Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 48 (2006) 960–970
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Prioritized interaction testing for pair-wise coverage with seeding and constraints ☆

Renée C. Bryce [a,*], Charles J. Colbourn [b]

[a] Department of Computer Science, University of Nevada at Las Vegas, Las Vegas, NV 89154-4019, USA
[b] Computer Science and Engineering, Arizona State University Tempe, AZ 85287-8809, USA

Received 27 February 2006; accepted 21 March 2006
Available online 4 May 2006

Abstract

Interaction testing is widely used in screening for faults. In software testing, it provides a natural mechanism for testing systems to be deployed on a variety of hardware and software configurations. In many applications where interaction testing is needed, the entire test suite is not run as a result of time or budget constraints. In these situations, it is essential to *prioritize* the tests. Here, we adapt a "one-test-at-a-time" greedy method to take importance of pairs into account. The method can be used to generate a set of tests in order, so that when run to completion all pair-wise interactions are tested, but when terminated after any intermediate number of tests, those deemed most important are tested. In addition, practical concerns of seeding and avoids are addressed. Computational results are reported.

Keywords: Biased covering arrays; Covering arrays; Greedy algorithm; Mixed-level covering arrays; Pair-wise interaction coverage; Software interaction testing; Test prioritization

1. Introduction

Software testing is an expensive and time-consuming activity that is often restricted by limited project budgets. Accordingly, the National Institute for Standards and Technology (NIST) reports that software defects cost the U.S. economy close to $60 billion a year [21]. They suggest that approximately $22 billion can be saved through more effective testing. There is a need for advanced software testing techniques that offer an effective cost-benefit ratio in identifying defects. Interaction testing may offer a benefit when used to complement current testing methods. For instance, interaction testing has been applied in numerous instances [1,10,11,18,19,27]. Interaction testing implements a model-based testing approach using combinatorial design. In this approach, all t-tuples of interactions in a system are incorporated into a test suite. Indeed, interaction testing measures interactions versus detecting interactions; the goal is to cover pairs rather than avoid aliasing [17]. Higher-order interactions are not treated in this paper.

Our research extends the software interaction testing paradigm to provide both coverage of interactions, and also prioritization of the order in which specific interactions are covered. A test case prioritization problem measures tests based on the value of the interactions that they cover. When testers specify that certain interactions are of higher priority, it is desirable to include them in the earliest tests of a test suite.

A test case prioritization problem is formally defined in [22]. Given:

- T, a test suite,
- $\prod$, the set of all test suites obtained by permuting the tests of T, and
- f, a function from $\prod$ to the real numbers.

Problem: Find $\pi \in \prod$ such that $\forall \pi' \in \prod, f(\pi) \geqslant f(\pi')$.

☆ A preliminary version of this paper appears in [3].

* Corresponding author. Tel.: +1 480 965 8267; fax: +1 480 965 2751.
E-mail addresses: reneebryce@cs.unlv.edu (R.C. Bryce), colbourn@asu.edu (C.J. Colbourn).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.03.004

In this definition, the possible prioritizations of T are referred to as $\prod$ and f is a function that is applied to evaluate the orderings.

The selection of the function f leads to many criteria to prioritize software tests. For instance, prioritization may be based upon code coverage, cost estimate (i.e., how much execution time a test incurs), areas of recent changes, areas that testers believe to be particularly fault-prone, and others [12,13,22]. Prioritization may even be based on user profiles, operational profiles, or specification-based usage modeling to generate stochastic models of sequences [25,26]. In any case, prioritization techniques strive to increase a test suite's *rate of fault detection* by trying to identify faults early in the testing process. When defects are found sooner, there is more time for developers to fix them.

Prioritization for interaction testing can be addressed in two ways. As above, an existing test suite can be reordered to reflect the prioritization criteria. Alternatively, prioritization can be incorporated into initial generation. We pursue the second option. However, we first review prior work in both directions.

Empirical studies have shown that reordering an existing test suite can be beneficial. Elbaum et al. [12] present empirical results of the effectiveness of six prioritization techniques. Rothermel et al. [22] also study several prioritization techniques and provide empirical studies that suggest that while some prioritization techniques may perform better than others, even their least sophisticated technique improved the rate of fault detection. Srivastava and Thiagarajan [23] apply prioritization based on changes that have been made to a system for regression testing; they are able to expose defects early. Elbaum et al. [12] suggest which prioritization techniques may work best in certain scenarios.

It is also possible to generate initial test suites to take prioritization into account. Markov chain usage models to do this have been useful in software testing [25,26].

We propose a method for constructing a prioritized test suite. We do not propose a new prioritization criterion; rather, a tester can apply the prioritization criteria of their choice to derive weights (priorities). These weights are then used as input to automatically generate a prioritized interaction test suite that covers as much weight as early as possible. This optimization is done locally for each test (row) so that if a tester stops after any given test, they should have executed an optimal number of highest priority interactions for their respective efforts. The test suite generated may be run to completion, or until available testing resources are exhausted.

Our essential observation is that in interaction testing, not all interactions are of equal importance or priority; indeed, some may be of crucial importance, some may be inconsequential, some may be deleterious, and some may be impossible to test. Our primary goal must be to include, early in the test suite, those interactions that are of the largest importance (and perhaps to omit those that are infeasible or undesirable).

In order to construct prioritized interaction test suites, biased covering arrays are proposed. This paper is organized as follows. Section 2 presents definitions and background. Section 3 discusses some issues arising in software interaction testing in practice. Section 4 describes a method for constructing biased covering arrays. Section 5 provides computational results.

2. Definitions and background

Consider a modular system in which components interact as shown in Table 1. There are three pieces of hardware; three operating systems; three network connections; and three memory configurations that can be combined to construct a system. The four components are referred to as *factors*, and the three options shown for each factor are called *levels*. Different end users may have different preferences and likely use different combinations of components. Each configuration adds to the cost of testing. Exhaustive testing of all combinations for the input shown in Table 1 requires $3^4 = 81$ combinations.

Instead of testing every combination, pair-wise interaction testing requires that every individual pair of interactions is included at least once in a test suite. A pair-wise interaction test suite for this example is shown in Table 2, and has only nine tests.

This reduction in tests amplifies on larger systems – a system with 20 factors and five levels each would require $5^{20} = 95{,}367{,}431{,}640{,}625$ exhaustive tests! Pair-wise interaction testing for 5^{20} can be achieved in 45 tests.

The example presented is represented behind the scenes with a combinatorial object called a covering array.

Table 1
A component-based system with four factors that each have three settings

Hardware	Operating system	Network connection	Memory (MB)
PC	Windows XP	Dial-up	64
Laptop	Linux	DSL	128
PDA	Solaris	Cable	256

Table 2
A pair-wise interaction test suite

Test No.	Hardware	Operating system	Network connection	Memory (MB)
1	PC	Windows XP	Dial-up	64
2	PC	Linux	DSL	128
3	PC	Solaris	Cable	256
4	Laptop	Windows XP	Cable	128
5	Laptop	Linux	Dial-up	256
6	Laptop	Solaris	DSL	64
7	PDA	Windows XP	DSL	256
8	PDA	Linux	Cable	64
9	PDA	Solaris	Dial-up	128

A *covering array*, $CA(N;t,k,v)$, is an array with N rows and k columns that satisfies the criteria that each t-tuple occurs at least once within these rows. In our application, k is the number of factors, v is the number of symbols associated with each factor, and t is the *strength* of the coverage of interactions. For instance, in the previous example shown in Tables 1 and 2, $k = 4$, $v = 3$, and $t = 2$.

This combinatorial object is fundamental when all factors have an equal number of levels. However, software systems are typically not composed of components (*factors*) that each have exactly the same number of parameters (*levels*). Then, the mixed-level covering array can be used.

A *mixed-level covering array*, $MCA(N;t,k,(v_1,\ldots,v_k))$, is an $N \times k$ array. Let $\{i_1,\ldots,i_t\} \subseteq \{1,\ldots,k\}$, and consider the subarray of size $N \times t$ obtained by selecting columns $i_1,\ldots,i_t$ of the MCA. There are $\prod_{i=1}^{t} v_i$ distinct t-tuples that could appear as rows, and an MCA requires that each appear at least once.

In practice, a tester may feel that some interactions are more important to cover than are others; the covering array does not distinguish this, as it assumes that all rows are to be run as tests and have equal priorities. Therefore, a covering array is not a sufficient model when a tester prefers to test certain interactions earlier than others.

From this point onward, we model the priorities in a simple manner for the special case of $t = 2$ (pair-wise coverage). Call the factors $f_1,\ldots,f_k$. Suppose that, for each i, f_i has ℓ_i possible values $\tau_{i,1},\ldots,\tau_{i,\ell_i}$. For each $\tau_{i,j}$, we assume that a numerical value $-1.0 \leqslant \omega \leqslant 1.0$ is available as a measure of importance, with -1.0 as least important and 1.0 as most important. Every value τ for f_i has an importance $\omega_{i,\tau}$. A *test* consists of an assignment to each factor f_i of a value τ_i with $1 \leqslant i \leqslant \ell_i$. The importance of choosing τ_i for f_i and τ_j for f_j together is $\omega_{i,\tau_i}\omega_{j,\tau_j}$.

The *benefit* of a test (in isolation) is $\sum_{i=1}^{k}\sum_{j=i+1}^{k}\omega_{i,\tau_i}\omega_{j,\tau_j}$.

Every pair covered by the test contributes to the total benefit, according to the importance of the selections for the two values. However, in general we are prepared to run many tests. Rather than adding the benefits of each test in the suite, we must account a benefit only when a pair of selections has not been treated in another test. Let us make this precise. Each of the pairs (τ_i,τ_j) covered in a test of the test suite may be covered for the first time by this test, or may have been covered by an earlier test as well. Its *incremental benefit* is $\omega_{i,\tau_i}\omega_{j,\tau_j}$ in the first case, and zero in the second. Then, the incremental benefit of the test is the sum of the incremental benefits of the pairs that it contains.

The total benefit of a test suite is the sum, over all tests in the suite, of the incremental benefit of the tests.

An *ℓ-biased covering array* is a covering array $CA(N;t,k,v)$ in which the first ℓ rows form tests whose total benefit is as large as possible. That is, no $CA(N';t,k,v)$ has ℓ rows that provide larger total benefit.

Although precise, this definition should be seen as a goal rather than a requirement. Finding an ℓ-biased covering array is NP-hard, even when all benefits for pairs are equal [9]. Worse yet, the value of ℓ is rarely known in advance. For these reasons, we use the term *biased covering array* to mean a covering array in which the tests are ordered, and for every ℓ, the first ℓ tests yield a large total coverage of weight.

Covering arrays have been extensively studied [8,15]. Specifically, combinatorial constructions appear in [14] and therein; heuristic search in [7] and therein; and greedy algorithms in [4,5] and therein. The main criteria for evaluation of these techniques have been the size of the test suites and the execution time. However, consideration has also been given to other practical concerns: flexibility in treating a variety of parameters; permitting *seeds*, or user specified tests; and blocking *avoids* that are not to be covered, to name a few. Recently, constructions that also provide *prioritized ordering* of tests [2] have been considered. The algorithm to be presented here constructs pair-wise covering arrays while covering the most important values early. This work may be extended to t-way coverage.

3. Practical considerations

The one-row-at-a-time greedy algorithm presented here addresses prioritization while maintaining a structure that easily accommodates seeding, hard constraints of combinations that cannot be included in any test, and soft constraints of combinations that are less desirable. We review these next.

3.1. User input

At a minimum, current tools that generate software interaction test suites require the number of factors (or components) and the number of levels (or options) associated with each factor. From this input, a test suite can be automatically generated to cover all t-tuples. The test suite is then used as a black box testing technique.

Prioritized software interaction testing follows the same framework; however, additional information on priorities is needed. In addition to the number of factors and their numbers of levels, the weight (or priority) must also be specified. These weights may be determined using any prioritization criterion.

In any case, the prioritization weight is scaled to a weight ω in the range $-1.0 \leqslant \omega \leqslant 1.0$. Since priorities can change during testing, our algorithm actually generates the next test with respect to the current priorities defined. The weights may be applied to individual t-tuples for smaller problems, but the effort to assign weights to each t-tuple can be prohibitive on larger problems. Therefore, for practical purposes, we allow users to specify as much or as little granularity as desired. A user may start with all t-tuples given equal priorities by default and then modify the weights of individual tuples as desired. Alternatively, weights can even be automatically computed if a tester wishes to assign weight values to individual factors or levels. In this case, weights are calculated by multiplying the (factor,value) weights between pairs.

Table 3
Weights of three web services having 15 options each with the first 10 options and reliability ratings

Service	Rain forecast WS rating	Temperature forecast WS rating	Wind forecast WS rating
0	0.86	0.99	0.86
1	0.78	0.99	0.95
2	0.84	0.96	0.91
3	0.57	0.98	0.87
4	0.91	0.97	0.91
5	0.78	0.97	0.96
6	0.78	0.99	0.89
7	0.76	0.98	0.88
8	0.68	0.98	0.91
9	0.68	0.99	0.86

For instance, consider the example of prioritized interaction testing applied to web services as presented in [2]. Web services technology permits dynamic composition of new services based on available services published on the Internet. This development paradigm offers a competitive market for service providers because web services published on the Internet can be searched, tested, and used remotely. As a result, for the same functional specification, many alternative (good and bad) services can be offered by different service providers for varying costs. A challenge exists to efficiently determine which stand-alone services are the most appropriate to combine into a composite service based on concerns such as functionalities, licensing costs, and known reliability. In [2], reliability is of main concern when evaluating weather prediction services. Potential services are identified and assigned priority values based on their reliability ratings (weights).

In this example, three types of services are considered for inclusion of a system that determines whether to launch a satellite at a specific location on a specified date and time. These include: (1) rain forecast, (2) temperature forecast, and (3) wind forecast. For each of these three components, 10 options of acceptable reliability are available, as shown in Table 3 with their respective reliability ratings. (See [2] for details on the computation of reliability rankings.) This input is used to generate a prioritized interaction test suite based on these reliability ratings.

This example assigned weights to individual levels. However, again, prioritization values can be assigned based on any criteria deemed to be important by testers. Prioritization values can also be applied using less granularity if a tester prefers to apply prioritization values to factors, or with more granularity by assigning prioritization values to individual tuples.

3.2. Seeds

Testers often have a collection of tests that are prescribed, for instance by a requirements specification; these are *seeds*. We recognize that testers run such tests, and avoid redundant coverage of interactions. We identify the t-way interactions that have already been covered in the seeds. The interaction test suite is then extended to complete coverage of all interactions.

Table 4
An interaction test suite with 11 seeded rows requires only four extra tests to cover all pair-wise interactions

Test No.	f_0	f_1	f_2	f_3
1	0	3	6	9
2	0	3	7	9
3	0	5	7	9
4	0	3	8	10
5	0	4	7	9
6	0	3	8	9
7	0	5	6	11
8	0	4	6	10
9	2	3	7	11
10	1	3	7	10
11	1	5	8	9
12	2	4	8	9
13	2	5	6	10
14	1	4	6	11
15	0	3	8	11

For instance, consider an input with four factors that each have three levels. An optimal solution for pair-wise coverage can be constructed in nine tests as presented earlier in Table 2. However, assume that 11 prescribed tests have already been run. Rather than appending nine tests that cover all pair-wise interactions, we instead append only four new tests that cover all previously uncovered pair-wise interactions. The expanded test suite is shown in Table 4.

3.3. Hard and soft constraints

A test environment often exhibits constraints on interactions. These constraints can be classified into three main types. A combination of factor values is a *hard constraint*, or *forbidden configuration*, if it cannot be included in any test. The weight of a forbidden configuration is $\omega = -1.0$. A combination may also be a *soft constraint* if it is an avoid or neutral combination. An *avoid* is a combination that is permitted in a test, but its inclusion is undesirable for some reason. An avoid has a negative weight, $-1.0 < \omega < 0.0$. Indeed, the magnitude of the negative weight chosen indicates the importance of avoiding the tuple, in the same way that the magnitude of a positive weight indicates the importance of including it. It is *neutral* if we have no reason to include it in a test and no reason to avoid it. Neutral combinations have a weight, $\omega = 0.0$. These three types of constraints appear not to have been clearly delineated in previous work.

3.3.1. Hard constraints: forbidden configurations

Certain combinations of interactions produce un-testable or invalid tests when combined. They always reduce the number of t-tuples to be covered. However, they may result in a smaller sized test suite, a larger sized test suite, or there may be no feasible solution at all. Fig. 1 shows examples of each of these cases.

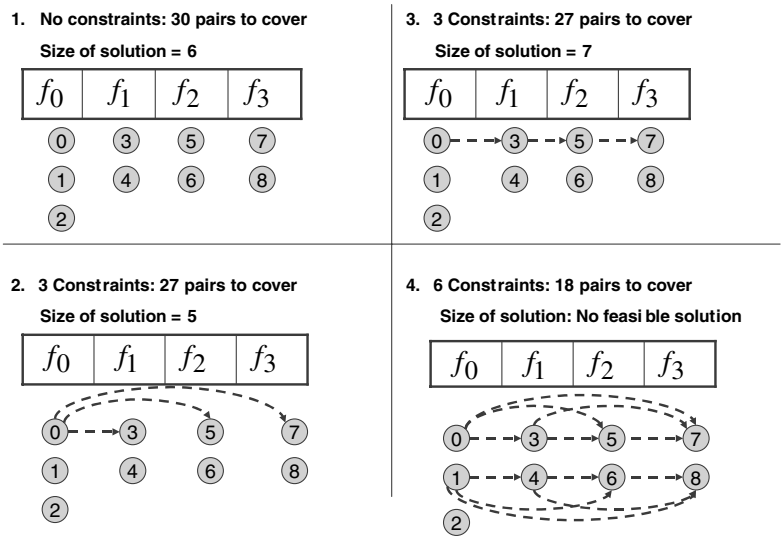

Fig. 1. Scenarios of constraints for input $3^1 2^3$ and pair-wise coverage.

The first example in Fig. 1 has no forbidden configurations (hard constraints) and there are 30 pairs to cover. A solution for pair-wise testing can be achieved in six tests. In the second example of Fig. 1, there are three forbidden configurations and only 27 pairs need to be covered. The reduction in the number of pairs to cover results in a reduction of test suite size to five tests. In the third example, there are also three forbidden configurations and only 27 pairs to cover. However, a larger test suite is needed to cover all pairs. Indeed, at least three rows must contain $(5, \neg 7)$; at least three rows must contain $(\neg 5, 7)$; and at least one row cannot contain either levels 5 or 7 for a total of at least seven rows. Finally, the 4th example includes 12 constraints and only 18 pairs to cover. However, there is no feasible solution at all. No matter which values are selected for f_1 and f_2, no valid selections for f_3 remain.

3.3.2. Soft constraints: avoids and neutral combinations

Avoids are combinations that are permitted but are undesirable in a test. Our algorithm negatively weights avoids so that if they are included, if at all, they only appear in the last tests of a test suite. Neutral interactions are those for which there is no preference whether the interaction is covered or not. This can arise, for example, when extending a seed; interactions covered in the seed are thereafter neutral. In Table 4, the combination $(8, 11)$ is specified as neutral and consequently is included only in the last test (since no benefit arises from its coverage).

3.3.3. The combinatorial requirements

In the setting of hard constraints, soft constraints, and seeds, our definition of biased covering arrays is insufficient. Suppose that S is a set of rows corresponding to seeded tests, A is a set of pairs to be avoided, and C is a set of pairs that cannot appear. Then an *ℓ-biased covering array* with respect to (S, A, C) is an $N \times k$ array in which all pairs not in $A \cup C$ appear in at least one row, no pair in C appears in a row, all rows in S are included in the array, and in which the first ℓ rows form tests whose total benefit is as large as possible. The array formed is not necessarily a covering array, since pairs in A need not be covered and pairs in C cannot be.

Only soft constraints (avoids and neutral positions) are addressed by our method. The reason is simple. In specific instances, it may not be possible to determine whether every pair (tuple) that is to be covered occurs in a test that contains no forbidden configuration, and this is a basic necessary condition for the construction of a test suite. This is a "constraint satisfaction" problem, and these are known to be NP-complete in general [20]. Even when we treat only pair-wise coverage and assume that every factor takes on only three values, it is NP-complete to decide whether there is even a single test that violates none of the constraints; to see this, consider a graph $G = (V, E)$. Treat vertices as factors; vertex $x \in V$ can take on one of three possible values, $\{x_0, x_1, x_2\}$. For each edge $e = \{x, y\} \in E$, the constraints are that factors x and y cannot be assigned the values x_i and y_i together, for $i \in \{0, 1, 2\}$. A test violating no constraints is equivalent in this way to a proper 3-vertex-colouring of G, and determining 3-colourability is a well-known NP-complete problem [24].

4. An algorithm for prioritization

We consider the construction of a test suite with k factors, adapting the deterministic density algorithm (DDA) [4,9]. We first describe DDA at a high level and then provide a modification of the algorithm that adapts to prioritization.

In the deterministic density algorithm (DDA), one row is constructed at a time and new rows are generated until all t-tuples are covered. Each factor is assigned a level value one-at-a-time. A factor that has been assigned a value is referred to as *fixed*; one that has not, as *free*. For each

factor, the level value that covers the largest density is selected. This density is calculated in one of three ways: (1) for each new tuple that is covered in relation to fixed factors, density increases by 1.0, or (2) when no new tuples are covered, density does not change, and (3) for each new partial tuple that may be covered with free factors, a density formula calculates the likelihood of covering future tuples.

Consider the latter case of calculating density, but for the special case of pair-wise coverage. The number of levels for factor i is denoted by ℓ_i and the maximum number of levels for any factor is denoted as $\ell_{\max}$. For factors i and j, the *local density* is $\delta_{i,j} = \frac{r_{i,j}}{\ell_{\max}}$ where $r_{i,j}$ is the number of uncovered pairs involving a value of factor i and a value of factor j. In essence, $\delta_{i,j}$ indicates the fraction of pairs of assignments to these factors which remain to be tested.

To modify DDA to account for prioritization, the density formula is modified. Instead of computing the ratio of uncovered pairs to be covered, the amount of weight to be covered is computed. For factors i and j, the *total benefit* $\beta_{i,j}$ is $\sum_{a=1}^{\ell_i}\sum_{b=1}^{\ell_j}\omega_{i,a}\omega_{j,b}$, while the *remaining benefit* ρ_{ij} is the same sum, but of the incremental benefits. Define the *local density* to be $\delta_{i,j} = \frac{\rho_{i,j}}{\beta_{ij}}$. In essence, $\delta_{i,j}$ indicates the fraction of benefit that remains available to accumulate in tests to be selected. We define the *global density* to be $\delta = \sum_{1\leqslant i<j\leqslant k}\delta_{i,j}$ Since the densities $\delta_{i,j}$ and $\delta_{j,i}$ are equal, we sum only those with $i<j$. At each stage, we endeavor to find a test whose incremental benefit is at least δ.

To select such a test, we repeatedly fix a value for each factor, and update the local and global density values. At each stage, some factors are *fixed* to a specific value, while others remain *free* to take on any of the possible values. When all factors are fixed, we have succeeded in choosing the next test. Otherwise, select a free factor f_s. We have $\delta = \sum_{1\leqslant i<j\leqslant k}\delta_{i,j}$ which we separate into two terms:

$$\delta = \sum_{\substack{1\leqslant i<j\leqslant k\\ i,j\neq s}}\delta_{i,j} + \sum_{\substack{1\leqslant i\leqslant k\\ i\neq s}}\delta_{i,s}.$$

Whatever level is selected for factor f_s, the first summation is not affected, so we focus on the second.

Write $\rho_{i,s,\sigma}$ for the ratio of the sum of incremental benefits of those pairs involving some level of factor f_i, and level σ of factor f_s to the sum of (usual) benefits of the same set of pairs. Then, rewrite the second summation as

$$\sum_{\substack{1\leqslant i\leqslant k\\ i\neq s}}\delta_{i,s} = \frac{1}{\ell_s}\sum_{\sigma=1}^{\ell_s}\sum_{\substack{1\leqslant i\leqslant k\\ i\neq s}}\rho_{i,s,\sigma}.$$

For prioritization, we choose σ to maximize $\sum_{\substack{1\leqslant i\leqslant k\\ i\neq s}}\rho_{i,s,\sigma}$. Then, $\sum_{\substack{1\leqslant i\leqslant k\\ i\neq s}}\rho_{i,s,\sigma} \geqslant \sum_{\substack{1\leqslant i\leqslant k\\ i\neq s}}\delta_{i,s}$. We then fix factor f_s to have value σ, set $v_s = 1$, and update the local densities setting $\delta_{i,s}$ to be $\rho_{i,s,\sigma}$. In plain terms, this means that we choose the level for a factor that offers the largest incremental benefit of density. In the process, the density has not been decreased (despite some possible – indeed necessary – decreases in some local densities).

We iterate this process until every factor is fixed. The factors could be fixed in *any order at all*, and the final test has density at least δ. Of course, it is possible to be greedy in the order in which factors are fixed. Fig. 2 provides pseudocode for this algorithm.

This method ensures that each test selected furnishes at least the *average* incremental benefit. This may seem to be a modest goal, and that one should instead select the test with maximum incremental benefit. However, even when all importance values are equal, it is NP-hard to select such a test (see [9]).

The run time for the algorithm is greatly influenced by the characterization of the input. For instance, more factors or more levels can exponentially increase the number of calculations required to compute density. An exact method to generate a solution would take $O(\ell^{Nk})$, whereas, our method is in line with the AETG method. In our algorithm, there are $O(Nk)$ times that a level is selected for a factor, $O(\ell)$ possibilities for a level value, and $O(k)$ computations to compare the possibilities, for total of $O(Nk^2\ell)$.

4.1. Algorithm Walk-through

In this section, we illustrate the algorithm with a small example of building a row for a test suite. Consider the input in Table 5 with three factors. The first factor (f_0) has four levels, the second (f_1) has three levels, and the third (f_2) has two levels. Each level value is labeled in Table 5 with a unique ID, and a weight for each in parentheses.

4.1.1. Step 1 – Identify the order to assign levels to factors

As mentioned earlier, factors may be assigned levels in any order at all. We instantiate a factor ordering base on

```
1.  start with empty test suite
2.  while uncovered pairs remain do
3.      for each factor
4.          compute factor interaction weights
5.          initialize a new test with all factors not fixed
6.          while a factor remains whose value is not fixed
7.              select such a factor f that has the largest factor interaction
8.                  weight, using a factor tie-breaking rule
9.              compute level interaction weights for each level of factor f
10.             select a level ℓ for f which offers the largest increase in
11.                 weighted density using a level tie-breaking rule
12.             fix factor f to level ℓ
13.         end while
14.     add test to test suite
15. end while
```

Fig. 2. Pseudocode for a greedy algorithm to generate biased covering arrays.

Table 5
Input of three factors and their levels and weights

Factor	v_0	v_1	v_2	v_3
f_0	0 (.2)	1 (.1)	2 (.1)	3 (.1)
f_1	4 (.2)	5 (.3)	6 (.3)	–
f_2	7 (.1)	8 (.9)	–	–

Table 6
Factor interaction weights

Factor interaction weight	f_0	f_1	f_2	Total weight
f_0	–	.4	.5	.9
f_1	.4	–	.8	1.2
f_2	.5	.8	–	1.3

weighted densities here. Factors are assigned values one at a time in order of decreasing factor interaction weights.

Factors that have been assigned values are *fixed* while those that have not been assigned values are called *free*. Factor interaction weights are calculated between two factors by multiplying the weights between all levels of the two factors. The maximum weight is denoted as $w_{\max}$.

Table 6 shows the factor interaction weights for the input from Table 5. The algorithm assigns levels for factors in decreasing order of their factor interaction weights: f_2, f_1, and f_0.

4.1.2. Step 2 – Assign levels for each factor

To select a value for f_2, either of its two levels τ_0 or τ_1 may be selected. The first level, τ_0, has a value of 7 in Table 5, while the second (τ_1) has a value of 8. There are two scenarios to calculate the weighted density for level selection. If a level is being selected in relation to a factor that has already been fixed, then the contributing weight of selecting the new level is 0.0 if no new weight is covered; otherwise the contributing weight is the product of the current level under evaluation's weight and the fixed level's weight. However, when selecting a level value in relation to free factors, weighted density is calculated for a level, ℓ, and a factor i that has a number of levels called $\tau_{\max}$, as: $\sum_{j=1}^{\tau_{\max}} \left(\frac{wf_{i_j} * w\ell}{w_{\max}} \right)$. We refer to this calculation as *factor-level interaction* weight.

For instance, the factor-level interaction weights used to select a value for f_2 are:

$$f_{2_{\tau_0}} = (.05/1.3) + (.08/1.3) = .1,$$

$$f_{2_{\tau_1}} = (.45/1.3) + (.72/1.3) = .9.$$

According to the level selection criteria, the one with the largest factor-level interaction weight is chosen, so level τ_1 (8) is selected for factor f_2.

The second factor to receive a value is f_1 since it has the second largest factor interaction weight. Factor f_1 has three levels to choose from. For each of these levels, weighted density is calculated in relation to the other factors. Since f_2 has already been fixed with a value, density is increased by the product of the weight of f_2's fixed level and the weight of the level being evaluated for selection. Factor f_0 on the other hand has not yet been fixed with a level so weighted density is increased by the formula that includes the likelihood of covering weight with this factor in the future.

$$f_{1_{\tau_0}} = (.1/1.3) + (.2 * .9) = .2569,$$

$$f_{1_{\tau_1}} = (.15/1.3) + (.3 * .9) = .38538,$$

$$f_{1_{\tau_2}} = (.15/1.3) + (.3 * .9) = .38538.$$

Table 7
Output of the prioritized greedy algorithm for the case study example

Row number	f_0	f_1	f_2
1	0	5	8
2	1	6	8
3	2	4	8
4	3	5	8
5	0	6	7
6	0	4	7
7	1	5	7
8	2	5	7
9	3	6	7
10	2	6	7
11	1	4	7
12	3	4	7

In this case, there is a tie between τ_1 and τ_2, broken here by taking the lexicographically first, τ_1.

Finally, the third factor to fix is f_0. The weighted density for f_0's levels is straightforward as it is the increase of weight offered in relation to the other fixed factors.

$$f_{0_{\tau_0}} = (.2 * .3) + (.2 * .9) = .24,$$

$$f_{0_{\tau_1}} = (.1 * .3) + (.1 * .9) = .12,$$

$$f_{0_{\tau_2}} = (.1 * .3) + (.1 * .9) = .12,$$

$$f_{0_{\tau_3}} = (.1 * .3) + (.1 * .9) = .12.$$

Level τ_0 offers the largest increase in weight and is assigned to f_0 in this test.

When the algorithm is given the opportunity to complete, the output is shown in Table 7.

5. Empirical results

Weights may be applied to factors, levels, or directly to individual tuples. We first experiment with weights applied to levels, and then to individual tuples.

5.1. Weighting experiments

The weighted density algorithm generates test suites that order rows (in general) in decreasing order of importance. Two algorithms are compared using the data in Table 9. The first utilizes unweighted density and the second uses weighted density to construct a biased covering array. The amount of weight covered in each row is shown in Table 8. Weighted density covers more weight earlier. Fig. 3 shows the difference in cumulative weight covered after each row.

Initially, one may have thought that the prioritization would result in a significant increase in the number of tests. For instance, in one example we ran 7^8 with half of the factors assigned the highest priority and the other half a low priority. This resulted in 85 tests as opposed to 77 when everything is weighted equally. Remarkably, though, unequal weights in Table 9 did not adversely affect the overall size of the test suite. We explore this further, considering several schemes for assigning weights.

Table 8
Weight covered in each row using Unweighted and Weighted density

Row number	Unweighted density (%)	Weighted density (%)
1	4.71	30.00
2	4.12	22.94
3	4.12	17.06
4	17.06	7.06
5	30.00	6.47
6	22.94	3.53
7	6.47	4.12
8	2.35	2.35
9	3.53	2.35
10	1.18	1.76
11	1.76	1.18
12	1.76	1.18

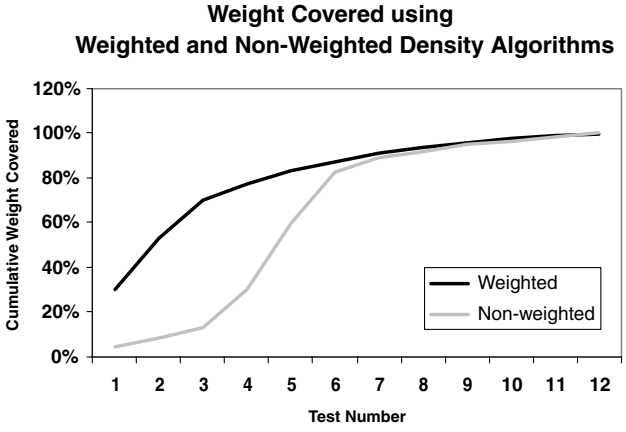

Fig. 3. Cumulative weight covered using Unweighted and Weighted density.

Table 9
Input of three factors and their levels and weights

	v_0	v_1	v_2	v_3
f_0	0 (.2)	1 (.1)	2 (.1)	3 (.1)
f_1	4 (.2)	5 (.3)	6 (.3)	–
f_2	7 (.1)	8 (.9)	–	–

Consider the input $20^2 10^2 3^{100}$ (this is a shorthand for 2 factors of 20 values each, 2 of 10 each, and 100 of 3 each), with four different weight distributions:

- *Distribution 1 (Equal weights)*. All levels have the same weight,
- *Distribution 2 ($\frac{50}{50}$ split)*. Half of the weights for each factor are set to .9 and the other half to .1,
- *Distribution 3 ($\left(\frac{1}{v_{\max}}\right)^2$ split)*. All weights of levels for a factor are equal to $\left(\frac{1}{v_{\max}}\right)^2$, where $v_{\max}$ is the number of levels associated with the factor,
- *Distribution 4 (Random)*. Weights are randomly distributed.

The rate of cumulative weight coverage for an input depends on the associated weight distribution. Fig. 4 shows the percentage of cumulative weight covered in the first

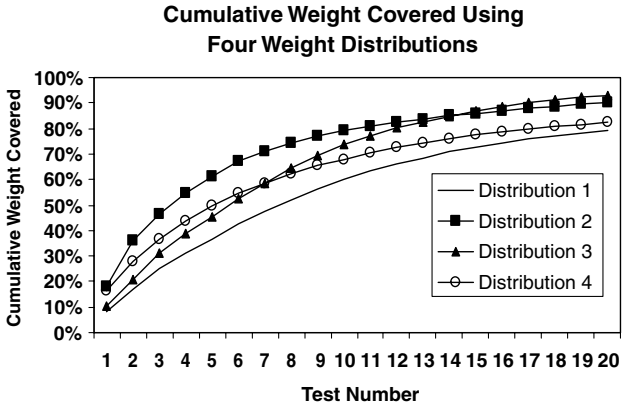

Fig. 4. Cumulative weight covered in the first 20 tests using input $20^2 10^2 3^{100}$.

20 tests for each of the four distributions. When all weights are equal, the result is a (non-biased) covering array. This exhibits the slowest growth of weight coverage early on. However, when there is more of a difference in the distribution of weights, a biased covering array can often amass more weight in the earliest rows. For instance, the $\frac{50}{50}$ split shows the most rapid coverage of growth earliest. This may be expected because half of the levels with a weight of .9 comprise the majority of the weight and are quickly covered in the early rows. The $\left(\frac{1}{v_{\max}}\right)^2$ split and random are similar to each other, and intermediate between the two extremes considered.

The size of the test suites generated also varies. For the four distributions, the final test suite has 401, 400, 416, or 405 rows, respectively. Distribution 3 emphasizes the importance of pairs involving factors with few levels, and in this example yields a larger covering array than unweighted density. In a more extreme example, we distributed weight as $\left(\frac{1}{v_{\max}}\right)^{10}$, producing a result of 433 rows.

This suggests that weighted density does generate tests of greater weight early, and that the weights themselves cause substantial variation in the ultimate size of covering array produced. We consider several more scenarios to examine this further. Table 10 shows the results in seven cases: three have all factors with the same number of levels and four are mixed-level. Unweighted density usually produces the smallest sized covering array. Figs. 5–8 show results of the cumulative weight distribution for each

Table 10
Sizes of biased covering arrays with different weight distributions

Weighting	Equal	$\left(\frac{1}{v_{\max}}\right)^2$	$\frac{50}{50}$ Split	Random
3^4	9	13	9	13
10^{20}	206	314	225	223
3^{100}	32	38	31	32
$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$	94	125	98	101
$8^2 7^2 6^2 2^4$	70	98	77	81
$15^1 10^5 5^1 4$	175	238	185	188
$3^{50} 2^{50}$	28	35	28	28

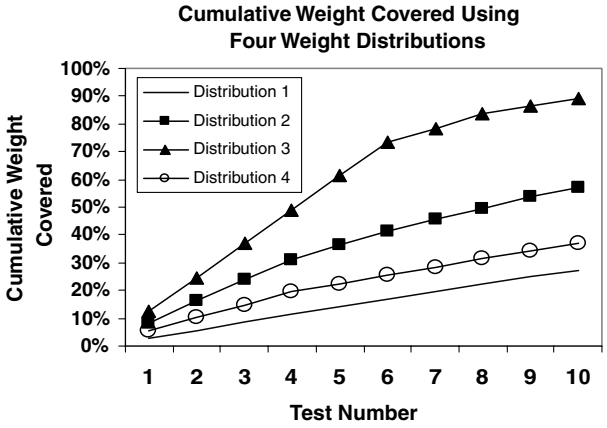

Fig. 5. Cumulative weight covered in the first 10 tests using input $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$.

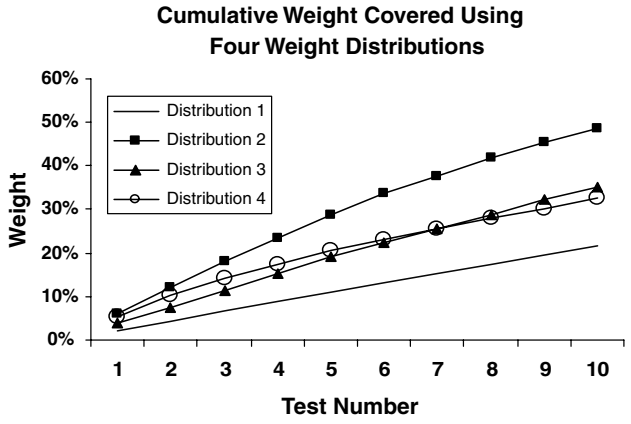

Fig. 6. Cumulative weight covered in the first 10 tests using input $8^2 7^2 6^2 2^4$.

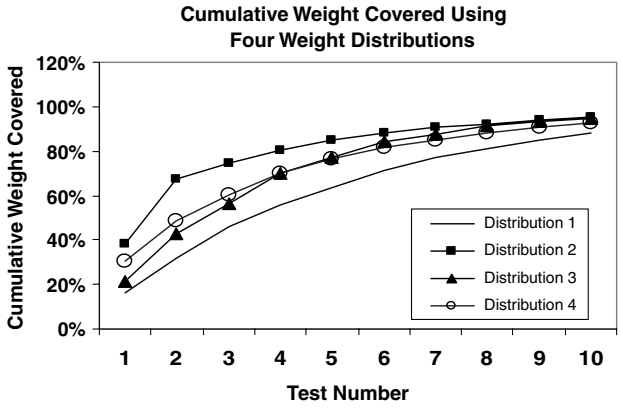

Fig. 7. Cumulative weight covered in the first 10 tests using input $3^{50}2^{50}$.

weight distribution. Naturally, in a typical application we cannot specify the weights, and hence these figures merely illustrate the algorithm's ability to accumulate large weight early. For instance, the first half of tests using any of the four weight distributions for 3^{100}, over 90% of the weight is covered; the first half of tests using any of the four weight distributions for $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$ covers between 75% and 99% of the total weight; and in $3^{50}2^{50}$, the first half of tests covers over 95% of the total weight. When simply generating a covering array, we can control the weights used to minimize the size of covering array generated. The variation in sizes is reported in Table 10.

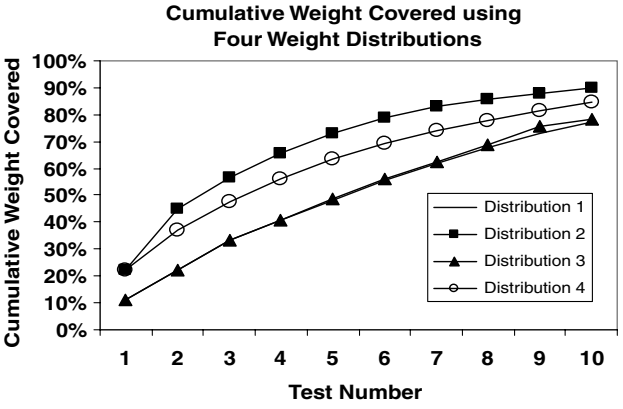

Fig. 8. Cumulative weight covered in the first 10 tests using input 3^{100}.

5.2. Avoids: an example

Consider a small component-based system with three components, as in Table 11. There are 16 pair-wise interactions; weights are shown in Table 12. Three of the pairs are neutral, and two are avoids, so only 11 pairs are required to be covered. Table 13 shows the output of a test suite that covers all pairs using a prioritized ordering.

Table 11
A small component-based system, $3^1 2^2$

Component	f_0	f_1	f_2
Option 1	0	3	5
Option 2	1	4	6
Option 3	2	–	–

Table 12
Weights of component pairs

f_x	f_y	w
0	3	0.5
0	6	0
1	5	0.375
2	4	−0.5
3	5	0.75
4	6	0
0	4	0
1	3	0.5
1	6	0.125
2	5	−.75
3	6	0.25
0	5	0.375
1	4	0.025
2	3	0.5
2	6	0.125
4	5	0.0375

Table 13
A prioritized test suite for pair-wise coverage

Test no.	f_0	f_1	f_2
1	0	3	5
2	1	4	5
3	2	3	6
4	1	3	6

All required pairs have been covered; the highest priority pairs are covered earliest. In this case, none of the two avoids or three neutral positions are covered.

In the next example, we generate $c = \{0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300, 350, 400\}$ avoids at random. When there are no avoids, there are 448 required pairs to cover for an input of eight factors that each have four levels (denoted as 4^8). As the number of avoids increases, the number of required pairs correspondingly decreases. In this example, all required pairs are weighted at 1.0 and the avoids are weighted at -1.0. Note that we do not use a uniform distribution on a non-biased covering array here as this reduces to the well-studied case of unordered covering arrays [5,9].

For each setting of c, we run our algorithm 20 times and plot the data. Fig. 9 shows the sizes of the covering arrays generated for the input 4^8 with varying numbers of avoids. The horizontal axis represents the number of avoids that are generated for input and the vertical axis indicates the sizes of the solutions. The sizes of solutions usually decrease when larger numbers of avoids are input. Fig. 10 shows that the percentage of avoids that ended up in solutions is small at 0–3.36%. This indicates that the algorithm does not include many avoids in end solutions.

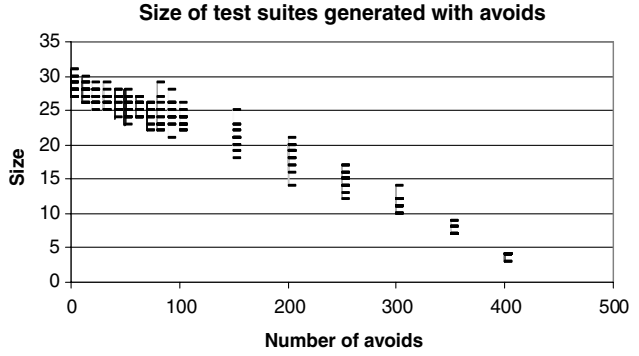

Fig. 9. Sizes of test suites for input 4^8 with $c = \{0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300, 350, 400\}$ randomly generated avoids.

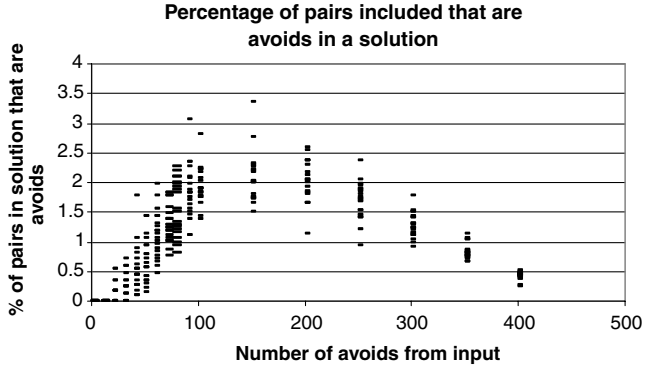

Fig. 10. Percentage of pairs in solutions that are avoids for input 4^8 and $c = \{0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300, 350, 400\}$ randomly generated avoids.

6. Conclusions

Independently, test prioritization and interaction testing have been widely studied. This paper describes a method to combine ideas from each. Algorithms to generate software interaction test suites have been evaluated on the basis of accuracy, execution time, consistency, and adaptability to seeding and constraints. This paper discusses a further criterion: prioritization based on user specified importance. An algorithm is described, and shown to be extensible to allow seeds and avoids. Computational results suggest that the method provides a useful and simple mechanism for generating prioritized test suites. Empirical studies are needed to identify the best prioritization criteria for test suites designed for interaction testing. In addition, effective heuristics for handling hard constraints (forbidden configurations) are sorely needed; constraint satisfaction methods [16] may offer an appropriate mechanism.

Acknowledgements

This research is supported by the Consortium for Embedded and Internetworking Technologies and by ARO Grant DAAD 19-1-01-0406.

References

[1] T. Berling, P. Runeson, Efficient evaluation of multifactor dependent system performance using fractional factorial design, IEEE Trans. Softw. Eng. 29 (9) (2003) 769–781.
[2] R.C. Bryce, Y. Chen, C.J. Colbourn, Biased covering arrays for progressive ranking and composition of web services, Int. J. Simul. Process Model., to appear.
[3] R.C. Bryce, C.J. Colbourn, Test prioritization for pairwise coverage, in: Proceedings of the Workshop on Advances in Model-Based Software Testing (A-MOST), St. Louis, MO, 2005, pp. 1–7.
[4] R.C. Bryce, C.J. Colbourn, M.B. Cohen, A framework of greedy methods for constructing interaction test suites, in: Proceedings of the 27th International Conference on Software Engineering (ICSE2005), 2005, pp. 146–155.
[5] D.M. Cohen, S.R. Dalal, M.L. Fredman, G.C. Patton, The AETG system: an approach to testing based on combinatorial design, IEEE Trans. Softw. Eng. 23 (7) (1997) 437–444.
[7] M.B. Cohen, C.J. Colbourn, P.B. Gibbons, W.B. Mugridge, Constructing test suites for interaction testing, in: Proceedings of the International Conference on Software Engineering (ICSE 2003), 2003, pp. 38–48.
[8] C.J. Colbourn, Combinatorial aspects of covering arrays, Le Matematiche (Catania) 58 (2004) 121–167.
[9] C.J. Colbourn, M.B. Cohen, R.C. Turban, A deterministic density algorithm for pairwise interaction coverage, in: Proceedings of the IASTED International Conference on Software Engineering, 2004, pp. 242–252.

[10] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, B.M. Horowitz, Model-based testing in practice, in: Proceedings of the International Conference on Software Engineering (ICSE '99), 1999, pp. 285–294.
[11] S. Dunietz, W.K. Ehrlich, B.D. Szablak, C.L. Mallows, A. Iannino, Applying design of experiments to software testing, in: Proceedings of the International Conference on Software Engineering (ICSE '97), 1997, pp. 205–215.
[12] S. Elbaum, A. Malishevsky, G. Rothermel, Test case prioritization: a family of empirical studies, IEEE Trans. Softw. Eng. 18 (2) (2002) 159–182.
[13] S. Elbaum, G. Rothermel, S. Kanduri, A. Malishevsky, Selecting a cost-effective test case prioritization technique, Softw. Q. J. 12 (3) (2004) 185–210.
[14] A. Hartman, L. Raskin, Problems and algorithms for covering arrays, Discrete Math. 284 (2004) 149–156.
[15] A. Hartman, Software and hardware testing using combinatorial covering suites, in: Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications, Springer, Berlin, 2005, pp. 237–266.
[16] B. Hnich, S. Prestwich, E. Selensky, Constraint-based approaches to the covering test problem, Lecture Notes Comput. Sci. 3419 (1) (2005) 172–186.
[17] D.S. Hoskins, C.J. Colbourn, D.C. Montgomery, Software performance testing using covering arrays, in: Fifth International Workshop on Software and Performance (WOSP 2005), Palma de Mallorca, Illes Balears, Spain, 2005, pp. 131–137.
[18] D. Kuhn, M. Reilly, An investigation of the applicability of design of experiments to software testing, in: Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002, pp. 91–95.
[19] D.R. Kuhn, D.R. Wallace, A.M. Gallo, Software fault interactions and implications for software testing, IEEE Trans. Softw. Eng. 30 (6) (2004) 418–421.
[20] A.K. Mackworth, Consistency in networks of relations, Artif. Intell. 8 (1977) 99–118.
[21] National Institute of Standards and Technology, The Economic Impacts of Inadequate Infrastructure for Software Testing, U.S. Department of Commerce, May 2002.
[22] G. Rothermel, R.H. Untch, C. Chu, M.J. Harrold, Prioritizing test cases for regression testing, ACM Trans. Softw. Eng. Methodol. 27 (10) (2001) 929–948.
[23] A. Srivastava, J. Thiagarajan, Effectively prioritizing tests in development environment, in: Proceedings of the International Symposium on Software Testing and Analysis, 2002, pp. 97–106.
[24] D. West, Introduction to Graph Theory, second ed., Prentice-Hall, Englewood Cliffs, NJ, 2000.
[25] J.A. Whittaker, J.H. Poore, Markov analysis of software specifications, ACM Trans. Softw. Eng. Methodol. 2 (1) (1993) 93–106.
[26] J.A. Whittaker, M.G. Thomason, A markov chain model for statistical software testing, IEEE Trans. Softw. Eng. 2 (10) (1994) 812–824.
[27] C. Yilmaz, M.B. Cohen, A. Porter, Covering arrays for efficient fault characterization in complex configuration spaces, in: Proceedings of the International Symposium on Software Testing and Analysis, 2004, pp. 45–54.