EXPECTED TIME TO DETECTION OF INTERACTION FAULTS

RENÉE C. BRYCE AND CHARLES J. COLBOURN

ABSTRACT. Software interaction test suites serve two complementary roles. They are employed to systematically verify that, for some strength t, no t-way interaction of a system's parameters causes a fault. They are also employed to locate a faulty configuration, when at least one interaction fault remains. Algorithms to find such test suites employing a number of tests close to the minimum have been extensively explored, in order to test all t-way interactions. However, when faults remain, the expected number of tests needed to reveal an interaction fault is also important. One might anticipate that the test suites of minimum size also have the lowest expected time to detection of an interaction fault; or, at the very least, that some test suite of minimum size does. However, in this paper it is shown that minimum test suite size and lowest expected time to fault detection are incompatible objectives. This underlies a challenging problem of how to generate test suites that have early coverage of t-way interactions, in order to reduce time to fault detection. A hybrid approach is developed that combines a simple greedy algorithm with heuristic search to construct one test at a time while attempting to maximize the number of t-way interactions covered by the earliest tests.

1. INTRODUCTION

Software testers often test for defects that they anticipate, while unforeseen defects, particularly those arising from interactions among components, are overlooked. To address this, numerous combination strategies have been developed [24]. In this paper, we focus on one of these, interaction testing [1,3,7,19,22,28,29,43]. Interaction testing is intended to reveal faults that are triggered by interactions; in the remainder of this paper, we use the term "fault" primarily in this context. Interaction testing is not a methodology to replace existing testing; rather it can augment current testing techniques with the systematic examination of interactions. It is applied both to check that a system exhibits no faulty interaction among few of its components prior to its release, and to determine as a part of integration or configuration testing that an interaction fault remains. These

applications appear to be identical; only the anticipated outcomes (absence versus presence of an interaction fault) differ. Nevertheless, testing to certify the absence of an interaction fault, and testing to certify the presence of (at least) one, turn out to be quite different.

Suppose that a software system has k *factors* or components that affect the system operation. Let $f_1, \ldots, f_k$ be the factors, and suppose that, for $1 \leq i \leq k$, factor f_i has a number v_i of admissible *levels* or settings. When each factor, set to a specific level, operates correctly in isolation, faults may nevertheless arise due to the interaction of more than one factor. Often one can attribute such an interaction fault to a relatively small number of factors and their levels. In general, a *strength* t is selected, large enough so that interactions of t or fewer factors reveal the potential faults. A *t-way interaction* is a selection of t factors, and an admissible level for each. The objective in interaction testing is to determine whether or not any s-way interaction for $s \leq t$ is faulty. To do this, form tests by choosing a level for each of the factors, and determine whether or not the system behaves as required with these settings; if not, a fault has been detected. A test suite is a collection of tests that suffices to detect the interaction faults involving t or fewer factors, if any are present.

We use a matrix representation. A *software interaction test suite* of *strength* t is an $N \times k$ array with N rows each representing a *test*, k columns so that column i represents *factor* f_i, and v_i symbols permitted in column i representing allowed *levels* of f_i. These are also known as *covering arrays*. A test suite is of *type* v^k when it has k factors with v levels each. More generally, its type is $v_1^{k_1} \cdots v_\ell^{k_\ell}$ when it has $k = \sum_{i=1}^{\ell} k_i$ factors, of which k_i have v_i levels for each $1 \leq i \leq \ell$. Then a t-way interaction is a set $\{(c_1, \sigma_1), \ldots, (c_t, \sigma_t)\}$ where each c_i indexes a column of the array, and each σ_i is a symbol that can appear in that column. The defining property of an interaction test suite is that for every possible t-way interaction $\{(c_1, \sigma_1), \ldots, (c_t, \sigma_t)\}$, there is some row (test) in which the level in column c_i is σ_i, for $1 \leq i \leq t$; we say that the test *covers* the interaction. *Pairwise testing* is the case when $t = 2$; *Higher strength* is the case when $t > 2$. As an example, Table 1(a) shows four factors that each have three levels and has type 3^4. An exhaustive test suite has 81 ($= 3^4$) tests, but pairwise interaction testing requires only 9 tests as in Table 1(b).

If the entire test suite is to be executed, testing cost correlates directly with the number of tests required. Hence minimizing the number of tests has been the central problem explored in the literature. We call this the *minimum size* problem. Naturally, this minimum depends on the number of factors, the number of levels, and the strength. Determining a strength t that is sufficient to detect faults is a challenging problem. Kuhn *et al.*

TABLE 1. (a) Example system of four components with three levels each. (b) A pairwise interaction test suite

f_0	f_1	f_2	f_3
0	3	6	9
1	4	7	10
2	5	8	11

(a) Type: 3^4

	f_0	f_1	f_2	f_3
1	0	3	6	9
2	0	4	7	10
3	0	5	8	11
4	1	3	8	10
5	1	4	6	11
6	1	5	7	9
7	2	3	7	11
8	2	4	8	9
9	2	5	6	10

(b)

compare reported bugs for the Mozilla web browser against results of interaction testing [28]. More than 70% of bugs are identified with 2-way interactions; approximately 90% of bugs are identified with 3-way interactions; and 95% of errors are detected by 4-way interactions. In [19,29], two other studies are given concerning the efficacy of higher strength interaction testing.

Many computational methods have been employed for addressing the minimum size problem. The most prevalent are greedy methods. One basic strategy adds one test at a time [6,9,39]; when a suitable test is chosen, it provides a strong theoretical guarantee on the size of the test suite produced [4, 5], and it provides a natural method to prioritize tests [3]. A second greedy strategy adds one factor at a time [23, 30, 38]. Often these produce test suites of acceptable size, but when more sophisticated methods can be applied they typically outperform greedy methods. Exact methods have been employed in [16, 35, 42], but the computational cost limits their application to few factors and small strength. Metaheuristic techniques have proved more successful, for example simulated annealing [11–13, 36, 37], tabu search [31], constraint satisfaction [27], genetic algorithms and ant colony techniques [34]. The combinatorial problem of constructing interaction test suites has also been extensively studied in the mathematical literature; see [15,25] for older surveys, and [17,18] for two of the main combinatorial constructions. Computational methods using these techniques to construct test suites appear in [8,26,41].

Substantial effort has gone into the development of methods to construct small test suites. In applying them, the primary objective is to detect the presence or absence of interaction faults; see [14,43] for examples. With

this in mind, there are two natural objectives. When t-way interaction faults are not present, our objective is to determine that using the fewest tests. But when faults are to be found, our objective is instead to minimize the expected time to find the first fault. These two objectives appear at first glance to be consistent with each other; indeed minimum test suite size is typically taken to imply fast fault detection. One primary contribution of this paper is to demonstrate that, while minimum size and minimum expected time to fault detection are cosmetically similar, neither is sufficient to ensure the other. In fact, as we show in Section 2, there are situations in which no test suite of minimum size leads to minimum expected time for fault detection, and conversely, no test suite with minimum expected time to fault detection has minimum size.

This should cause a software tester to determine their true objective carefully. Placing too much emphasis on test suite size can force an increase in the expected time to fault detection! Despite this difference, essentially all techniques have addressed minimum size. Therefore in Section 3, we adapt one-test-at-a-time greedy methods by using heuristic search techniques to select the next test, in order to produce test suites that reduce the expected time to fault detection. Section 4 provides empirical results, which show that the hybrid technique works better than either the greedy or heuristic search techniques alone. Section 4.2 examines the hybrid technique based on a different greedy algorithm. Section 5 provides conclusions.

2. MINIMUM SIZE OR MINIMUM EXPECTED TIME TO FAULT DETECTION?

Given a number of factors k, a number of levels v for each factor, and a strength t, the *minimum size* problem asks for the smallest N for which an $N \times k$ software interaction test suite on v symbols of strength t exists. Consider its application in the detection of interaction faults. Naturally we hope that all tests will be run, and none will reveal a fault. However, if there is a fault, and the tests of the array are run sequentially (i.e., executing the test corresponding to the first row, then the second, and so on), the first test in which a faulty interaction is encountered enables us to certify the software as faulty, and the remaining tests are not needed. Let us suppose that we have no *a priori* information about which t-way interaction might be faulty, if any. Then the natural objective is to ensure that, after running the ith test, we have covered the largest number of t-way interactions that *any* collection of i tests could cover. The best we might hope for is that this holds for every number i of tests run, until all t-way interactions are covered.

In order to measure the goodness of a test suite at detecting a fault early, let $\mathcal{T}$ be the set of all t-way interactions. For every t-way interaction $T \in \mathcal{T}$, compute the index $\rho(T)$ of the first row of the test suite that covers T. If T is the only faulty interaction, exactly $\rho(T)$ tests are executed in order to detect the presence of a fault. Therefore if every t-way interaction is equally likely to be the faulty one, the expected time to detect the fault is $\sum_{T \in \mathcal{T}} \rho(T)$ divided by the total number of t-way interactions. Denote by $\Lambda(t, (v_1, \ldots, v_k))$ the number of t-way interactions to cover for a test suite of strength t, having k factors with v_i levels for $1 \leq i \leq k$. Then a simple recursion can be used to compute this number: $\Lambda(t, (v_1, \ldots, v_k)) = 0$ if $t > k$; 1 if $t = 0$; and $v_1 \Lambda(t-1, (v_2, \ldots, v_k)) + \Lambda(t, (v_2, \ldots, v_k))$ otherwise.

The sum $\sum_{T \in \mathcal{T}} \rho(T)$ can also be calculated more directly. For each test S_i, $1 \leq i \leq N$, compute the number τ_i of t-way interactions that are covered by S_i but not covered by S_j for any $1 \leq j < i$. Then there are exactly τ_i t-way interactions T for which $\rho(T) = i$. Let $u_i = \sum_{\ell=i}^{N} \tau_\ell$, so that u_i is the number of interactions that are covered in a test numbered i or larger (that is, the number of uncovered interactions before executing the ith test). Because the total number of t-way interactions is $\Lambda(t, (v_1, \ldots, v_k))$, we obtain an explicit formula for the expected time to fault detection, when there is exactly one fault to be found:

$$\text{(1)} \qquad \frac{\sum_{i=1}^{N} i\tau_i}{\Lambda(t, (v_1, \ldots, v_k))} = \frac{\sum_{i=1}^{N} u_i}{\Lambda(t, (v_1, \ldots, v_k))}$$

The denominator in this ratio is independent of the particular test suite chosen, as is $\sum_{i=1}^{N} \tau_i = \Lambda(t, (v_1, \ldots, v_k))$. Therefore to reduce expected time to fault detection, the only opportunity is to cover more interactions earlier in the test suite; hence we want 'early coverage'.

When there are multiple faults, one could ask for the time to find the first, or the time to find all. In our context, finding the first is the more natural extension. This can again be easily calculated. If there are s faulty interactions and we have no *a priori* information about their location, the expected number of tests to detect the presence of a fault is

$$\text{(2)} \qquad \Phi_s = \frac{\sum_{i=1}^{N} \binom{u_i}{s}}{\binom{\Lambda(t, (v_1, \ldots, v_k))}{s}}$$

The question of minimizing expected time to fault detection is: Given a number k of factors, a number v of levels for each, and a strength t, and a number s of faulty t-way interactions, construct a software interaction test suite consisting of a set of tests $S_1, \ldots S_M$ that minimize (2). The test suite size is precisely Φ_0, as one would expect. When $s \geq 1$, if one test suite has uncovered interactions counted by $(u_1, \ldots, u_N)$ and a second

has $(u'_1, \ldots, u'_M)$, and $u_i \leq u'_i$ for all $1 \leq i \leq \min(N, M)$, the first has an expected time to detect a fault at least as low as the second, no matter how many faults are present. Hence the rate of coverage of interactions is crucial in obtaining low expected time to fault detection.

Rate of coverage is also important in other problems faced by testers. When a test suite has been constructed, but the time allocated to testing is shortened with little notice, a reasonable objective is to minimize the probability of an undetected fault after a specified number of tests have been run. If s random faults are present and $i-1$ tests have been run, this probability is just $\binom{u_i}{s}/\binom{\Lambda(t,(v_1,\ldots,v_k))}{s}$. We may not know in advance how many tests can be completed; in this case a reasonable objective is to minimize the average value of $\binom{u_i}{s}/\binom{\Lambda(t,(v_1,\ldots,v_k))}{s}$ over all numbers $0 \leq i-1 < N$ of tests; by (2), this is just Φ_s, and the problem is the same as that of minimizing the expected time to fault detection.

A further application also arises in the execution of portions of test suites. Suppose that a target coverage is to be specified, stating that a certain fraction f, $0 < f \leq 1$, of t-way interactions must be tested. This might occur, for example, when the goal is to find most but not all interaction faults. Choose $\ell = \min(i : u_i \leq f\Lambda(t,(v_1,\ldots,v_k)))$. Then the number of tests needed to cover fraction f of the interactions is exactly ℓ, which should be minimized. Once again, if the fraction f is not known in advance, one can average over all choices of the fraction f. Then the expected number of tests required is exactly Φ_s.

In order not to presuppose a certain number of faults, we can focus on the numbers of uncovered interactions, given by $(u_1, \ldots, u_N)$, to reduce as far as possible each u_i; or equivalently to increase as far as possible the numbers $(\tau_1, \ldots, \tau_N)$ of t-way interactions covered for the first time by each test.

2.1. **Small Test Suite Size Does Not Ensure Early Coverage.** We first consider seven factors with five levels each (type 5^7) of strength four. One might hope that a simple method that selects tests uniformly at random would afford good coverage among initial tests. A randomly generated test suite ('Random') had 6,382 tests. The large number arises in part because many tests generated are redundant, in that they cover no interactions not covered by an earlier test. Eliminating these redundant tests forms a suite, 'Random Unique', with 2,481 tests. A simple greedy one-test-at-a-time algorithm [5] provides a suite, 'One-Test-At-A-Time', of size 1,222. In [2] a combinatorial method is developed to form a test suite, 'Combinatorial (BC)', with 1,100 tests that is smaller than any that had

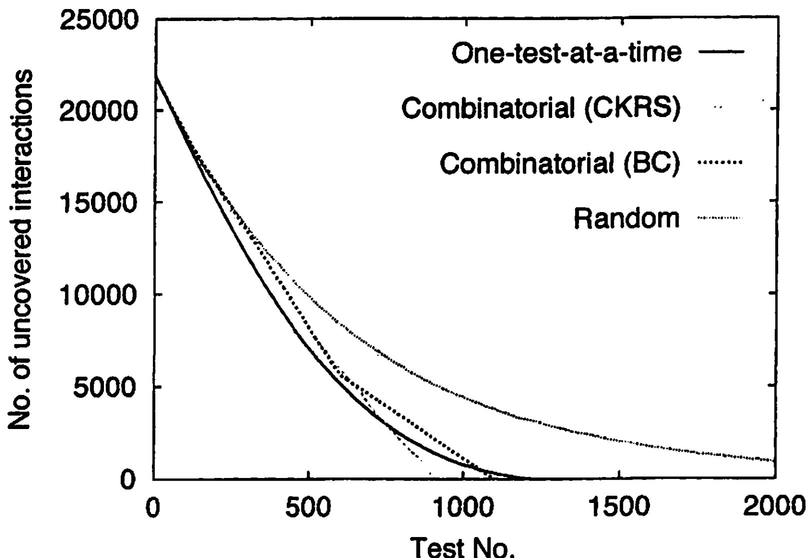


FIGURE 1. Rate of 4-way interaction coverage for type 5^7.

been previously published for these parameters. In [16], this has been dramatically improved on by a test suite, 'Combinatorial (CKRS)', with only 910 tests. The ultimate sizes of the test suites are strikingly different!

The rates of 4-way interaction coverage for the Random, One-Test-At-A-Time, Combinatorial (BC), and Combinatorial (CKRS) test suites are shown in Figure 1. One-Test-At-A-Time appears to be much more effective at covering 4-way interactions early on. In this regard it appears to outperform both of the smaller test suites. To make this precise, Table 2 gives the expected time to fault detection for the five suites.

TABLE 2. Expected Time to Detect 0, 1, 2, or 3 Faults for Five Test Suites of Type 5^7 and Strength 4

Suite	Φ_0	Φ_1	Φ_2	Φ_3
Random	6382	629.0	314.0	209.3
Random Unique	2481	603.1	312.1	208.8
One-Test-At-A-Time [5]	1222	394.3	234.3	167.8
Combinatorial (BC) [2]	1100	432.6	263.6	190.1
Combinatorial (CKRS) [16]	910	405.3	254.3	183.8

2.2. **The Effectiveness of Random Testing.** Random testing not only generates a much larger number of tests, it also fails to cover interactions as early as the other suites. If we are willing to track uncovered interactions, the size of the test suite improves dramatically, but the reduction in time to detect interaction faults is modest. In addition, the effort to track coverage of interactions is substantial, and doing so eliminates the primary reason for random generation of tests. Nevertheless, these results are at variance with some prior comparisons of interaction testing and random testing.

Dalal and Mallows [20] report that forming a test suite with complete coverage of t-way interactions, a random test suite with the same number of tests often has almost all t-way interactions covered and hence often provides an effective alternative to interaction test suites. When one limits their comparison to those involving a test suite with complete coverage, however, random suites provide coverage ranging from 65.4% to 99.9% for strength two, and from 65.6% to 93.1% for strength three. Even in these cases, the comparison is not definitive. For example, they employ a test suite for type 3^{13} of strength two with 27 tests providing complete coverage, while a random suite with 27 tests provides 95.8% coverage. Since that time, many constructions for software interaction test suites have been developed that affect this comparison. A test suite with complete coverage for these parameters exists having only 15 tests [31]; a random suite with 15 tests provides only 76.6% coverage. Arguably, the conclusion to draw is that, when no small interaction test suite with complete coverage is available, random testing provides a reasonable alternative. Hence it is more a commentary on the weaknesses of constructions that existed at that time than on the relative merits of interaction testing and random testing.

Schroeder, Bolaki, and Gopu [33] undertake a more detailed empirical analysis comparing interaction testing and random testing. In addition to the issue of coverage obtained by random testing, they argue that ultimately our concern is with fault detection, not with coverage. By injecting faults in software, and using both an interaction test suite and a random test suite, they compared their effectiveness at detecting a fault. The faults injected are not selected to be faults caused by an improper interaction; indeed in their study nearly half of the faults are not detected by either approach. When an injected fault is evidenced by a faulty interaction, the number of interactions made faulty by the single fault injected in the source is somewhat unpredictable. They choose an interaction test suite using a one-test-at-a-time greedy method, and then employ a random test suite with the same number of tests; they report the detection of a fault after all tests are run, and do not address the expected time to fault detection. Nevertheless, their study serves as a note of caution. Factors and their

levels affecting correctness must be selected appropriately for the problem at hand, and faults to be found should be the result of interactions among these. (See [20] for a discussion.) Once these requirements have been met, it remains the case that a deterministic method to produce a test suite with complete coverage may not provide more effective fault detection than a random suite, because the random suite itself may provide adequate coverage. This will happen if the interaction test suite chosen fails to cover many interactions in the earliest tests. Hence their study supports the need to clarify the differences between rate of fault detection and test suite size.

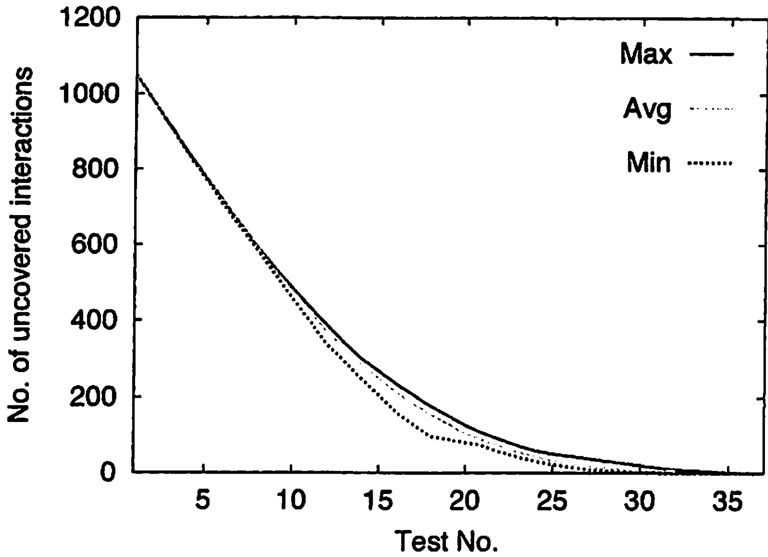
In our experiments, random suites are only competitive when one compares an interaction test suite produced with no concern for early coverage to a random test suite with the same number of tests, and when one runs all tests in each. As in Table 2, the performance of random test suites with respect to expected time to fault detection is not competitive, and so we focus on algorithmic techniques to improve the rate of coverage.

2.3. **Early Coverage Does Not Ensure Minimum Test Suite Size.** Despite having substantially more tests, the faster rate of coverage by the greedy test suite in Section 2.1 reduces time to fault detection. Smaller test suite sizes do not ensure earlier coverage of interactions, even when the test suites can be reordered arbitrarily. Nevertheless, there may be another 'small' test suite that does provide most coverage early.

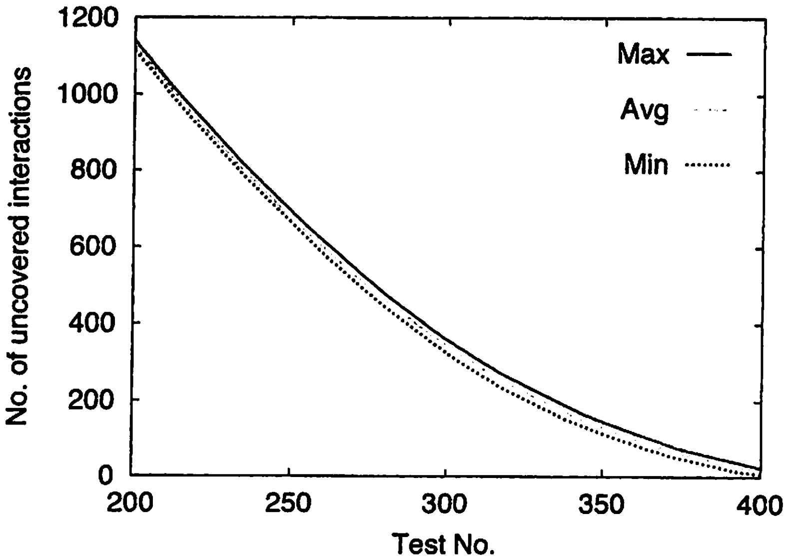
To explore this, we add tests one at a time, so that the next test maximizes the number of newly covered interactions. For type 2^8 of strength 4, we enumerate the 2^8 possible tests and repeatedly select one that covers the largest number of uncovered 4-way interactions. We break ties uniformly at random. The construction of a test suite is repeated 100 times. Figure 2 (a) shows the minimum, average, and maximum number of 4-way interactions left uncovered after each test. The smallest test suite for this type and 4-way coverage has 24 tests [16, 42]. However, the ultimate size of the test suites from the one-test-at-a-time method vary from 31 to 37.

For type 4^6, the first 88 tests cover the same minimum, average, and maximum number of 4-way interactions. After this, the results differ. The ultimate sizes range from 408 to 425 tests. But the smallest known suite has only 340 tests [16]. Hence any one-test-at-a-time approach that maximizes the number of t-way interactions covered in each test need not produce the smallest test suite.

2.4. **Minimum Test Suite Size and Early Coverage are Incompatible.** The preliminary comparisons just made guarantee that our primary



(a) 2^8



(b) 4^6

FIGURE 2. The minimum, average, and maximum rate of 4-way interaction coverage from a one-test-at-a-time exhaustive algorithm.

objectives of test suite size and expected time to detect faults do not coincide. Now we demonstrate the stronger statement that they are incompatible. In the comparisons made, one could argue that few test suites are compared, and perhaps there is another that has no more tests and has a rate of coverage that is at least as good. To address the comparison of test suite size and early coverage definitively, we consider type 2^{11} of strength three. The minimum number of tests is 12 [16], but much more is known here. Up to renaming symbols and reordering rows and columns, there is only one way to choose 12 tests to cover all 3-way interactions. Ordering this test suite to minimize expected time to fault detection, we find that $(\tau_1, \ldots, \tau_{12}) = (165, 155, 145, 135, 125, 115, 105, 95, 85, 75, 65, 55)$; this is the 'Fewest Tests' suite. However, it does not provide the best coverage after few tests are performed. We exhaustively tried every way to select, at each stage, a test that maximizes the number of additional 3-way interactions covered. This necessitates, no matter what selection is made, that at least eighteen tests be chosen in total. Indeed with these requirements we found a test suite with $(\tau_1, \ldots, \tau_{18})$ = (165, 165, 145, 145, 103, 102, 96, 90, 66, 64, 47, 43, 29, 23, 17, 13, 4, 3); this is the 'Most Greedy' suite. Despite requiring many more tests, it has covered more 3-way interactions after four tests than any 12-test suite can. Neither of these has the best expected time to detect a single fault. By a greedy approach that did not insist on the maximum number of newly covered interactions in each test, we found a 14-test solution with $(\tau_1, \ldots, \tau_{14}) = (165, 155, 145, 137, 131, 121, 111, 101, 76, 67, 37, 34, 32, 8)$; this is the 'Best Φ_1' suite.

Table 3 shows the values of Φ_s for $0 \leq s \leq 8$ determined by (2). It is striking that the solution with fewest tests does not lead to the best time for fault detection unless no faults are present. Moreover, optimizing the time to fault detection for a single fault does not provide the best expected time when many faults are present.

	Number of Faults Present								
Suite	0	1	2	3	4	5	6	7	8
Fewest Tests	12.0	5.42	3.55	2.71	2.25	1.95	1.75	1.60	1.49
Most Greedy	18.0	5.74	3.57	2.70	2.22	1.93	1.73	1.58	1.48
Best Φ_1	14.0	5.39	3.51	2.70	2.24	1.95	1.75	1.60	1.49

TABLE 3. Expected Time to Fault Detection for Three Test Suites for Type 2^{11} and Strength 3 when s Faults are present, $0 \leq s \leq 8$

It bears emphasis that in constructing these suites, no heuristics or random selections are used; all possibilities with 12 tests and all possibilities

obtained by selecting tests one by one that maximize additional coverage were examined. Indeed we have shown that minimum test suite size can require that one does *not* have the minimum expected time to fault detection. Consequently, improving rate of coverage is essentially different from generating small test suites. In the remainder of this paper, we focus on the problem of generating test suites that have early coverage of t-way interactions. We introduce a hybrid technique for this purpose.

3. A Hybrid Technique

Ultimately a test suite covers all t-way interactions. In order to minimize expected time to fault detection, our focus is on covering as many t-way interactions as possible in the earliest tests. The approach that we pursue is straightforward. We add one test at a time, making a greedy initial selection for the test and then iteratively improving it by heuristic search to increase the number of previously uncovered t-way interactions that it covers.

3.1. **Test initialization.** Existing one-test-at-a-time greedy algorithms for constructing test suites fall into a common framework [6]. For each test generated, M *candidate tests* are generated to select the test that covers the most t-way interactions. For the construction of a single test, *factor ordering* is the order in which factors are assigned levels. For each factor, a *level selection* rule specifies criteria for assigning a level to a factor.

Any one-test-at-a-time greedy algorithm may be used in our approach. We use a specific instantiation of the framework for experimentation. Only one candidate test is constructed each time. To begin, select a t-way interaction that has not yet been covered, and set the factors in this interaction to the corresponding levels. The remaining factors are ordered randomly. A factor that has been assigned a level is *fixed*; one that hasn't as *free*. For each factor in turn, the level that covers the largest number of previously uncovered t-way interactions in relation to fixed factors is selected. This algorithm is essentially that used by AETG [10].

3.2. **Test improvement by heuristic search.** We do not expect this naive greedy approach to produce a next test that maximizes the number of t-way interactions covered. Heuristic search has been applied to produce test suites of minimum size and often yields the smallest test suites, at the cost of higher execution times [40]. However, current heuristic search techniques do not consider the rate of coverage. Therefore, we apply heuristic search techniques instead to 'improve' the current test. As implemented

here, each has the same goal, to maximize the number of previously uncovered t-way interactions covered in a test. The heuristic search techniques examined for improving a given test are described next.

Hill climbing: Using hill climbing [32], for a test S, a factor is selected at random. A *move* is a change of the level of this factor to another level to form another test S'. The cost of a test is the number of t-way interactions still not covered if this test is incorporated in the test suite. If the cost of S' is no larger than the cost of S, then the move from S to S' is accepted (and we proceed using S' rather than S). After any number of iterations, the current test covers at least as many t-way interactions as the initial test.

Simulated Annealing: Simulated annealing operates similarly, but employs a more complicated acceptance criterion. To determine whether to accept a move, a global 'temperature' is maintained and adjusted downwards over time using a *cooling schedule.* A move from S to S' is always accepted when S' has cost no larger than S. When S' has cost larger than S, the move is accepted with a probability that is a function of the current temperature, so that lower temperature implies lower probability of acceptance. As more iterations are performed, it need not be the case that the current test is the best one encountered so far. Therefore, after the specified number of iterations are completed, the best test encountered in the search is selected. For our experiments to follow, we use an initial temperature of 10% of the total number of levels and the cooling schedule reduces by 1 degree for every 10% of the number of iterations specified. Fewer iterations mean faster cooling. See [12] for a more detailed discussion of simulated annealing for test suite generation.

Tabu search: This employs the same basic strategy. However, moves are accepted whether or not they increase cost, unless they are *tabu.* Generally, tabu moves are recorded on a *tabu list*; for our implementation, a tabu move is one that has occurred during the last T iterations, where T is the length of history maintained. A tabu list of length $T = 10$ is recommended in [31], however, in our implementation, tabu lists have size equal to 25% of the total number of levels. Again we report the best test encountered.

Great Flood: The Great Flood (or "Great Deluge") algorithm was introduced in [21]. This employs the same basic strategy; however, cost is now the number of covered t-way interactions. Moves are accepted exactly when the cost does not fall below a *rising threshold.* In our experiments, the rising threshold is 90% of the best solution encountered after the first iteration, and raises at a period of once every 10% of the number of iterations. The increases are to 95%, 98%, 99%, and 100%. We also incorporate

a tabu list of size 25% of the total number of levels. Again we report the best test encountered.

4. Experiments

Applying a heuristic search technique to improve upon a next test to add to a test suite can surely not result in the selection of a poorer test. However, the appropriate type of search to employ, and the number of iterations for which to run it, are by no means clear. Our experiments compare the four search techniques, and examine the effects of allowing each more iterations in which to make improvements.

The greedy algorithm described earlier initially generates each test. We allow each heuristic search technique 10, 100, or 1,000 iterations to improve the test. Each experiment is repeated 100 times and the average is reported. (This extends the preliminary work in [2], for which experiments were only run 5 times each. Nonetheless, we find similar results.)

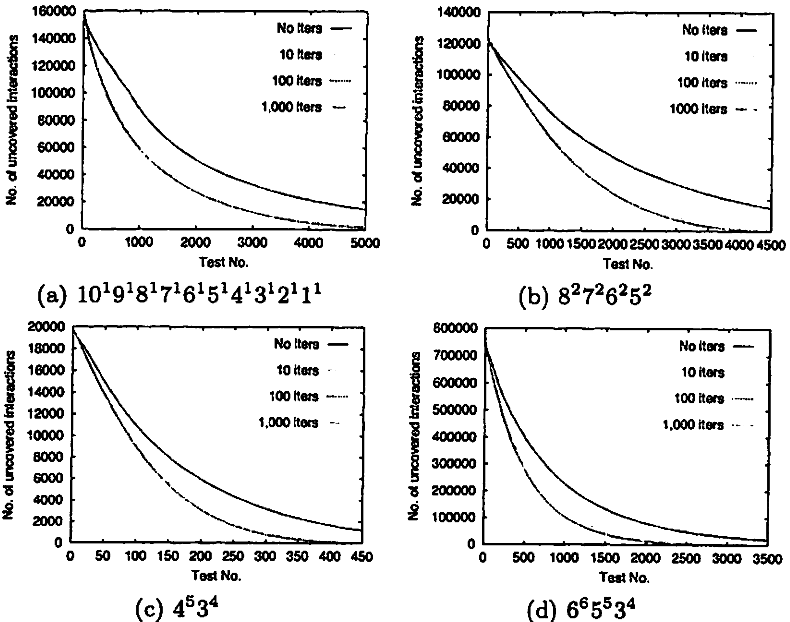
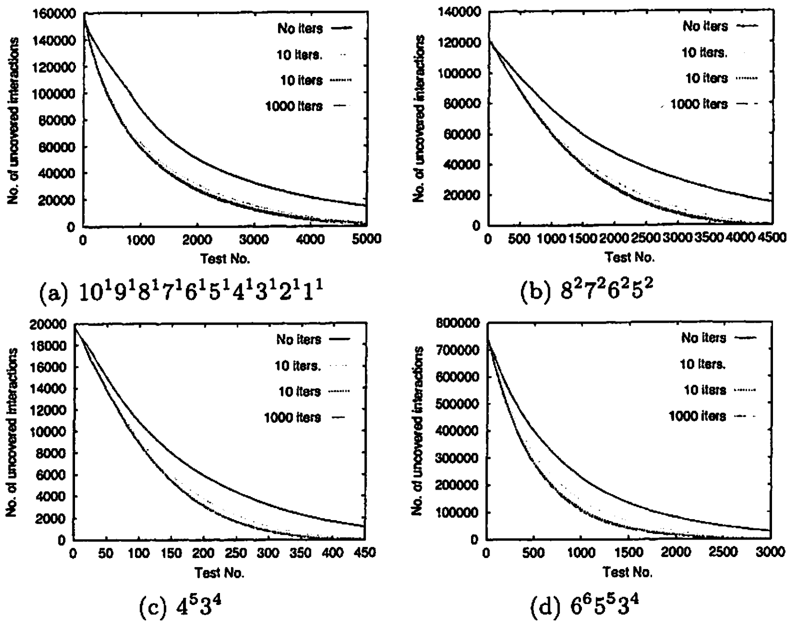


FIGURE 3. Rate of 4-way interaction coverage for four types using 0, 10, 100, and 1,000 iterations of Hill climbing.

We report experiments for types $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$, $8^2 7^2 6^2 5^2$, $4^5 3^4$, and $6^6 5^5 3^4$, each with strength 4. Figure 3 shows results for hill climbing;



(a) $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$ (b) $8^2 7^2 6^2 5^2$

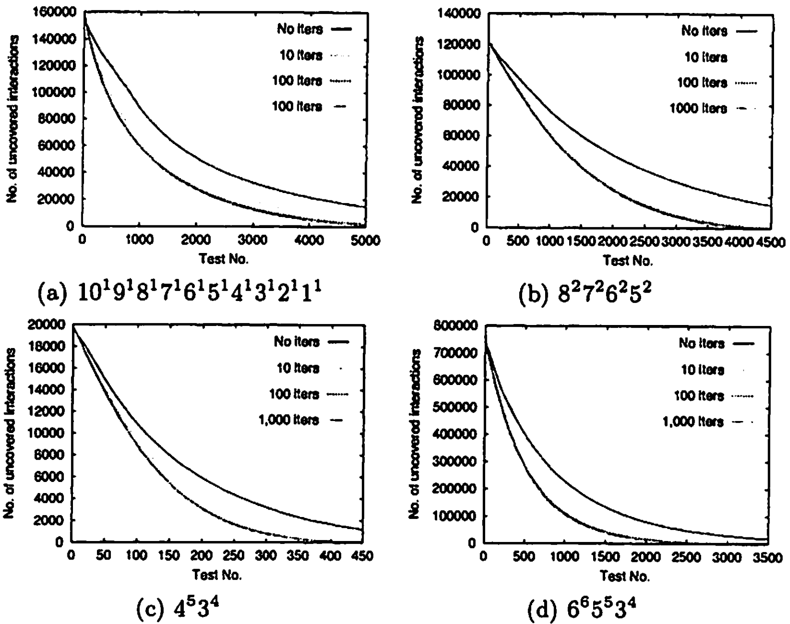
(c) $4^5 3^4$ (d) $6^6 5^5 3^4$

FIGURE 4. Rate of 4-way interaction coverage for four types using 0, 10, 100, and 1,000 iterations of Simulated Annealing.

Figure 4 for simulated annealing; Figure 5 for the great flood; and Figure 6 for tabu search.

For all four types and all four heuristic search methods, the rate of 4-way interaction coverage improves when the number of search iterations are increased from 0, to 10, to 100, to 1,000. Previous work [2] reports similar findings for types 3^{13} and 5^7.

As expected, the application of heuristic search improves the rate of 4-way interaction coverage over that of a greedy algorithm alone, and the rate of 4-way interaction coverage improves when the number of search iterations is increased. An iteration does not guarantee acceptance of a move, or improvement if the move is accepted. Figure 7 graphs the number of times that moves are accepted for each test using the four search techniques applied to type $5^1 3^8 2^2$. With 10 iterations, there are typically between 1 to 6 accepted moves per test. With 100 iterations and 1,000 iterations, for the first few tests there are many accepted moves, but this collapses after approximately 10 tests are selected, and then picks up during the remainder.



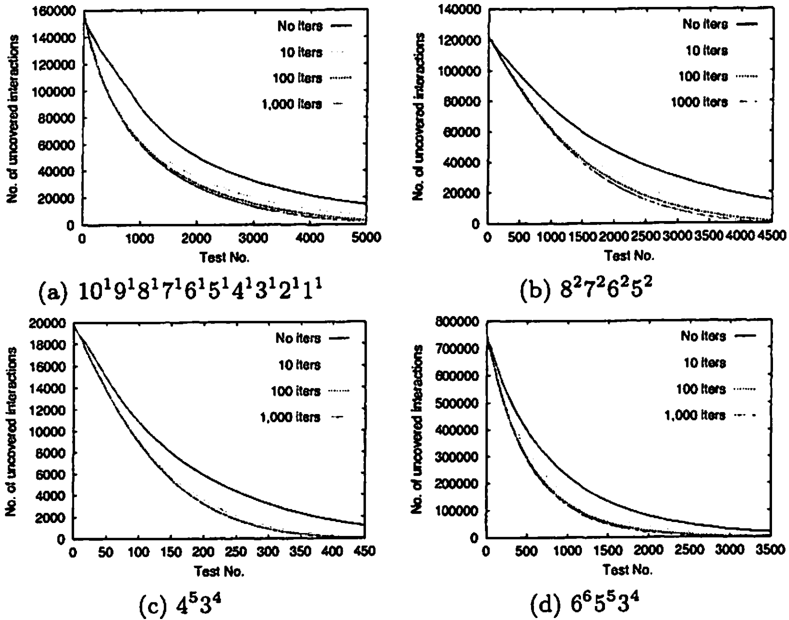
(a) $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$ (b) $8^2 7^2 6^2 5^2$

(c) $4^5 3^4$ (d) $6^6 5^5 3^4$

FIGURE 5. Rate of 4-way interaction coverage for four types using 0, 10, 100, and 1,000 iterations of Flood.

Each search technique improves on the greedy result, but each improvement costs execution time. Table 4 shows the average time in seconds to generate single tests for the experiments run on a SunBlade 5000 machine. To compute the average time per test, we generate full test suites and divide the time by the number of tests. Having amortized the initialization time across all tests, the time to generate each individual test is impacted in a small way. However, the initialization time is relatively small.

4.1. Rate of Coverage for Hill Climbing, Flood, Tabu, and Simulated Annealing. In Section 2, we saw that a minimum size test suite need not correspond to a fastest rate of t-way interaction coverage. Consequently we developed a hybrid greedy/heuristic search strategy to focus on a fast rate of coverage for interactions, and demonstrated that different heuristic search methods enable us to improve individual tests selected, especially if the method is permitted an adequate number of iterations to improve the test selected. Which heuristic search technique yields the fastest rate of t-way interaction coverage?



(a) $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$

(b) $8^2 7^2 6^2 5^2$

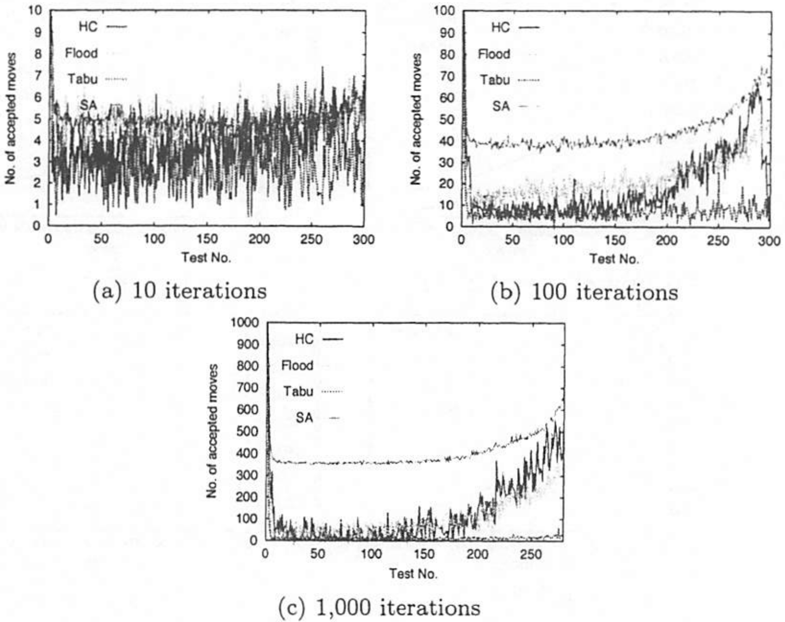
(c) $4^5 3^4$

(d) $6^6 5^5 3^4$

FIGURE 6. Rate of 4-way interaction coverage for four types using 0, 10, 100, and 1,000 iterations of Tabu.

To assess the rate of fault detection, we consider the values of Φ_s, the expected time to find an interaction fault when s random faults are present, for $1 \leq s \leq 3$. Table 5 shows the results for four types, with 10, 100, and 1000 search iterations. In each case, the same statistics are reported for each of the four search methods, and for using no search method but rather simply accepting the test that the greedy algorithm generates.

Hill-climbing consistently yields the best overall results when 10 or 100 search iterations are performed; however, with 1000 iterations, simulated annealing makes a more substantial improvement in each case, and great flood improves on hill-climbing for three of the four types. For these numbers of iterations, tabu search does not appear to be competitive. It is perhaps surprising that the least sophisticated search technique performs well, but this can be attributed to the relatively small number of iterations performed. At 1000 iterations, hill-climbing appears to have arrived at a plateau, while the other three continue to improve. Nevertheless, we think it is unlikely that substantial additional investment would be worthwhile in improving individual tests; with more time, a better investment of resources would be to focus on the selection of the test upon which to improve.



(a) 10 iterations

(b) 100 iterations

(c) 1,000 iterations

FIGURE 7. Number of accepted moves per test using 10, 100, and 1,000 iterations of Hill Climbing, Flood, Tabu, and Simulated Annealing for type $5^1 3^8 2^2$.

4.2. **Using the Density Algorithm.** It could be argued that the improvements obtained in Table 5 indicate that the greedy method used is making poor initial selections for tests in some cases, leaving much room for improvement. Therefore we also employed a different greedy method. The density algorithm is a one-test-at-a-time greedy algorithm that appears to produce smaller size test suites than the greedy algorithm here [4, 5]. Figure 8 shows that the heuristic search techniques also improve the rate of t-way interaction coverage when tests are initialized with the density algorithm. The density algorithm always selects a next test that covers at least the average number of previously uncovered t-way interactions, and consequently appears to produce a faster rate of t-way interaction coverage than the greedy method used for Table 5. Nevertheless the same pattern repeats here, as shown in Figure 8.

5. CONCLUSIONS

Software interaction testing provides a means to systematically test combinations of parameters to a system. The higher the strength of interaction

TABLE 4. Execution time per test (in seconds) using hill climbing, simulated annealing, tabu search, and great flood with 10, 100, and 1,000 search iterations.

	No. of iters.	HC	SA	Tabu	Flood
$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$	10	0.03	0.02	0.02	0.02
$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$	100	0.14	0.05	0.03	0.03
$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$	1000	1.29	0.25	0.16	0.16
$8^2 7^2 6^2 5^2$	10	0.01	0.01	0.01	0.01
$8^2 7^2 6^2 5^2$	100	0.06	0.02	0.01	0.01
$8^2 7^2 6^2 5^2$	1000	0.50	0.09	0.06	0.06
$6^6 5^5 3^4$	10	0.22	0.22	0.14	0.15
$6^6 5^5 3^4$	100	0.95	0.36	0.22	0.27
$6^6 5^5 3^4$	1000	8.26	1.85	1.13	1.33
$4^5 3^4$	10	0.01	0.01	0.01	0.00
$4^5 3^4$	100	0.04	0.02	0.01	0.01
$4^5 3^4$	1000	0.41	0.12	0.08	0.09

coverage, the closer the testing is to exhaustive. Previous work focuses on minimizing test suite size. In this paper, we differentiate between two different goals for test generation: constructing a test suite of smallest size versus covering as many t-way interactions as possible in the earliest tests. We established that these are not just different goals – they are incompatible goals. We identified two settings in which early coverage is important, when faults are present and are to be detected as quickly as possible, and when time constraints on executing a test suite may prevent the entire suite from being completed. We demonstrated that simply selecting tests at random, while fast, does not meet either goal well. Therefore we developed a hybrid approach to generate tests in order so that the expected time to fault detection is reduced. The hybrid approach uses a fast but effective greedy method to produce an initial candidate for a test, which it then modifies repeatedly to improve the coverage provided by that test. We experimented with four heuristic search techniques for modifying tests, and found that a simple hill-climbing technique is effective. Each of the heuristic search methods appears to be worthwhile when early coverage is desirable.

While there is a substantial mathematical and experimental literature on minimizing test suite size, little is known about minimizing expected time

TABLE 5. Expected Times to Fault Detection

Type $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$									
	10 iterations			100 iterations			1000 iterations		
Search	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
none	1901.8	886.7	568.1	1901.8	886.7	568.1	1901.8	886.7	568.1
HC	1129.3	514.0	319.2	1062.2	490.0	305.9	1062.2	489.8	305.7
Flood	1272.3	563.4	345.8	1089.6	499.6	311.0	1059.6	490.2	306.3
Tabu	1329.1	585.6	358.7	1155.7	521.4	322.0	1091.1	501.4	312.1
SA	1158.7	531.5	329.2	1081.4	496.9	309.5	1052.8	487.3	304.6
Type $8^2 7^2 6^2 5^2$									
	10 iterations			100 iterations			1000 iterations		
Search	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
none	2068.9	1065.3	713.8	2068.9	1065.3	713.8	2068.9	1065.3	713.8
HC	1259.0	705.0	492.0	1203.2	684.8	481.2	1202.8	684.5	481.0
Flood	1396.4	756.1	519.6	1228.0	694.9	486.9	1202.7	685.4	481.7
Tabu	1459.9	778.5	531.3	1300.0	718.5	498.5	1231.1	694.4	486.1
SA	1308.9	728.0	504.4	1220.7	691.9	485.1	1193.4	681.8	479.8
Type $6^6 5^5 3^4$									
	10 iterations			100 iterations			1000 iterations		
Search	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
none	870.4	411.7	267.2	870.4	411.7	267.2	870.4	411.7	267.2
HC	548.8	270.8	179.2	503.8	254.8	170.4	503.1	254.5	170.3
Flood	608.3	292.3	191.3	521.1	261.0	173.8	505.5	256.0	171.2
Tabu	636.5	303.1	197.2	557.4	272.3	179.5	534.0	264.2	174.8
SA	597.6	291.8	190.9	519.1	260.0	173.0	502.1	254.3	170.2
Type $4^5 3^4$									
	10 iterations			100 iterations			1000 iterations		
Search	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
none	162.7	85.3	58.3	162.7	85.3	58.3	162.7	85.3	58.3
HC	114.3	63.8	44.7	109.6	62.1	43.8	109.7	62.2	43.8
Flood	123.4	67.1	46.4	110.8	62.6	44.1	109.1	62.0	43.8
Tabu	116.3	64.5	45.0	111.2	62.7	44.1	111.5	62.8	44.1
SA	118.1	65.5	45.6	110.3	62.4	44.0	109.1	62.0	43.8

to fault detection. In part, this is a result of the apparently reasonable, but incorrect, belief that minimum test suite size is the correct objective. A test suite intended for validation prior to release anticipates that no faults remain, and then test suite size is a sensible metric; but a test suite intended for screening, when some faults are anticipated, requires a different metric for their evaluation. Our results explain, in part, the apparent usefulness

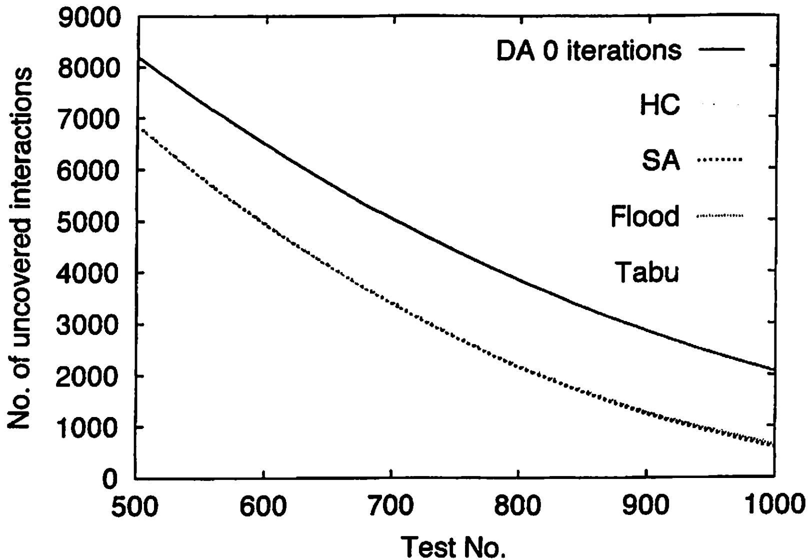


FIGURE 8. Rate of 4-way interaction coverage for type 5^7 using the Density Algorithm and four search techniques, each employing 1000 search iterations. Results are reported from 500 to 1000 tests.

of one-test-at-a-time greedy techniques, because they are fundamentally concerned with the rate of coverage rather than test suite size.

6. ACKNOWLEDGEMENTS

This work is supported in part by the National Institute of Standards and Technology, Information and Technology Lab Award under award number 70NANB10H048. Any opinions, findings and conclusions expressed herein are the authors and do not reflect those of the sponsors. This submission is an extension of work from [2] and our talk from the Twenty-Fifth Midwest Conference on Combinatorics, Cryptography, and Computing.

REFERENCES

[1] R. C. Bryce, Y. Chen, C. J. Colbourn, Biased covering arrays for progressive ranking and composition of web services, International Journal of Simulation and Process Modeling 3 (2007) 80–87.

[2] R. C. Bryce, C. J. Colbourn, One-test-at-a-time heuristic search for interaction test suites, in: Genetic and Evolutionary Computation Conference (GECCO), Search-based Software Engineering track (SBSE), (2007), pp. 1082–1089.
[3] R. C. Bryce, C. J. Colbourn, Prioritized interaction testing for pairwise coverage with seeding and avoids, Information and Software Technology Journal 48 (2006) 960–970.
[4] R. C. Bryce, C. J. Colbourn, The density algorithm for pairwise interaction testing, Software Testing, Verification, and Reliability 17 (2007) 159–182.
[5] R. C. Bryce, C. J. Colbourn, A density-based greedy algorithm for higher strength covering arrays, Software Testing, Verification, and Reliability 19 (2009) 37–53.
[6] R. C. Bryce, C. J. Colbourn, M. B. Cohen, A framework of greedy methods for constructing interaction tests, in: Proceedings of the 27th International Conference on Software Engineering (ICSE), IEEE, Los Alamitos, CA, (2005), pp. 146–155.
[7] K. Burr, W. Young, Combinatorial test techniques: Table-based automation, test generation, and code coverage, in: Proceedings of the International Conference on Software Testing Analysis and Review, ACM, New York, (1998), pp. 503–513.
[8] C. Cheng, A. Dumitrescu, P. Schroeder, Generating small combinatorial test suites to cover input-output relationships, in: Proceedings of the Third International Conference on Quality Software (QSIC '03), IEEE, Los Alamitos, CA, (2003), pp. 76–82.
[9] D. M. Cohen, S. R. Dalal, M. L. Fredman, G. C. Patton, The AETG system: An approach to testing based on combinatorial design, IEEE Transactions on Software Engineering 23 (1997) 437–44.
[10] D. M. Cohen, S. R. Dalal, J. Parelius, G. C. Patton, The combinatorial design approach to automatic test generation, IEEE Software 13 (1996) 82–88.
[11] M. B. Cohen, Designing test suites for software interaction testing, Ph.D. thesis, The University of Auckland, Department of Computer Science (2004).
[12] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, W. B. Mugridge, Constructing test suites for interaction testing, in: Proc. Intl. Conf. on Software Engineering (ICSE 2003), IEEE, Los Alamitos, CA, (2003), pp. 38–48.
[13] M. B. Cohen, C. J. Colbourn, A. C. H. Ling, Constructing strength three covering arrays with augmented annealing, Discrete Math. 308 (2008) 2709–2722.
[14] M. B. Cohen, M. B. Dwyer, J. Shi, Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach, IEEE Transactions on Software Engineering 34 (2008) 633–650.
[15] C. J. Colbourn, Combinatorial aspects of covering arrays, Le Matematiche (Catania) 58 (2004) 121–167.
[16] C. J. Colbourn, G. Kéri, P. P. Rivas Soriano, J.-C. Schlage-Puchta, Covering and radius-covering arrays: Constructions and classification, Discrete Applied Mathematics, 158 (2010), 1158-1190.
[17] C. J. Colbourn, S. S. Martirosyan, G. L. Mullen, D. E. Shasha, G. B. Sherwood, J. L. Yucas, Products of mixed covering arrays of strength two, J. Combin. Des. 14 (2006) 124–138.
[18] C. J. Colbourn, S. S. Martirosyan, Tran Van Trung, R. A. Walker II, Roux-type constructions for covering arrays of strengths three and four, Des. Codes Cryptogr. 41 (2006) 33–57.
[19] S. R. Dalal, A. J. N. Karunanithi, J. M. L. Leaton, G. C. P. Patton, B. M. Horowitz, Model-based testing in practice, in: Proc. Intl. Conf. on SoftwareEngineering,(ICSE '99), (1999), pp. 285–294.
[20] S. R. Dalal, C. L. Mallows, Factor-covering designs for testing software, Technometrics 40 (1998) 234–243.

[21] G. Dueck, New optimization heuristics - the great deluge algorithm and the record-to-record travel, Journal of Computational Physics 104 (1993) 86–92.
[22] S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, A. Iannino, Applying design of experiments to software testing, in: Proc. Intl. Conf. on Software Engineering (ICSE '97), IEEE, Los Alamitos, CA, (1997), pp. 205–215.
[23] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, D. R. Kuhn, Refining the in-parameter-order strategy for constructing covering arrays, J. Res. Nat. Inst. Stand. Tech. 113 (2008) 287–297.
[24] M. Grindal, J. Offutt, S. F. Andler, Combination testing strategies – a survey, Software Testing, Verification, and Reliability 5 (2005) 167–199.
[25] A. Hartman, Software and hardware testing using combinatorial covering suites, in: M. C. Golumbic, I. B.-A. Hartman (Eds.), Interdisciplinary Applications of Graph Theory, Combinatorics, and Algorithms, Springer, Norwell, MA, (2005), pp. 237–266.
[26] A. Hartman, L. Raskin, Problems and algorithms for covering arrays, Discrete Math. 284 (2004) 149–156.
[27] B. Hnich, S. Prestwich, E. Selensky, B. M. Smith, Constraint models for the covering test problem, Constraints 11 (2006) 199–219.
[28] D. Kuhn, M. Reilly, An investigation of the applicability of design of experiments to software testing, in: Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop, IEEE, Los Alamitos, CA, (2002), pp. 91–95.
[29] D. R. Kuhn, D. R. Wallace, A. M. Gallo, Software fault interactions and implications for software testing, IEEE Trans. Software Engineering 30 (2004) 418–421.
[30] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, J. Lawrence, IPOG/IPOD: Efficient test generation for multi-way software testing, Software Testing, Verification, and Reliability 18 (2008) 125–148.
[31] K. Nurmela, Upper bounds for covering arrays by tabu search, Discrete Applied Mathematics 138 (2004) 143–152.
[32] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice-Hall, Chapter 4, (1995).
[33] P. J. Schroeder, P. Bolaki, V. Gopu, Comparing the fault detection effectiveness of n-way and random test suites, in: Proc. International Symposium on Empirical Software Engineering (ISESE04), (2004), pp. 49–59.
[34] T. Shiba, T. Tsuchiya, T. Kikuno, Using artificial life techniques to generate test cases for combinatorial testing, in: Intl. Conference on Computer Software and Applications Conference (COMPSAC), (2004), pp. 72–77.
[35] N. J. A. Sloane, Covering arrays and intersecting codes, J. Combin. Des. 1 (1993) 51–63.
[36] J. Stardom, Metaheuristics and the search for covering and packing arrays, Master's thesis, Simon Fraser University (2001).
[37] B. Stevens, Transversal covers and packings, Ph.D. thesis, Mathematics, University of Toronto (1998).
[38] K. C. Tai, L. Yu, A test generation strategy for pairwise testing, IEEE Transactions on Software Engineering 28 (2002) 109–111.
[39] Y. W. Tung, W. S. Aldiwan, Automating test case generation for the new generation mission software system, in: Proc. 30th IEEE Aerospace Conference, IEEE, Los Alamitos, CA, (2000), pp. 431–437.
[40] R. C. Turban, Algorithms for covering arrays, Ph.D. thesis, Arizona State University (May 2006).

[41] A. W. Williams, Determination of test configurations for pair-wise interaction coverage, in: Testing of Communicating Systems: Tools and Techniques, Kluwer, Boston Dordrecht London, (2000), pp. 59–74.

[42] J. Yan, J. Zhang, A backtracking search tool for constructing combinatorial test suites, J. Systems Software 81 (2008) 1681–1693.

[43] C. Yilmaz, M. B. Cohen, A. Porter, Covering arrays for efficient fault characterization in complex configuration spaces, in: Intl. Symp. on Software Testing and Analysis, IEEE, Los Alamitos, CA, (2004), pp. 45–54.

UNIVERSITY OF NORTH TEXAS, 1155 UNION CIRCLE #311366, DENTON, TX 76203

E-mail address: Renee.Bryce@unt.edu

SCHOOL OF COMPUTING, INFORMATICS, AND DECISION SYSTEMS ENGINEERING, ARIZONA STATE UNIVERSITY, TEMPE AZ 85287-8809, U.S.A. *and* STATE KEY LABORATORY OF SOFTWARE DEVELOPMENT ENVIRONMENT, BEIHANG UNIVERSITY, BEIJING 100191, CHINA

E-mail address: Charles.Colbourn@asu.edu