverage weighting, w_{cov}	0.8	0.8	0.95
Diversity target, t_{div}	0.95	0.80	0.995
Diversity weighting, w_{div}	0.2	0.2	0.05

The searches were run on a server class machine running a customized version of Slackware Linux. Each search was a single-threaded process and used the equivalent of one core of a CPU running at 3 GHz with a 4 MB cache. The CPU user times taken by both successful (those finding a suitable distribution) and unsuccessful searches were recorded. The CPU user time is approximately equivalent to the wall-clock time when running on an otherwise unloaded server.

The parameters used for the search runs are listed in Table 2. Since the objective of the experiment is to show practicality of the technique, little time was spent in tuning the parameters and, where possible, the same parameters were used for all three SUTs. The only difference between the 32 runs were the seeds provided to the pseudorandom number generator (PRNG).

Using multiple runs in this way allows us to estimate the expected proportion of successful searches and the expected time taken across all possible seeds to the PRNG. The choice of 32 runs is a compromise between the accuracy of these estimates and the resources (computing power and time) that were available for experimentation. During the analysis of the results in Section 7.2, we provide confidence intervals for these estimates as a measure of the accuracy that was achieved.

The coverage target—equivalent to the lower bound of the coverage element probabilities—is near the optimal values for *simpleFunc* (0.25) and *bestMove* (0.1667). These optimal values are straightforward to determine by a manual examination of the control flow graph of both these SUTs. The optimal lower bound itself is not used as the coverage target since it is difficult to obtain this exact fitness, even when the distribution itself is optimal. It requires a random sample where each input vector occurs with a frequency that is exactly proportional to its probability in the distribution, and this is unlikely for the relatively small, finite samples used to evaluate the fitness.

The optimal lower bound for *nsichneu* is difficult to determine from its control flow graph owing to the complexity of the data dependencies. Instead, we use a

coverage target that is close to the best value found during preliminary experimentation.

This experiment is designed to provide evidence for Hypothesis 1, that it is practical to use automated search to derive probability distributions for statistical testing.

6.4 Experiment B

Experiment B measures the fault-detecting ability of test sets generated by 10 of the probability distributions found in Experiment A. For each of the 10 distributions, 10 test sets were generated by random sampling without replacement, using a different PRNG seed in each case. Each time a test case was added to the test set, the mutation score was assessed using the method described in Section 6.2.

For comparison with random testing, the same experiment was performed using a uniform distribution in place of that derived by automated search. One hundred test sets were generated from the uniform distribution so that the two samples had the same size.

This experiment is designed to provide data to demonstrate Hypotheses 2 and 3 that, when using automated search, the superior efficiency of statistical testing compared to uniform random and deterministic structural testing is maintained.

6.5 Experiment C

Experiment C compares the fault-detecting ability of probability distributions that have different lower bounds. It is designed to provide evidence for Hypothesis 4, that probability distributions with higher lower bounds generate more efficient test sets.

For each SUT, a sequence of probability distributions was taken at points along the trajectory of one search run. In general, such a sequence consists of distributions with increasing probability lower bounds. The search run used the same parameters as in Experiment A. Twenty test sets were generated from each distribution and their mutation scores assessed over a range of test sizes.

6.6 Experiment D

Experiment D compares the fault-detecting ability of test sets generated from distributions found by automated search with and without a diversity constraint. It provides data to support Hypothesis 5, that the use of a diversity constraint results in more efficient test sets.

A sample of 10 distributions was found using the parameters specified in Table 2. The parameters have a nonzero value for w_{div} and so add a diversity constraint to the search objectives. The coverage target parameters (t_{cov}) for *bestMove* and *nsichneu* were less than the near-optimal values of Experiment A in order to demonstrate that a lack of diversity has a significant effect even when using distributions with only moderate lower bounds.

For comparison, a further sample of distributions was found using the same parameters, but with w_{div} set to zero in order to disable the diversity constraint. More than 10 such distributions were found, and a subset of size 10 was selected in a principled manner so that the distributions of the coverage fitnesses across the two samples were as similar as possible. This was designed to minimize the effect of probability lower bounds on this experiment.

Ten test sets were generated from each distribution in the two samples, and their mutation scores assessed over a range of test sizes.

TABLE 3
Search Times (Experiment A)

	simpleFunc	bestMove	nsichneu
Proportion successful, π_+	1.0	$0.72\,^{+0.13}_{-0.19}$	$0.31\,^{+0.16}_{-0.16}$
CPU user time (mins)			
Successful search, T_+	$0.04\,^{+0.02}_{-0.01}$	$80\,^{+12}_{-14}$	$144\,^{+25}_{-30}$
Unsuccessful search, T_-		$124\,^{+3}_{-4}$	$204\,^{+5}_{-4}$
Until success, C_+	$0.04\,^{+0.02}_{-0.01}$	$129\,^{+76}_{-40}$	$592\,^{+702}_{-252}$

7 Results and Analysis

7.1 Statistical Analysis

In this section, results are summarized using the mean. Although the median is potentially a more robust statistic for the skewed distributions that the results could exhibit, it was found to be misleading when, on occasion, the data formed similarly sized clusters around two (or more) widely separated values. In this case, the median returned a value from one of the clusters, while the mean gave a more meaningful statistic located between the two clusters.

Confidence intervals quoted for the mean values, and the error bars shown in the graphs, are at the 95 percent confidence level. They are calculated using bootstrap resampling, specifically the bias corrected and accelerated percentile method [36].

Nonparametric statistical tests are used to analyze the data. Since parametric statistical tests can be invalidated by small deviations from the assumptions that the tests make [37], the use of nonparametric tests ensures the validity of the analysis, while avoiding the need to perform additional analysis to verify that the data conform to the test assumptions.

To compare samples, the nonparametric Mann-Whitney-Wilcoxon or rank-sum test is applied [38]. The null hypothesis for the rank-sum test is that the samples are from the same distribution; the alternate hypothesis is that the distributions are different. We apply the test at a 5 percent significance level.

Given a sufficiently large sample, hypothesis tests such as the rank-sum test can demonstrate statistically significant results even when the underlying differences are extremely small. Therefore, we use an additional test to show that the effect size—in this case, a difference in the ability to detect faults—is large enough to be meaningful, given the variability in the results. The nonparametric Vargha-Delaney A-test [39] is used here since its value can be calculated from the statistic used by the rank-sum test. We use the guidelines presented in [39] that an A-statistic of greater than 0.64 (or less than 0.36) is indicative of a "medium" effect size and greater than 0.71 (or less than 0.29), of a "large" effect size.

7.2 Experiment A

The results of Experiment A are summarized[2] in Table 3.

In this table, π_+ is the proportion of search algorithm runs that found a suitable probability distribution (i.e., with

2. The unsummarized data for all four experiments is available from: http://www.cs.york.ac.uk/~smp/supplemental.

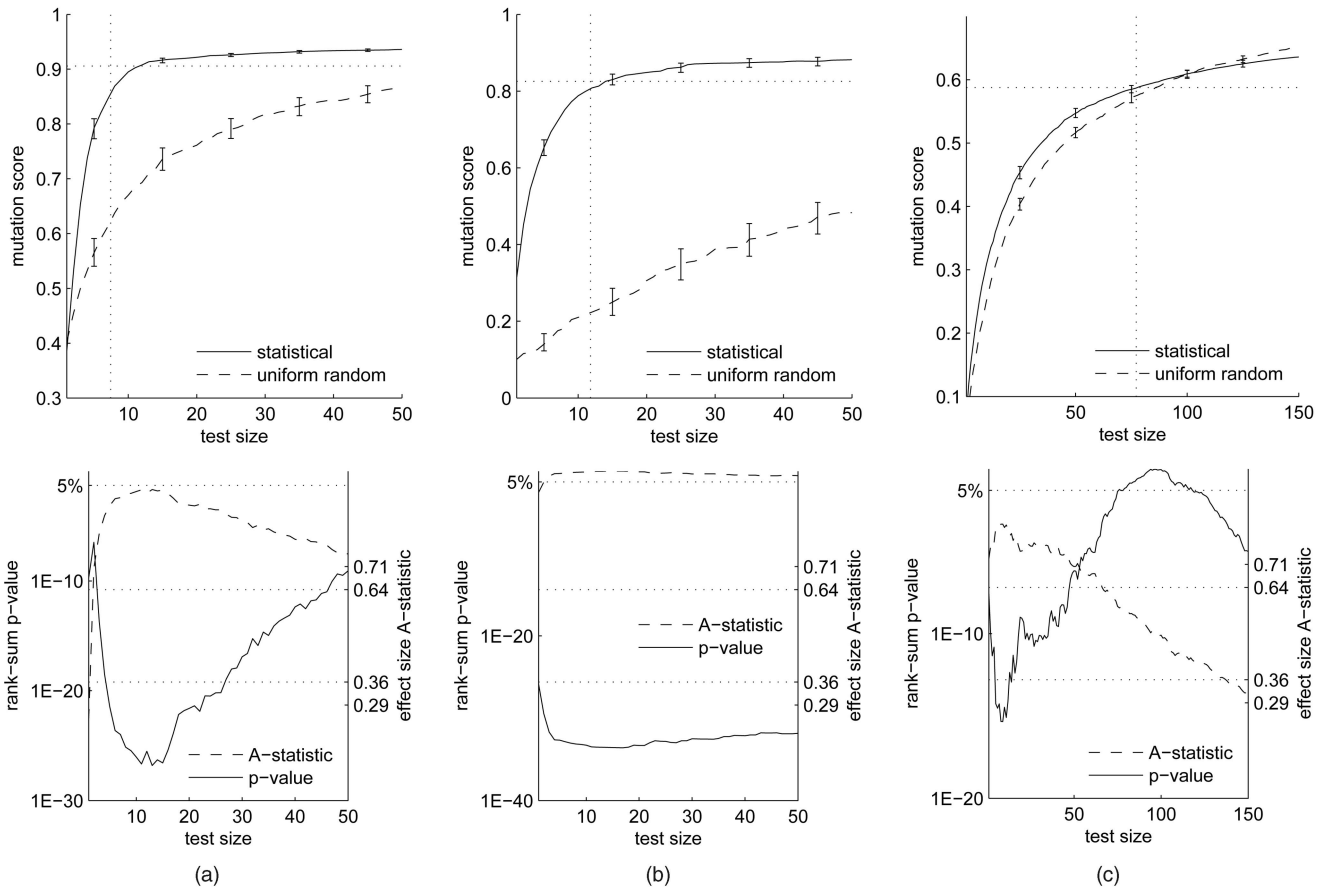

Fig. 4. Fault-detecting ability of statistical testing compared to uniform random testing (Experiment B). (a) simpleFunc, (b) bestMove, (c) nsichneu.

a probability lower bound greater than or equal to the target value, t_{cov} given in Table 2). T_+ is the mean time taken by these successful runs (measured as CPU user time), and T_- the mean time taken by unsuccessful runs.

We may consider each run of the algorithm to be equivalent to a flip of a coin where the probability of the coin landing with "heads up" or of the algorithm finding a suitable distribution is π_+. Trials such as these, where there are two possible outcomes and they occur with fixed probabilities, are termed *Bernoulli trials* [40]. In practice, after an unsuccessful search, we would run another search with a new, different seed to the PRNG; in other words, we would perform a series of Bernoulli trials until one is successful. The number of failures that occur before a successful outcome in a series of Bernoulli trials follows a geometric distribution [40], with the mean number of failures being given by $(1 - \pi_+)/\pi_+$ [41]. Therefore, we may calculate the mean time taken until a successful search occurs (including the time taken for the successful search itself) as:

$$C_+ = \frac{1 - \pi_+}{\pi_+} T_- + T_+. \tag{11}$$

The calculated values for C_+ are given in the Table 3 and are a prediction of how long, on average, it would take to find a suitable probability distribution.

It is a conservative prediction in that it assumes the availability of only one CPU core on which searches can be run. Software engineers are likely to have access to significantly more computing resources than this, even on their desktop PC, which would allow searches to be performed in parallel.

We argue that the predicted time until success is indicative of practicality for all three SUTs. The longest time—the value of C_+ for *nsichneu*—is approximately 10 hours, rising to 22 hours at extreme end of its confidence interval. These results provide strong evidence for Hypothesis 1, that automated search can be a practical method of deriving probability distributions for statistical testing.

7.3 Experiment B

The results of Experiment B are summarized in Fig. 4.

The upper row of graphs compares the mutation score of test sets derived from statistical and uniform random testing over a range of test sizes. The mutation scores were assessed at each integer test size in the range, but for clarity, error bars are shown at regular intervals only.

The lower row of graphs show the p-value (left-hand axis using a log scale) for the rank-sum test applied at each test size, and the effect size A-statistic (right-hand axis). Dotted lines indicate a 5 percent p-value and the lower boundaries of the medium effect size regions. p-values below the 5 percent line indicate that the differences in the mutation scores are statistically significant. A-statistic values above the 0.64 line, or below the 0.36 line, indicate a medium or large effect size.

The graphs demonstrate statistically significant differences in the mutation scores of test sets derived by statistical and uniform random testing, with the former having the

greater fault-detecting ability for most test sizes. The effect size is medium or large for many test sizes.

These results support Hypothesis 2 that test sets generated using statistical testing are more efficient than test sets generated by uniform random testing: For the same test size, the former test sets typically have greater fault-detecting ability.

The exception is for larger test sets (size > 100) applied to *nsichneu*. For these larger test sets, uniform random testing is significantly more effective at detecting faults. We suspect that this is due to a lack of diversity in the probability distribution, and this motivates the use of a diversity constraint during search.

The horizontal dotted lines in the upper row of graphs in Fig. 4 are the mean mutation score for each statistical test set at the size at which it first exercises all of the coverage elements in the SUT at least once. (The vertical dotted lines are the mean of these sizes.) At these sizes, the test sets satisfy the adequacy criterion of deterministic structural testing. However, due to the stochastic nature in which they are generated, they will usually contain more test cases than a minimally sized test set. The additional test cases can only improve the mutation score of the test set; therefore, their mutation scores—the horizontal dotted line—represent an upper bound for the mutation scores achievable by deterministic structural testing.

For each SUT, the mutation scores obtained by statistical testing exceed this upper bound at sufficiently large test sizes. This is evidence in support of Hypothesis 3, that test sets derived using statistical testing where each coverage element is exercised multiple times detect more faults than test sets typical of deterministic structural testing that exercise each element only once.

7.4 Cost-Benefit Analysis

The results of Experiments A and B taken together enable a simple cost-benefit analysis that compares statistical testing with uniform random testing. This can be performed in a number of ways: for example, by considering the difference in fault detecting ability of test sets of the same size for the two methods or the difference in the cost of testing for test sets of different size that have same fault detecting ability. We take the second of these two approaches in order to avoid speculation on the costs of later rectifying faults that are not discovered during testing, as these costs are highly dependent on the context.

If we choose a fault-detecting ability equivalent to a mutation score of 0.55 for *bestMove*, the results of Experiment B show that this occurs at a test size of 4 for statistical testing and at a test size of 61 for uniform random testing. (These sizes are obtained from the data used to plot the graphs of Fig. 4b.) Statistical testing incurs a cost in finding a suitable probability distribution, and from Table 3, this search takes, on average, 129 minutes. If executing, and—often more significantly—checking the results of the tests against a specification or an oracle takes longer than 129 minutes for the additional 57 test cases required by random testing, then statistical testing will be more efficient in terms of *time*. We argue that when checking test results involves a manual comparison against a specification, then the superior time efficiency of statistical testing is likely to be realized in this case. We also note that the search for a probability distribution is automated, requiring little or no manual effort, and so speculate that statistical testing is likely to be superior in terms of *monetary cost* (even in situations where it is not superior in terms of time) if the execution and checking of the additional 57 test cases is a predominantly manual, and therefore expensive, process.

We may repeat this analysis for *nsichneu*, again choosing a fault detecting ability equivalent to a mutation score of 0.55. In this case, random testing requires 14 more test cases than statistical testing: The test sizes are 66 and 52, respectively. The search for a probability distribution suitable for statistical testing takes 592 minutes. The superiority of statistical testing in terms of time is not as convincing for *nsichneu*: Random testing is quicker overall if executing and checking the extra 14 test cases takes no longer than 10 hours. However, we again note that the search for the probability distribution is automated, and so, considering monetary cost, statistical testing may be cheaper than random testing if checking the results of the extra 14 test cases is a manual process.

The simple analyses of this section are dependent on the mutation score we choose. Nevertheless, for *bestMove*, the large difference in the fault-detecting ability of the two methods shown by Fig. 4b suggests that the general conclusion remains the same for most choices of mutation score. For *nsichneu*, however, any benefit of statistical testing is lost for mutation scores above approximately 0.6. As can be seen in Fig. 4c, mutation scores above this level are achieved by random testing with test sizes that are smaller than those required by statistical testing. This underlines the importance of investigating whether the use of a diversity constraint improves the fault-detecting ability of statistical testing.

7.5 Experiment C

The results of Experiment C are summarized in Fig. 5. Each graph shows the mutation scores for a sequence of probability distributions taken from the trajectory of one search run. The probability lower bound for each distribution (evaluated accurately using a large sample size) is plotted on the x-axis, and the mutation score (with error bar) on the y-axis. The lines connect mean mutation scores calculated at the same test size. Note that the left-hand point in each graph is the uniform probability distribution that is used to initialize the search: This distribution is equivalent to that used for uniform random testing.

At small values of the lower bound—at the left of the graphs—mutation scores get better as the probability lower bound increases. However, for larger values of the lower bound at the right of Fig. 5c, the mutation score begins to decrease. There is also some evidence for this effect at the highest lower bound values in Fig. 5b.

We suspect that is because diversity can be lost as the search proceeds—the probability distribution is "over-fitted" to the coverage constraint—and so, despite a high lower bound, the distribution generates test sets that are relatively poor at detecting faults. This again motivates the use of a diversity constraint in order to avoid this loss of efficiency as the search proceeds.

The results of Experiment C provide some evidence in support of Hypothesis 4 that distributions with higher lower bounds generate test sets with a greater ability to detect faults. However, this is not true in all cases: Searching for near-optimal lower bounds can be counter-productive for some SUTs, and we suspect this is a result of losing diversity.

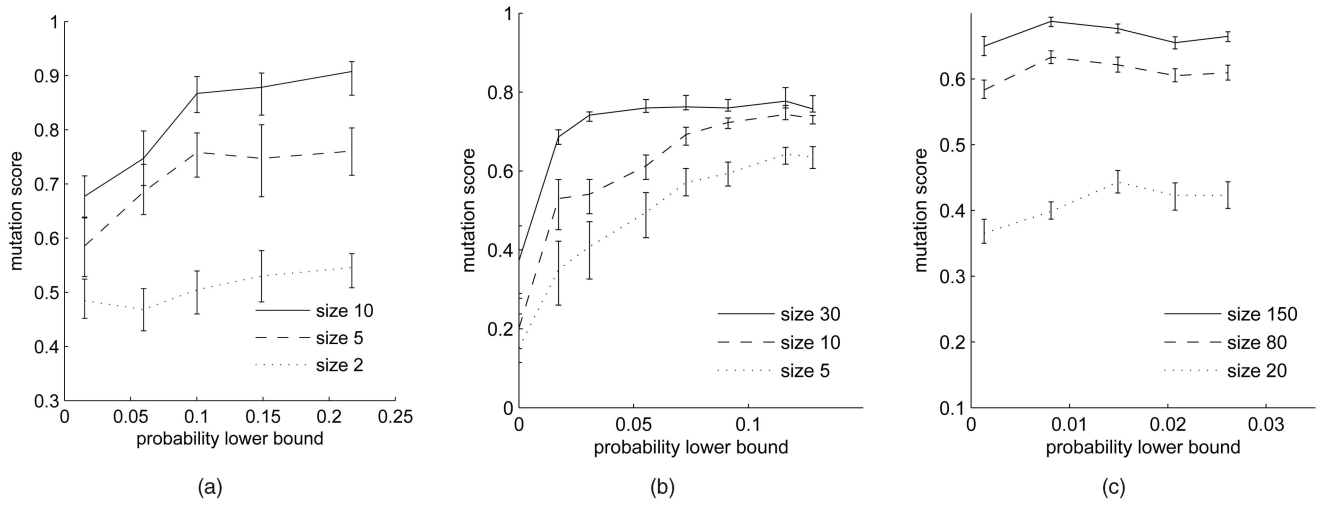

Fig. 5. Fault-detecting ability of probability distributions at a sequence of points along a search trajectory (Experiment C). (a) simpleFunc, (b) bestMove, (c) nsichneu.

7.6 Experiment D

The results of Experiment D are summarized in Fig. 6. The upper row of graphs compares the mutation score of test sets derived from probability distributions found, using automated search with and without a diversity constraint, and the lower row shows the rank-sum p-value and effect size A-statistic, as for Experiment B.

It can be seen that probability distributions found using a diversity constraint are better at detecting mutants. However, the p-values and A-statistics indicate that the effect is only significant for some test sizes. For *simpleFunc* and *bestMove*, the improvement is greatest for larger tests sizes. For *nsichneu*, the converse is true: The improvement

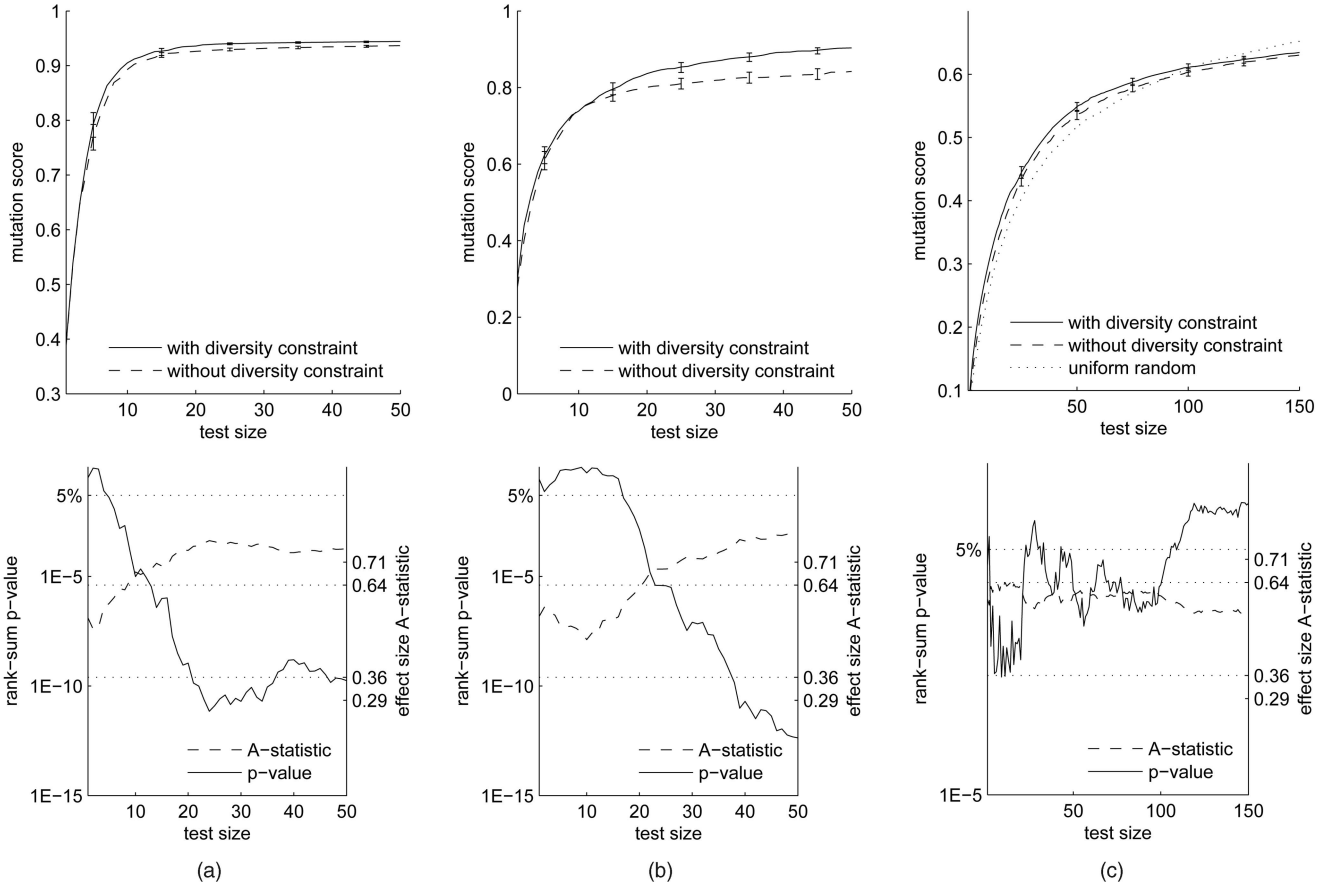

Fig. 6. Fault-detecting ability of probability distributions found by search with and without a diversity constraint (Experiment D). (a) simpleFunc, (b) bestMove, (c) nsichneu.

brought about by the diversity constraint is greatest at smaller test sizes, but the effect size is generally small.

The additional dotted line in Fig. 6c plots the mutation scores for uniform random testing taken from Experiment B. It can be seen that the improved efficiency enabled by the diversity constraint is insufficient to close the gap with uniform random testing at the largest test sizes.

These results provide partial evidence for Hypothesis 5 that using a diversity constraint during the search improves the efficiency of statistical testing. However, the nature of the improvement depends on both the SUT and the test size.

8 Conclusions and Future Work

The experimental results support a number of the hypotheses proposed in Section 4. We have shown that using automated search to derive near-optimal probability distributions for statistical testing is not only viable, but practical for different types of SUT (Hypothesis 1). We have demonstrated that statistical testing using automated search continues to show superior fault-detecting ability compared to uniform random and deterministic structural testing (Hypotheses 2 and 3). There was also some evidence that searching for distributions with the highest probability lower bounds resulted in the most efficient test sets (Hypothesis 4).

However, uniform random testing was more efficient than statistical testing at large test sizes for one of the SUTs, and distributions with near-optimal lower bounds showed diminished fault-detecting ability. We hypothesize that both effects result from a lack of diversity in the generated test data. Although the addition of a diversity constraint as a search objective did improve test set efficiency (Hypothesis 5), the results indicate that the metric used is probably too crude to retain all of the important forms of diversity in the generated data.

Further work is therefore indicated on different diversity metrics, with the goal of identifying a measure that correlates well with the ability of the test set to detect faults. In addition, we plan to expand the notion of diversity to nondiscrete input domains.

One of the challenges of researching software engineering techniques is the vast range of software to which a technique may be applied. We have experimented on three programs that we feel have contrasting characteristics. However, in the absence of a detailed understanding of the software characteristics that affect our proposed technique, it is inappropriate to extrapolate more widely. In particular, the cardinality of the input domain for the SUTs was relatively small as a result of using integer data types. It will be important to demonstrate that the technique remains effective and practical for larger input domains.

This suggests further work to investigate the software characteristics that affect the efficacy of our technique. As for other SBSE applications, a barrier to take up by software engineers is the lack of guidance on the types of problems for which use of search is effective, and how to configure the search algorithm based on the characteristics of the problem. For the experimental work in this paper, we took the conservative approach of using a simple search algorithm and used similar algorithm parameters across all three SUTs. However, greater scalability would be possible if more effective and faster algorithms are identified—particularly those that make full use of available computing resources through parallel processing—and if the optimal algorithm parameters could be set a priori based on relevant characteristics of the SUT.

An alternative to tuning the algorithm *in advance* based on the characteristics of the SUT would be to adapt parameter values *during* the algorithm run itself. Eiben et al. refer to this approach as *parameter control* in their recent survey of both parameter tuning and parameter control [42]. Given the vast range of software that may be tested (noted above) and the difficulty in identifying SUT characteristics that affect algorithm performance, parameter control may be a viable technique if it avoids the need to identify such characteristics.

Currently, the fitness metric for the coverage constraint uses a count of the elements exercised by a sample of input vectors. However, if an element is exercised by no input vectors in the sample, little guidance is provided to the search. In this case, the incorporation of metrics used by other search-based test data generation techniques—such as the approach level and branch distance discussed in Section 3—into the fitness function might also enable the technique to scale to larger and more complicated SUTs.

Another path for improvement in algorithm performance is the use of optimization methods that efficiently accommodate—or even make use of—the noise in the fitness function. (We speculate that some noise in the fitness may actually be beneficial to the hill climbing algorithm used for this paper by occasionally permitting moves to less fit distributions in order to escape nonglobal local optima.) A number of methods are suggested by existing work on noisy fitness functions for simulated annealing and evolutionary algorithms [43], [44], [45], [46].

Experiment D used two competing objectives—the coverage and diversity constraints—to find the most efficient probability distribution. It might therefore be constructive to apply optimization techniques that are explicitly multiobjective, rather than using a single fitness function that combines the constraints, as we did in this paper. This use of efficient multi-objective optimization algorithms is an approach taken by many recent SBSE applications [7], [10], [13], [47].

The wider applicability of the search technique proposed in this paper requires an extension of the representation to other input data types. The current use of real numbers in the internal representation of probability distributions, and of binning to control the size of the representation, promises a relatively straightforward extension to floating point arguments. However, the incorporation of nonscalar data types, such as strings, objects, and pointers, will be a significant challenge, and is a current topic of research for other search-based test data generation techniques (e.g., [48]).

Finally, there is scope to search for distributions satisfying broader adequacy criteria. The coverage elements could be specified by other testing objectives, such as coverage of the software's functionality. The criterion on the coverage probability distribution could be expressed in terms of properties other than the lower bound, such as an increased probability of exercising parts of the software that have previously shown a propensity for faults, or that are particularly critical to the correct or safe operation of the program.

ACKNOWLEDGMENTS

The authors would like to thank Jose Carlos Maldonado and Auri M. R. Vincenzi for their assistance with the Proteum/IM mutation testing tool, Marie-Claude Gaudel for her discussion of statistical testing, and Hélène Waeselynck for clarifying the techniques used in her work with Pascale Thévenod-Fosse and which inspired the research described in this paper. This work is funded by Engineering and Physical Sciences Research Council grant EP/D050618/1, SEBASE: Software Engineering by Automated SEarch.

REFERENCES

[1] P. Thévenod-Fosse and H. Waeselynck, "An Investigation of Statistical Software Testing," *J. Software Testing, Verification and Reliability,* vol. 1, no. 2 pp. 5-26, 1991.

[2] P. Thévenod-Fosse, "Software Validation by Means of Statistical Testing: Retrospect and Future Direction," *Dependable Computing for Critical Applications,* A. Avizienis and J. C. Laprie, eds., pp. 23-50, Springer, 1991.

[3] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet, "Software Statistical Testing," Technical Report 95178, Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS (LAAS), 1995.

[4] P. Thévenod-Fosse and H. Waeselynck, "Statemate Applied to Statistical Testing," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 99-109, 1993.

[5] M. Harman and B.F. Jones, "Search-Based Software Engineering," *Information and Software Technology,* vol. 43, pp. 833-839, 2001.

[6] J. Clark, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd, "Reformulating Software Engineering as a Search Problem," *IEE Proc. Software,* vol. 150, no. 3, pp. 161-175, 2003.

[7] Y. Zhang, M. Harman, and S.A. Mansouri, "The Multi-Objective Next Release Problem," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1129-1137, 2007.

[8] E. Alba and F. Chicano, "Software Project Management with GAs," *Information Sciences,* vol. 177, no. 11, pp. 2380-2401, 2007.

[9] P. Emberson and I. Bate, "Minimising Task Migrations and Priority Changes in Mode Transitions," *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symp.,* pp. 158-167, 2007.

[10] M. Harman and L. Tratt, "Pareto Optimal Search Based Refactoring at the Design Level," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1106-1113, 2007.

[11] C. Hao, J.A. Clark, and J.L. Jacob, "Synthesising Efficient and Effective Security Protocols," *Proc. Workshop Automated Reasoning for Security Protocol Analysis,* pp. 25-40, 2004.

[12] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability,* vol. 14, no. 2, pp. 105-156, 2004.

[13] D. White, J.A. Clark, J. Jacob, and S. Poulding, "Evolving Software in the Presence of Resource Constraints," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1775-1782, 2008.

[14] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proc. 29th Int'l Conf. Software Eng. Future of Software Eng.,* pp. 342-357, 2007.

[15] J. Duran and S. Ntafos, "An Evaluation of Random Testing," *IEEE Trans. Software Eng.,* vol. 10, no. 4, pp. 438-444, July 1984.

[16] D. Hamlet, "When Only Random Testing Will Do," *Proc. First Int'l Workshop Random Testing,* pp. 1-9, 2006.

[17] H. Zhu, P.A.V. Hall, and J.H.R. May, "Software Unit Test Coverage and Adequacy," *ACM Computing Surveys,* vol. 29, no. 4, pp. 366-427, 1997.

[18] M.J. Harrold, R. Gupta, and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methodology,* vol. 2, no. 3, pp. 270-285, 1993.

[19] J.A. Jones and M.J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. Software Eng.,* vol. 29, no. 3, pp. 195-209, Mar. 2003.

[20] S. Ntafos, "On Random and Partition Testing," *ACM SIGSOFT Software Eng. Notes,* vol. 23, no. 2, pp. 42-48, 1998.

[21] *DO-178B, Software Considerations in Airborne Systems and Equipment Certification,* Radio Technical Commission for Aeronautics, 1992.

[22] *Defence Standard 00-55(PART2)/Issue 2, Requirements for Safety Related Software in Defence Equipment, Part 2: Guidance,* Ministry of Defence, 1997.

[23] P. Thévenod-Fosse and H. Waeselynck, "Towards a Statistical Approach to Testing Object-Oriented Programs," *Proc. IEEE Ann. Int'l Symp. Fault Tolerant Computing,* pp. 99-108, 1997.

[24] A. Denise, M.-C. Gaudel, and S.-D. Gouraud, "A Generic Method for Statistical Testing," Technical Report 1386, Laboratoire de Recherche en Informatique (LRI), CNRS—Univ. de Paris Sud, Apr. 2004.

[25] N. Baskiotis, M. Sebag, M.-C. Gaudel, and S. Gouraud, "A Machine Learning Approach for Statistical Software Testing," *Proc. Int'l Joint Conf. Artificial Intelligence,* pp. 2274-2279, 2007.

[26] R.P. Pargas, M.J. Harrold, and R.R. Peck, "Test-Data Generation Using Genetic Algorithms," *Software Testing, Verification and Reliability,* vol. 9, pp. 263-282, 1999.

[27] B. Korel, "Automatic Software Test Data Generation," *IEEE Trans. Software Eng.,* vol. 16, no. 8, pp. 870-879, Aug. 1990.

[28] N. Tracey, J. Clark, and K. Mander, "Automated Program Flaw Finding Using Simulated Annealing," *ACM SIGSOFT Software Eng. Notes,* vol. 23, no. 2, pp. 73-81, 1998.

[29] N.J. Tracey, "A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software," PhD dissertation, Dept. of Computer Science, Univ. of York, 2000.

[30] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing," *Information and Software Technology,* vol. 43, pp. 841-854, 2001.

[31] S. Forrest and M. Mitchell"Relative Building-Block Fitness and the Building Block Hypothesis," *Foundations of Genetic Algorithms 2,* D.L. Whitley, ed., pp. 109-126, Morgan Kaufmann, 1993.

[32] M. Galassi et al., *GNU Scientific Library Reference Manual,* second ed. Network Theory, 2006.

[33] "JDK 1.4 Demo Applets." http://java.sun.com/applets/jdk/1.4/index.html, 2010.

[34] "WCET Analysis Project." http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2010.

[35] M.E. Delamaro and J.C. Maldonado, "Proteum/IM 2.0: An Integrated Mutation Testing Environment," *Mutation Testing for the New Century,* pp. 91-101, Kluwer Academic Publishers, 2001.

[36] T.J. DiCiccio and B. Efron, "Bootstrap Confidence Intervals," *Statistical Science,* vol. 11, no. 3, pp. 189-212, 1996.

[37] N.L. Leech and A.J. Onwuegbuzie, "A Call for Greater Use of Nonparametric Statistics," technical report, US Dept. of Education, Educational Resources Information Center, 2002.

[38] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics Bull.,* vol. 1, no. 6, pp. 80-83, 1945.

[39] A. Vargha and H. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *J. Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132, 2000.

[40] E.W. Weisstein, "'Bernoulli Distribution.' From MathWorld—A Wolfram Web Resource." http://mathworld.wolfram.com/BernoulliDistribution.html, 2010.

[41] E.W. Weisstein, "'Geometric Distribution.' From MathWorld—A Wolfram Web Resource." http://mathworld.wolfram.com/GeometricDistribution.html, 2010.

[42] A.E. Eiben, Z. Michalewicz, M. Schoenauer, and J.E. Smith, "Parameter Control in Evolutionary Algorithms," *Parameter Setting in Evolutionary Algorithms,* F.G. Lobo, C.F. Lima, and Z. Michalewicz, eds., vol. 54, pp. 19-46, Springer, 2007.

[43] W.J. Gutjahr and G.C. Pflug, "Simulated Annealing for Noisy Cost Functions," *J. Global Optimization,* vol. 8, no. 1, pp. 1-13, 1996.

[44] S. Rana, D.L. Whitley, and R. Cogswell, "Searching in the Presence of Noise," *Parallel Problem Solving from Nature,* pp. 198-207, Springer, 1996.

[45] H.-G. Beyer, "Evolutionary Algorithms in Noisy Environments: Theoretical Issues and Guidelines for Practice," *Computer Methods in Applied Mechanics and Eng.,* vol. 186, pp. 239-267, 2000.

[46] S. Markon, D.V. Arnold, T. Bäck, T. Beielstein, and H.-G. Beyer, "Thresholding—A Selection Operator for Noisy ES," *Proc. 2001 IEEE Congress on Evolutionary Computation,* pp. 465-472, 2001.

[47] S. Yoo and M. Harman, "Pareto Efficient Multi-Objective Test Case Selection," *Proc. 2007 Int'l Symp. Software Testing and Analysis,* pp. 140-150, 2007.

[48] K. Lakhotia, M. Harman, and P. McMinn, "Handling Dynamic Data Structures in Search Based Testing," *Proc. Genetic and Evolutionary Computation Conf.,* pp. 1759-1766, 2008.

Simon Poulding received the BSc degree in mathematics and the MSc degree in mathematics with modern applications from the University of York in 1989 and 2006, respectively. From 1989 to 2005, he was a software engineer in the financial and healthcare industries, specializing in enterprise management systems and database applications. He is currently a lecturer in the Department of Computer Science, University of York. His research interests include the scalability and robustness of search-based software engineering algorithms and the use of reliable and efficient empirical methods in computer science.

John A. Clark received the MA degree in mathematics and the MSc degree in applied statistics from the University of Oxford in 1985 and 1986, respectively, and the PhD degree in computer science from the University of York in 2002. After industrial experience in security research and development, he joined the Department of Computer Science, University of York, as a CSE lecturer in safety-critical systems. He is currently a professor of critical systems at York, where he leads a team of researchers working on software engineering and security. He has a particular interest in optimization-based approaches to applications from both of these fields.