A Model for Spectra-based Software Diagnosis

LEE NAISH,
HUA JIE LEE
and
KOTAGIRI RAMAMOHANARAO
University of Melbourne

This paper presents an improved approach to assist diagnosis of failures in software (fault localisation) by ranking program statements or blocks according to how likely they are to be buggy. We present a very simple single-bug program to model the problem. By examining different possible execution paths through this model program over a number of test cases, the effectiveness of different proposed spectral ranking methods can be evaluated in idealised conditions. The results are remarkably consistent to those arrived at empirically using the Siemens test suite and Space benchmarks. The model also helps identify groups of metrics which are equivalent for ranking. Due to the simplicity of the model, an *optimal* ranking method can be devised. This new method out-performs previously proposed methods for the model program, the Siemens test suite and Space. It also helps provide insight into other ranking methods.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms: Performance, Theory

Additional Key Words and Phrases: fault localization, program spectra, statistical debugging

1. INTRODUCTION

Despite the achievements made in software development, bugs are still pervasive and diagnosis of software failures remains an active research area. One of many useful sources of data to help diagnosis is the dynamic behaviour of software as it is executed over a set of test cases where it can be determined if each result is correct or not (each test case passes or fails). Software can be instrumented automatically to gather data such as the statements that are executed in each test case. A summary of this data, often called program spectra, can be used to rank the parts of the program according to how likely it is they contain a bug. Ranking is done by sorting based on the value of a numeric function (we use the term *ranking metric* or simply metric) applied to the data for each part of the program. There is extensive literature on spectra-based methods in other domains, notably classification in botany, and this is the source for many ranking metrics that can be used for software diagnosis. We make the following contributions to this area:

(1) We propose a model-based approach to gain insight into software diagnosis.

Thanks to Tim Miller and the anonymous referees for their comments on drafts of this paper.

We present a very simple program that captures important aspects of the diagnosis problem. In this paper we focus on programs with a single bug, though the methodology can be adapted to multiple bugs.

(2) Because the model allows a more formal approach, we are able to define metrics for spectra-based ranking that are *optimal* for the case we examine. This helps us compare different metrics.

(3) We give the first comprehensive evaluation of spectra-based ranking methods. Over thirty ranking metrics are evaluated using our model, the Siemens test suite (several small C programs) and Space (a more realistically sized program). The results from the model closely match the empirical results, including the superior performance of the optimal metrics for diagnosis of single-bug programs.

(4) We prove that although all the metrics examined are distinct functions, several are equivalent for ranking purposes (they produce identical rankings).

Although diagnosing code with multiple bugs is the norm, there are several reasons for focussing on the single bug case. First, it provides a simpler instance of the problem to trial our model-based approach. A simpler model can be used, which allows much simpler analysis and theoretical results. Second, empirical comparison with other work requires the use of the same benchmarks, and the most widely used have a strong single-bug bias. Third, the program having a single bug is a sufficient condition for our results but it is not a necessary condition. A key condition, which depends on the set of test cases and is sufficient for some technical results, is a bug exists that is executed in every failed test case. This condition may be satisfied even though some of the failures are caused by other bugs and/or there are still other bugs that cause no failures. Furthermore, we typically have some control over the set of test cases. There is some previous work on partitioning test cases that attempts to satisfy this weaker condition — we discuss this in Section 10.

The rest of the paper is organized as follows. Section 2 provides the necessary background on spectra-based diagnosis, including the formulas used in the many ranking metrics from the literature. Section 3 discusses the related idea of using data on predicates rather than statement executions, and how the two approaches can be equivalent in some cases. Section 4 presents our model program. Section 5 describes our methodology for considering multisets of execution paths of the model program and how we evaluate performance of metrics in the model. In Section 6, we discuss optimal metrics. In Section 7 all metrics are compared, analysed and evaluated using our model. Various metrics are shown to be equivalent. Section 8 gives empirical performance results for the metrics using the Siemens test suite and Space benchmarks. Sections 9 and 10 discuss related and further work, respectively. Section 11 concludes.

evaluated, compared, and analyzed using our model

2. BACKGROUND

A program spectrum is a collection of data that provides a specific view of the dynamic behavior of software. It contains information about the parts of a program that were executed during the execution of several test cases. In general the parts could be individual statements, basic blocks, branches or larger regions such as functions. Here we use individual statements. This is equivalent to considering basic

Table I. Example program spectra with tests $T_1 \dots T_5$

	T_1	T_2	T_3	T_4	T_5	a_{np}	a_{nf}	a_{ep}	a_{ef}
Statement$_1$	1	0	1	1	1	0	1	3	1
Statement$_2$	1	1	0	1	0	2	0	1	2
Statement$_3$	0	1	0	1	1	1	1	2	1
⋮									
Test Result	1	1	0	0	0				

blocks, assuming normal termination (a statement within a basic block is executed if and only if the whole basic block is executed). During execution of each test case, data is collected indicating the statements that are executed. Additionally, each test case is classified as passed or failed.

For each statement, four numbers are ultimately produced. They are the number of passed/failed test cases in which the statement was/wasn't executed. Adapting Abreu et al. [2006] we use the notation $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$, where the first part of the subscript indicates whether the statement was executed (e) or not (n) and the second indicates whether the test passed (p) or failed (f). For example, a_{ep} of a statement is the number of tests passed and executed the statement. The raw data is often presented as a matrix of numbers (binary in this case), with one row for each program statement and one column for each test case, where each cell indicates whether a particular statement is executed (the value is 1) or not (the value is 0) for a particular test case. Additionally, there is a (binary) vector indicating the result (0 for pass and 1 for fail) of each test case. This data allows us to compute the a_{ij} values, $i \in \{n, e\}$ and $j \in \{p, f\}$. Table I gives an example with five tests, the first two of which fail.

Applying a function that maps the four a_{ij} values to a single number (we call such functions *ranking metrics*) for each statement allows us to rank the statements: those with the highest metric values are considered the most likely to be buggy. We would expect buggy statements to generally have relatively high a_{ef} values and relatively low a_{ep}. When a_{ef} is maximal and a_{ep} is minimal (the statement is executed in all failed tests but no passed tests) all the metrics we consider return maximal values. In general, different metrics give different weights to passed and failed test cases so they result in different rankings.

Diagnosis can proceed by the programmer examining statements starting from the top-ranked statement until a buggy statement is found. In reality, programmers are likely to modify the ranking due to their own understanding of whether the code is likely to be buggy, based on other information such as static analysis, the history of software changes, *et cetera*, and checking correctness generally cannot be done by a single statement (or even one basic block) at a time. Our evaluation of different metrics ignores such refinements and just depends on where the bug(s) appear in the ranking.

Mathematically, many (though not all) proposed metrics are closely related to *norms* of the associated *metric space* (our use of the term *metric* is not related to metric spaces in any precise technical sense). A norm is a measure of the distance between two points: it must be positive for distinct points, symmetric ($dist(a, b) = dist(b, a)$) and must satisfy the triangle inequality ($dist(a, c) \leq$

$dist(a,b)+dist(b,c)$). Given a norm, statements (rows in the matrix) can be ranked according to how close they are to the result vector (whether each test passes or fails). Some of the ranking metrics we use are referred to as norms in the literature; others are called measures or distances. Some are defined in more general cases, such as where we have a matrix and vector of arbitrary numbers. We restrict attention to the special case of binary numbers where the data is summarised by the four a_{ij} values. This induces equivalence of some ranking metrics that are distinct in more general cases. Additional equivalences are induced because we are only interested in the ranking of statements produced — not the metric values per se (see Section 7.2).

All metrics project points in many dimensions (proportional to the number of test cases) onto a single dimension. This is a simple instance of the *clustering* problem where there are just two clusters we want to distinguish — the correct statements (with low metric values) and buggy statements (with high metric values). The methodology can be used in many domains. For example, instead of program statements and test cases that fail or pass we might have genes and individuals who do or do not have some form of disease; the goal being to determine the genes most likely to be associated with the disease. Usually there can be both "false negatives" (for example, the disease is not present or the test case succeeds even though the bad gene is present or the buggy statement is executed) and "false positives" (for example, the disease is present even though the bad gene is not). In general the data may not be binary: genes may be expressed to varying degrees (even in software diagnosis its possible to consider the number of times a statement is executed in each test case) and the severity of the disease may vary. There are infinitely many possible metrics and numerous metrics have been proposed in the literature for predicting the statements most likely to be buggy or for the equivalent classification in other domains — see Table II. We briefly review their sources next.

The oldest of the metrics is Jaccard [1901], originally used for classification in the botany domain but since used in many areas. Botany has also spawned several other metrics: Ochiai [1957], Russell and Rao [1940], Sørensen and Dice [Dice 1945; Duarte et al. 1999], Rogers and Tanimoto [1960], Anderberg [1973] and Simple-Matching [Meyer et al. 2004]. Jaccard, Dice, Overlap and a more general version of the Ochiai metric called Cosine have been used in the field of information retrieval [Dunham 2002]. Hamming [1950] was originally introduced for error detecting/correcting codes; for binary numbers it is equivalent to Lee [1958] and Manhattan [Krause 1973] (we refer to these as Hamming etc. in Table II). Manhattan and Euclid [Krause 1973] have been used in clustering. Jaccard and Simple-Matching have also been used in clustering, along with with two unnamed metrics that we refer to as M1 and M2, respectively [Everitt 1978]. In the area of biometrics various metrics have been introduced: Goodman and Kruskal [1954], Scott [1955], Fleiss [1965], Cohen [1960], Geometric Mean [Maxwell and Pilliner 1968] as well as Arithmetic Mean, Harmonic Mean, Rogot1 and Rogot2 [Rogot and Goldberg 1966]. In the field of clustering using the Self-Organizing Maps (SOM) algorithm various similarity measures or metrics have been introduced and evaluated: Kulczynski1, Kulczynski2, Hamann and Sokal [Lourenço et al. 2004].

Spectra-based ranking has also been applied to software diagnosis by various

Table II. Definitions of ranking metrics previously used

Name	Formula	Name	Formula
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$	Anderberg	$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$
Sørensen-Dice	$\frac{2a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}$	Dice	$\frac{2a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Kulczynski1	$\frac{a_{ef}}{a_{nf}+a_{ep}}$	Kulczynski2	$\frac{1}{2}\left(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ef}}{a_{ef}+a_{ep}}\right)$
Russell and Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Hamann	$\frac{a_{ef}+a_{np}-a_{nf}-a_{ep}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$
Simple Matching	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Sokal	$\frac{2(a_{ef}+a_{np})}{2(a_{ef}+a_{np})+a_{nf}+a_{ep}}$
M1	$\frac{a_{ef}+a_{np}}{a_{nf}+a_{ep}}$	M2	$\frac{a_{ef}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$
Rogers & Tanimoto	$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+2(a_{nf}+a_{ep})}$	Goodman	$\frac{2a_{ef}-a_{nf}-a_{ep}}{2a_{ef}+a_{nf}+a_{ep}}$
Hamming etc.	$a_{ef}+a_{np}$	Euclid	$\sqrt{a_{ef}+a_{np}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$	Overlap	$\frac{a_{ef}}{\min(a_{ef},a_{nf},a_{ep})}$
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$
Ample	$\left\lvert\frac{a_{ef}}{a_{ef}+a_{nf}}-\frac{a_{ep}}{a_{ep}+a_{np}}\right\rvert$	Wong1	a_{ef}
Wong2	$a_{ef}-a_{ep}$		
Wong3	$a_{ef}-h,\ \text{where } h=\begin{cases}a_{ep} & \text{if } a_{ep}\le 2\\ 2+0.1(a_{ep}-2) & \text{if } 2<a_{ep}\le 10\\ 2.8+0.001(a_{ep}-10) & \text{if } a_{ep}>10\end{cases}$		
Ochiai2	$\frac{a_{ef}a_{np}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$		
Geometric Mean	$\frac{a_{ef}a_{np}-a_{nf}a_{ep}}{\sqrt{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}}$		
Harmonic Mean	$\frac{(a_{ef}a_{np}-a_{nf}a_{ep})((a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np}))}{(a_{ef}+a_{ep})(a_{np}+a_{nf})(a_{ef}+a_{nf})(a_{ep}+a_{np})}$		
Arithmetic Mean	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{nf})+(a_{ef}+a_{nf})(a_{ep}+a_{np})}$		
Cohen	$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(a_{np}+a_{ep})+(a_{ef}+a_{nf})(a_{nf}+a_{np})}$		
Scott	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$		
Fleiss	$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$		
Rogot1	$\frac{1}{2}\left(\frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}}+\frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}}\right)$		
Rogot2	$\frac{1}{4}\left(\frac{a_{ef}}{a_{ef}+a_{ep}}+\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{np}}{a_{np}+a_{ep}}+\frac{a_{np}}{a_{np}+a_{nf}}\right)$		

researchers. In Section 9 we give more details of this work. The Tarantula system [Jones and Harrold 2005] is credited as the first system to apply spectra-based ranking to software diagnosis (for an imperative language). The AMPLE system [Dallmeier et al. 2005] was designed for diagnosis of object-oriented software. The Ample metric we use is actually taken from subsequent work [Abreu et al. 2007]; it is just one possible generalisation of the actual formula used in the AMPLE system, which only ever has a single failed test case. The Pinpoint web diagnosis system [Chen et al. 2002] uses the Jaccard metric. Ochiai, Jaccard, Ample and Tarantula metrics have been evaluated for diagnosis using the Siemens test suite [Abreu et al. 2007]. More recently, the Zoltar metric [Gonzalez 2007] has been proposed and used for software diagnosis and in [Wong et al. 2007] several metrics (Wong1-3) were proposed and evaluated.

The intuition behind the metrics varies greatly. Here we give three brief examples. Jaccard devised a way to measure the similarity of two sets. It is the size of their intersection divided by the size of their union. We can apply this idea to spectral diagnosis by using the set of test cases that fail and the set of test cases for which a particular statement is executed. A second approach is to view a row of the matrix as a vector in n-dimensional space. The cosine of the angle between this vector and the vector of test results is another measure of similarity — this is what the Ochiai metric computes. A third way is to think of the rows and results as bit strings. The number of bits that differ in two strings is a measure of dis-similarity (the Hamming distance). By first taking the complement of one of the bit strings we obtain a measure of similarity — this is what our formula for the Hamming metric computes.

Several of the metrics contain quotients where the denominator can be zero. If the numerator is zero we use zero otherwise we use a suitably large value. For example, the Overlap formula we can use the number of tests plus 1, which is larger than any value which can be returned with a non-zero denominator. An alternative is to add a suitably small ϵ to the denominator.

3. INSTRUMENTING PREDICATES

An alternative approach to bug localization is to instrument certain *predicates* in the code rather than statement execution [Liblit et al. 2005; Liu et al. 2005]. Each predicate is associated with a single program point. Examples of useful predicates are conditions of `if` statements, and whether the "return" expressions of functions are positive, negative or zero. During execution of a test case, data can be gathered on the predicates that are "observed" (execution has reached that program point) and of those, the ones that were true (at least once). This data can then be used to rank the different predicates according to how well they predict failure of a test case. For example, it may be that a failure occurs if and only if a certain variable is zero at a certain program point. If we have a predicate that tests this, it is likely to be highly ranked and useful for diagnosis.

One implementation of these ideas is the Cooperative Bug Isolation (CBI) system [Liblit 2004; Liblit et al. 2005], in which predicates are ranked as follows. Each predicate B has values *Failure(B)*, defined as the number of failed tests where B was true divided by the total number of tests where B was true, and $Context(B)$, defined as the number of failed tests where B was observed (executed) divided by the total number of tests where B was observed. One ranking metric proposed is $Increase(B)$, defined as $Failure(B) - Context(B)$. There are also two variations, which attempt to combine *Increase* with another simple metric (the proportion of failed tests in which B is true), using a harmonic mean. One uses logarithms [Liblit et al. 2005]; the other (from the source code distribution) uses square roots.

If the only predicates instrumented are related to control flow, instrumenting statement execution provides the same information. For example, in the program segment *S1; if B then S2*, we know B is observed if and only if *S1* is executed and B is true if and only if $S2$ is executed. Note that although the information about B is available, it is spread across two different statements, so the program spectra methods we consider (which never combine a_{ij} values from different statements) may not always be able to reproduce the predicate-based ranking. An apparent advantage of predicate-based systems is that predicates unrelated to control flow can be introduced. This allows our *domain knowledge* to be used to obtain more relevant information. For example, whether the return value of a function is negative is often important. However, any predicate P deemed worth instrumenting can be used as a control flow predicate by introducing a dummy `if` statement that tests the predicate then does nothing. Some predicate based system, such as CBI, use sampling of data points rather than collecting all the data, but this can equally be applied to statement-based methods. Hence there is no fundamental difference between instrumenting predicates and statements.

The model program we use is very simple, and for a given set of tests, *Context* is the same for all control flow predicates (they are always observed). This allows the predicate-based metrics used in CBI to be translated into the statement-based program spectra formalism and compared fairly with other metrics (there is a bijection between statements and predicates and our translation of the formulas preserves the ranking in all cases). We refer to these metrics as CBI Increase (abbreviated Inc), CBI Log and CBI Sqrt. The definitions are given in Table III. We use the same formulas in our empirical experiments. However, for the test suite programs, *Context* can vary between different predicates. This means our translation of the CBI predicate-based metrics to statement-based metrics does not preserve ranking, so the empirical performance we report for these metrics may not reflect that of the CBI system.

4. A MODEL PROGRAM

Figure 1 gives the code for the If-Then-Else-2 (*ITE2*) program segment which we use as a model for single-bug programs. It is assumed function `s1()` and `s2()` exist and may have side effects such as assignments to variables, whereas Boolean functions `t1()`, `t2()` and `t3()` return Booleans but have no side effects. The intention is that *ITE2* should assign `True` to variable `x`. There are also intentions for the individual statements and these are met by the program except for *S4*, which may sometimes assign `False` instead of `True` to `x`.

We use `t3()` to model the fact that buggy code may sometimes behave correctly and sometimes trigger a failure. We use `t2()` (and the second if-then-else) to model the fact that buggy code may not be executed in every run. We use `t1()` to

```
if (t1())
    s1();        /* S1 */
else
    s2();        /* S2 */
if (t2())
    x = True;    /* S3 */
else
    x = t3();    /* S4 - BUG */
```

Fig. 1. Program segment If-Then-Else-2 (ITE2)

model the fact that ranking methods (and debugging in general) must cope with "noise". It may be that the execution of *S1*, for example, is strongly correlated with failed test runs. This may be due to logical dependencies within the program or the particular selection of test data. The "signal" we want to detect is associated with `t2()` — the buggy statement is executed if and only if `t2()` returns `False`. The signal is essentially attenuated by `t3()` — if `t3()` almost always returns `True` there is little signal we can detect (and its more likely that the noise will be greater, leading to *S1* or *S2* being ranked top).

Our intuition suggested that having noise and an attenuated signal were the two most important features we needed in a model, and *ITE2* is the simplest model program we could think of that has these features. Despite its simplicity, this model has been very useful in evaluating, understanding and improving the performance of spectral diagnosis methods, as our later results show. There are many ways the model could be extended, for example, by:

—having more bugs,

—having more sources of noise,

—simulating loops, so both branches of an if-then-else can be executed in a single test, and

—having statements that are executed more or less often over typical sets of tests (in $ITE2$ all statements are executed in half the tests, on average).

The way we evaluate performance of metrics (described in the next section) is independent of the model. We have experimented with all these extensions and report some general observations here.

5. PERFORMANCE EVALUATION USING MULTISETS OF EXECUTION PATHS

We use the term *execution path* to mean the set of statements executed for a particular test, along with the result of the test. A single test case determines the execution path (for deterministic programs at least). A set of test cases determines a *multiset* of execution paths (two or more distinct test cases might result in the same execution path), which determines the a_{ij} values for each statement and the performance of a given metric for that set of tests cases. We abstract away the details of test cases (and sidestep the issue of nondeterminism) and focus on multisets of execution paths. A typical metric ranks the buggy statement highly for some multisets of execution paths but not for others. Ideally we would like a test set to give an even coverage of all execution paths but in reality test sets are

```
if (t3()) x = True; else x = False;    /* S4 - BUG */
```

Fig. 2. Coding of *S4* with two explicit paths

often far from this ideal. Our methodology for evaluating overall performance of a metric uses the number of tests as a parameter. Given a number of tests t, we determine the average performance over *all possible multisets* of t execution paths (Table IV gives results from our model for each metric for various values of t we choose between 2 and 1000). When the number of possible multisets is not too large we generate each distinct multiset once to evaluate the overall performance of each metric. For larger numbers of multisets we compute estimates by randomly sampling multisets from a uniform distribution.

5.1 Generating multisets of execution paths

There is a naive way of generating a random multiset of execution paths — simply execute the model program the required number of times with each predicate being a random choice of true or false. However, this leads to extremely skewed distributions. For example, consider a program with just two paths, one correct and one that always results in failure. With 500 tests there are 501 possible multisets of execution paths (with zero up to 500 failed tests). However, the probability of generating the one with 500 failed tests is $1/2^{500}$, rather than $1/501$. It is hard to determine the most realistic distribution, but a uniform distribution is the simplest assumption we can make conceptually (though not the simplest to implement). Also, other distributions can easily be generated from a uniform distribution, or sub-sets of the total space can be examined (which is what we report in our experiments). Even without refining the distribution, the results obtained from our model fit reasonably well with empirical results.

To distinguish between failed and passed tests, the two possible return values of `t3()` in $ITE2$ are treated as two separate execution paths. This is equivalent to re-coding statement *S4* as in Figure 2. For the first if-then-else of $ITE2$ there are two possible paths: *S1* and *S2*. For the second if-then-else, there are three possible paths: *S3* (always returning True), *S4* returning True and *S4* returning False. This gives a total of six execution paths of interest for the whole program segment. However, *S4* is used in twice as many paths as *S3*. To remove this bias we also treat *S3* as having two paths (both correct), giving a total of four possible paths for the second if-then-else and eight paths overall. We use this model in our experiments and refer to it as $ITE2_8$. Increasing the number of execution paths can greatly increase the number of multisets but since we just use random sampling it has relatively little effect on our ability to evaluate metrics.

The system we use to conduct our experiments supports a high level definition of the model program code, including the number and placement of buggy statements, program points that lead to test case failure and duplication of execution paths. This has allowed us to experiment with a wide range of other models. A small Prolog program converts the model definition to C macros, which define such things as the number of statements, the statements that are buggy, the total number of execution paths and how the program spectra are computed for each statement, given a multiset of execution paths. These are included in a C program which generates multisets and computes the performance of different metrics.

Given p (≥ 1) execution paths in the program (8 for $ITE2_8$) and t (≥ 0) tests, the number of distinct multisets of execution paths, $f(t, p)$, can be determined as follows. A multiset can be represented as a sequence of natural numbers of length p (the execution count for each path) that sum to t. If $t = 0$ or $p = 1$ only one multiset is possible. In general, the first number can be anything from 0 to t and the rest of the sequence is length $p - 1$, so we can use the recurrence:

$$f(t, p) = \sum_{i=0}^{t} f(i, p - 1)$$

The problem is equivalent to the more classical "balls and pins" or "books and bookends" problems (the number of ways t books can be partitioned on a shelf using $p - 1$ bookends) and there are several equivalent recurrences. The solution is the Binomial numbers (also known as Pascal's triangle) — $f(t, p) = C(t + p - 1, t)$ where

$$C(n, k) = \frac{n!}{k!(n-k)!}$$

We use a bijection between integers in the range 1 to $f(t, p)$ and sequences of p integers that sum to t. We repeatedly generate a random number in this range (or generate each number once if the range is small enough) and map it to a sequence of path counts which represents a multiset.

5.2 A scoring function to evaluate performance

For each sequence (multiset), the a_{ij} are determined and the ranking metric is evaluated for each statement. We determine the *score* for that multiset (we use this scoring function for $ITE2_8$ but generalise it for other models in a straightforward way):

Definition 5.1 *Score for ranking metric M with paths X.* Suppose M is a ranking metric, X is a multiset of execution paths and m_1, m_2, m_3 and m_4 are M applied to the a_{ij} tuples corresponding to X for statements $S1$, $S2$, $S3$ and $S4$, respectively. The *score* for M with X is 0 if m_4 is less than any other m_i. Otherwise, if exactly k of the m_i are equal-highest (including m_4), the score is $1/k$.

For example, suppose $t = 5$ and there is one execution of *S1*;*S4* that fails, two executions of *S2*;*S4* that pass and two executions of *S1*;*S3* that pass. For *S1* and *S4*, $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$ would be $\langle 2, 0, 2, 1 \rangle$ and for *S2* and *S3* it would be $\langle 2, 1, 2, 0 \rangle$. Using the Jaccard metric m_1, m_2, m_3 and m_4 would be 1/3, 0, 0 and 1/3, respectively so $S4$ (the bug) and $S1$ would be ranked equal-highest and the score would be 1/2.

This scoring function is designed to be as simple as possible while treating equally ranked statements in a reasonable way. If a score of one was given when $S4$ was ranked equal-highest with other statements the score could be maximised by a trivial metric that always returns zero, for example. If equal ranked statements resulted in a score of zero it would be advantageous if the metric was nondeterministic: adding a tiny random ϵ to each result would eliminate most ties, increasing the score in some cases but never decreasing it. We further discuss scoring functions in Section 8.2.

Definition 5.2 *Total score for ranking metric M with t tests.* The *total score* for ranking metric M with test set size t is the sum of the scores for M with paths X, over all possible multisets X of t execution paths that contain at least one failed path.

In our experiments we report percentage scores: the total score for all multisets examined divided by the number of multisets, times 100. We give percentage scores for multisets of paths with certain characteristics (such as particular numbers of failed cases). Multisets of paths without these characteristics are simply ignored when computing percentage scores. We always ignore multisets with no failed tests. For example, with a single test there are just eight multisets of paths. Six of these have no failed tests and are ignored. The remaining two multisets have a single execution path containing $S1$ or $S2$ followed by (the failing path through) $S4$. For both these multisets the Jaccard metric (and other reasonable metrics) give a score of 0.5. The percentage score is thus $(0.5 + 0.5)/2 \times 100 = 50\%$.

6. OPTIMAL RANKING

For any given multiset of paths, metrics exist that rank $S4$ top (or equal top) but no single metric exists that ranks *S4* top for all multisets. Thus no metric is best in all cases. However, it is possible to have metrics that are optimal in the sense that they maximise the total score over all possible multisets.

Definition 6.1 *Optimal ranking metric for t tests.* A ranking metric is *optimal* for test set size t if no other ranking metric has a higher total score with t tests.

This definition is most appropriate if we assume a uniform distribution, since each multiset of paths is considered equally. It would be possible to generalise the definition so each multiset has a *weight*, which is multiplied by the score for that multiset. Ignoring some multisets, as we do in our experiments, is equivalent to having a weight of zero for those multisets and a weight of one for other multisets. Note that a metric that is optimal for a uniform distribution (equal weights) may not be optimal for other distributions, but may be close to optimal.

6.1 Optimal Ranking metric O

We now define a ranking metric and prove it is optimal (for $ITE2_8$) for all numbers of tests.

Definition 6.2 *Ranking metric O.*

$$O(a_{np}, a_{nf}, a_{ep}, a_{ef}) = \begin{cases} -1 & \text{if } a_{nf} > 0 \\ a_{np} & \text{otherwise} \end{cases}$$

Superficially this is a rather odd metric as it only uses a_{nf} and a_{np}, which appear to be the least important variables in most other metrics. However, for single-bug programs such as $ITE2_8$, a_{nf} is always zero for the buggy statement, so any statement with a non-zero a_{nf} can be given the lowest rank. Furthermore, any two statements that have $a_{nf} = 0$ must have the same a_{ef} value since $a_{nf} + a_{ef}$ is the total number of failed tests (which is the same for all statements). Similarly,

$a_{np} + a_{ep}$ is the total number of passed tests, so there is only one non-trivial degree of freedom. There is no *a priori* reason to suppose a buggy statement is executed more or less often than a correct statement. However, since all failed tests use the buggy statement, a_{ep} will tend to be smaller for buggy statements, so a_{np} will tend to be higher.

PROPOSITION 6.3. *For single-bug programs, increasing the value returned by a ranking metric when $a_{nf} > 0$ never increases the total score for the metric.*

PROOF. Since there is a single bug, $a_{nf} > 0$ only for non-buggy statements. Increasing the rank of a non-buggy statement never increases the score for a multiset, so the total score is never increased. □

The key lemma below shows that O cannot be improved by making a minimal change to the ranking it produces. O ranks a statement with $a_{nf} = 0$ and $a_{np} = x$ lower than a statement with with $a_{nf} = 0$ and $a_{np} = x + 1$, and no statements are ranked between these. We consider the effect of swapping the ranks of two such statements, or making the ranks equal.

LEMMA 6.4. *Suppose O' and O'' are ranking metrics such that for a single x*

$$O'(a_{np}, a_{nf}, a_{ep}, a_{ef}) = \begin{cases} x + 1 & \text{if } a_{np} = x \text{ and } a_{nf} = 0 \\ O(a_{np}, a_{nf}, a_{ep}, a_{ef}) & \text{otherwise} \end{cases}$$

$$O''(a_{np}, a_{nf}, a_{ep}, a_{ef}) = \begin{cases} x + 1.5 & \text{if } a_{np} = x \text{ and } a_{nf} = 0 \\ O(a_{np}, a_{nf}, a_{ep}, a_{ef}) & \text{otherwise} \end{cases}$$

For $ITE2_8$, O has a total score at least as high as O' and O'' for all test set sizes t.

PROOF. For O'' the ranking of statements where $a_{np} = x$ and $a_{np} = x + 1$ is swapped compared to the ranking using O and for O' they have equal rank; the relative rank of all other pairs of statements is the same as that using O. Since $S3$ is not executed in the failed test(s) it has a metric value of -1, which is strictly less than value for $S4$ and will not affect the score for any multiset. Similarly, at least one of $S1$ and $S2$ have a score of -1 for each multiset: whenever $S1$ is used $S2$ is not used and vice versa, so they can't both be executed in (all) the failed test(s). We can never have a tie between $S1$ and $S2$ unless both have value -1.

Thus O has a total score greater or equal to that of O' and O'' if and only if the number of multisets such that $\langle a_{np}, a_{nf}\rangle = \langle x + 1, 0\rangle$ for $S4$ and $\langle a_{np}, a_{nf}\rangle = \langle x, 0\rangle$ for $S1$ or $S2$ (whichever has the larger score) is greater or equal to the number of multisets such that $\langle a_{np}, a_{nf}\rangle = \langle x + 1, 0\rangle$ for $S1$ or $S2$ (whichever has the larger score) and $\langle a_{np}, a_{nf}\rangle = \langle x, 0\rangle$ for $S4$. Due to the symmetry between $S1$ and $S2$, we can just take the numbers of multisets where $S1$ has the larger score, and double it. All these multisets have $a_{nf} = 0$ for $S1$ and $S4$ so we leave this constraint implicit below.

The six passed paths through $ITE2_8$ can be classified as follows: two pass through $S1$ but not $S4$, one passes through $S4$ but not $S1$, two pass through neither $S4$ or $S1$, and one passes through $S4$ and $S1$ (due to symmetry the same holds if we replace $S1$ by $S2$ here). Let b be the number of tests using neither $S4$ or $S1$ (it contributes to the a_{np} total for *both* since a_{np} is the number of passed tests that *do not* execute the statement). Also, let $y = x - b$ and $z = t - 2y - b - 1$.

The number of multisets such that $a_{np} = x + 1$ for $S4$ and $a_{np} = x$ for $S1$ is the number of multisets where there are $y + 1$ paths that contribute only to a_{np} for $S4$, y paths that contribute only to a_{np} for $S1$, b paths that contribute to both and the remaining z paths that contribute to neither. The number of distinct paths in these categories in $ITE2_8$ are two, one, two and one, respectively (as noted above), and b can range between zero and $\lfloor t/2 \rfloor$ so the number of multisets is

$$\sum_{b=0}^{\lfloor t/2 \rfloor} f(y+1,2)f(y,1)f(b,2)f(z,1)$$

Similarly for $S2$. By the same reasoning, the number of multisets such that $a_{np} = x + 1$ for $S1$ and $a_{np} = x$ for $S4$ is

$$\sum_{b=0}^{\lfloor t/2 \rfloor} f(y,2)f(y+1,1)f(b,2)f(z,1)$$

Similarly for $S2$. It is sufficient to show that $f(y+1,2)f(y,1) \geq f(y,2)f(y+1,1)$ in all cases. Using definitions of f and C and simplifying we obtain

$$C(y+2, y+1)C(y,y) \geq C(y+1,y)C(y+1,y+1)$$

$$C(y+2,y+1) \geq C(y+1,y)$$

$$y+2 \geq y+1$$

which is true. □

Theorem 6.5 Optimality of O. *Ranking metric O is optimal with respect to $ITE2_8$ for all test set sizes.*

Proof. Suppose some ranking metric O' differs from O. Due to Proposition 6.3 any difference in ranking for tuples such that $a_{nf} > 0$ cannot make O' better than O. For tuples in which $a_{nf} = 0$, a_{ef} is the number of failed tests, which is the same for all tuples, and $a_{np} + a_{ep}$ is the number of passed tests. Thus the number of distinct values for the tuples is the number of distinct values for a_{np} and O' is equivalent to a function that maps these tuples to the range of a_{np} values. O can be modified to obtain an equivalent ranking in a finite number of steps by swapping the ranking of adjacent values or making them equal. But by Lemma 6.4 none of these steps would increase the total score. □

6.2 Other optimal ranking metrics

The proofs above can be generalised to show that a ranking metric M is optimal for $ITE2_8$ if, given a fixed number of passed and failed tests,

(1) when $a_{nf} = 0$, M is increasing in a_{np} or (equivalently) decreasing in a_{ep} (the key requirement for Lemma 6.4), and

(2) when $a_{nf} > 0$, the value returned is always less than any value returned when $a_{nf} = 0$ (allowing use of Proposition 6.3).

Table III. Definitions of new ranking metrics used

Name	Formula	Name	Formula
O	-1 if $a_{nf} > 0$, otherwise a_{np}	O^p	$a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$
Binary	0 if $a_{nf} > 0$, otherwise 1	Ample2	$\frac{a_{ef}}{a_{ef}+a_{nf}} - \frac{a_{ep}}{a_{ep}+a_{np}}$
CBI Inc	$\frac{a_{ef}}{a_{ef}+a_{ep}} - \frac{a_{ef}+a_{nf}}{a_{ef}+a_{nf}+a_{np}+a_{ep}}$	CBI Log	$\frac{2}{\frac{1}{CBI\ Inc}+\frac{\log(a_{ef}+a_{nf})}{\log a_{ef}}}$
CBI Sqrt	$\frac{2}{\frac{1}{CBI\ Inc}+\frac{\sqrt{a_{ef}+a_{nf}}}{\sqrt{a_{ef}}}}$	Wong3′	$\begin{cases} -1000 & \text{if } a_{ep}+a_{ef}=0 \\ Wong3 & \text{otherwise} \end{cases}$

O is the simplest optimal metric from an information-theoretic perspective as it only gives different ranks when necessary. Metrics can also be considered from a geometrical perspective — each metric defines a surface in three dimensions (the four a_{ij} values give just two degrees of freedom if we fix the number of passed and failed tests). We propose O^p (see below), which defines a very simple surface — a plane. It is optimal for $ITE2_8$ since a_{ef} is maximal when $a_{nf} = 0$ and a_{ep} varies from 0 to at most the number of passed tests, so the fractional component is strictly less than one.

Definition 6.6 *Ranking metric O^p.*

$$O^p(a_{np}, a_{nf}, a_{ep}, a_{ef}) = a_{ef} - \frac{a_{ep}}{P+1}$$

where P is the number of passed test cases.

O^p has the advantage of performing more rationally than O for multiple-bug programs. If there is more than one bug, a_{nf} can be non-zero for *all* statements, leading to O being -1 in all cases. In contrast, O^p ranks statements first on their a_{ef} value and, even if this is not maximal, second on their a_{ep} value. O^p and other optimal metrics can be helpful in comparison of metrics (see section 7.6). We also use O^p in our empirical evaluation of metrics (to aid comparison with other work, some multiple bug programs are used). In our experiments we also evaluate the performance of a simplified version of O, called Binary, which ignores a_{np} and allows us to see the relative importance of the two components of O. We also include a variation of the Ample metric which avoids taking the absolute value and a variation of the Wong3 metric which has a special case for statements that are not executed in any test case (motivated by our empirical studies). The definitions of all the new metrics we use are shown in Table III.

Although we have formally proved the optimality for $ITE2_8$ only, O and O^p are optimal for a much broader class of single bug programs. Proposition 6.3 holds for all single bug programs and the combinatorial argument in the proof of Lemma 6.4 can be generalised. With larger numbers of paths and/or sources of “noise” it is typically sufficient to show $\forall y\ f(y+1, j+k)f(y, j) \geq f(y, j+k)f(y+1, j)$, where j and k are positive integers dependent on the number of paths through the program with particular characteristics.

7. MODEL RESULTS

There are several reasons why our approach of modelling the diagnosis problem using a simple program such as $ITE2_8$ is beneficial. First, it is simple to run many experiments where various parameters are controlled precisely. The vagaries, biases, restrictions and cost of constructing or gathering "real" code and test suites are avoided. Such experiments can inspire hypotheses about behaviour of various metrics *et cetera*, which can then be investigated further. For example, we have proved the equivalence of various metrics having noticed they produced the same results for all test suite sizes using our model — see Section 7.2. Second, if some result holds for the model and the Siemens test suite, say, we can be more confident it will also hold for other situations (more than if the result holds for only one of these at least). Third, a model allows a more analytical approach, rather than empirical. A clear example is our development of optimal metrics.

7.1 Test suite size

We first investigate how well different ranking methods perform with different numbers of tests using the $ITE2_8$ model. We give the percentage score for each of the metrics discussed earlier for a variety of test suite sizes — see Table IV. To compute these figures we considered every multiset for up to 50 tests and sampled 2.1 billion multisets for larger numbers of tests (for 50 tests the number of distinct multisets is around 264 million and for 1000 tests it is around 2×10^{17}). Except where noted, we get consistent results to four decimal figures. The metrics are ordered according to performance for 100 tests. We use the same ordering for subsequent tables of results for $ITE2_8$. Several sets of metrics give the same results in all cases (we prove they are equivalent for ranking in Section 7.2). The first set (other than O and O^p, which we have already discussed), labelled "Russell etc." is Russell and Rao, Wong1 and Binary. The second is Jaccard, Anderberg, Sørensen-Dice, Dice, Goodman and Kulczynski1. The third is Tarantula and CBI Increase. The fourth is Scott and Rogot1. The last is Rogers and Tanimoto, Simple Matching, Hamann, Sokal, Wong2, M1, Euclid and Hamming (and therefore also Manhattan and Lee).

As expected, Table IV shows the optimal metrics perform equal to or better than all others for all numbers of tests considered. Since more tests provide more information, we would expect the performance to increase as the number of tests grows. This is born out in our results for all metrics except Ample, which has a decrease between two and five tests. With some metrics, such as Rogers and Tanimoto and Jaccard, the performance grows relatively slow. With others, such as Overlap, it grows more quickly. Surprisingly, the simplest metric, Binary (Russell etc), performs extremely well for large numbers of tests. Ample performs particularly poorly for our model. This is because it always ranks $S4$ and $S3$ equally (since $S4$ is executed if and only if $S3$ is not executed). Thus the maximum score the Ample metric can achieve is 0.5 rather than 1. These results are reasonably consistent with previous empirical studies comparing Ochiai, Jaccard, Tarantula and Ample for software diagnosis [Abreu et al. 2007] and Ochiai, Simple Matching and Rogers and Tanimoto for another domain [Meyer et al. 2004].

Table IV. Influence of test suite size and metric on total score (%) for $ITE2_8$

Num. of tests	2	5	10	20	50	100	500	1000
O, O^p	60.00	72.59	81.10	87.98	94.13	96.81	99.31	99.65
Wong3$'$	60.00	67.04	76.51	86.87	94.06	96.81	99.31	99.65
Wong3	56.67	63.15	75.59	86.79	94.06	96.81	99.31	99.65
Zoltar	60.00	71.48	79.97	87.53	94.06	96.80	99.31	99.65
M2	60.00	72.59	80.98	87.34	92.41	94.33	95.87	96.05
Kulczynski2	60.00	71.48	79.72	86.30	91.78	93.96	95.78	95.99
Russell etc	53.33	61.11	69.57	78.79	88.89	93.80	98.64	99.31
Overlap	48.89	54.88	66.57	77.91	88.79	93.79	98.64	99.31
Ochiai	60.00	71.48	79.12	84.90	89.51	91.28	92.75	92.93
Rogot2	56.67	67.78	77.10	83.50	88.42	90.28	91.81	92.00
HMean	48.33	67.13	77.06	83.50	88.42	90.28	91.81	92.00
GMean	48.33	67.13	76.92	83.33	88.23	90.07	91.59	91.78
AMean	48.33	67.13	76.75	83.18	88.01	89.86	91.39	91.58
Ample2	56.67	67.78	76.45	82.79	87.84	89.73	91.28	91.48
Jaccard etc	60.00	71.48	78.22	83.12	87.02	88.55	89.85	90.02
Ochiai2	48.33	67.13	75.13	80.93	85.24	86.89	88.30	88.47
Cohen	48.33	67.13	75.16	80.63	84.75	86.35	87.70	87.87
Tarantula etc	55.56	62.10	69.68	75.68	80.40	82.21	83.73	83.92
Fleiss	56.67	65.93	72.54	76.70	80.10	81.48	82.70	82.86
Scott etc	56.67	66.67	72.43	76.46	79.66	80.95	82.12	82.27
CBI Log	33.33	50.86	63.69	73.13	78.70	80.30	82.35	82.89
CBI Sqrt	28.89	46.73	60.71	70.99	77.16	78.69	79.63	79.73
Rogers etc	56.67	63.15	67.60	71.02	73.93	75.15	76.26	76.41
Ample	36.67	34.54	38.28	41.40	43.92	44.87	45.64	45.74

7.2 Equivalence of ranking metrics

Section 7.1 shows empirically that several sets of ranking metrics are equivalent for $ITE2_8$ with our chosen method of performance evaluation. For some metrics this is obvious: Dice is simply twice Jaccard, for example. For others it is more subtle. For example, the equivalence of the Tarantula metric with the formula for error detection accuracy [Lee et al. 2008] and a_{ef}/a_{ep} was discovered by implementing both metrics in our model and noting the results were identical. Running "real" benchmarks takes significantly more time and effort and since this form of evaluation is not systematic, it is easier to dismiss equal performance of different benchmarks. Here we prove equivalence (for ranking purposes) of some of the other metrics. All but the last of these results are independent of the program (including the number of bugs), the set(s) of tests and the method of performance evaluation.

Lemma 7.1 [Lee et al. 2008]. *A ranking metric $m(\bar{a})$ produces the same rankings as $f(m(\bar{a}))$ if f is a monotonically increasing function.*

Proposition 7.2. *The Jaccard, Anderberg, Sørensen-Dice, Dice, Goodman and Kulczynski1 metrics are all equivalent for ranking.*

Proof. We show they are all equivalent to Kulczynski1, $a_{ef}/(a_{nf} + a_{ep})$. For Jaccard, we can apply the (monotonic) function $f(x) = \frac{1}{(1/x)-1}$ to obtain the same result. For Anderberg, we can apply the function $\frac{2}{(1/x)-1}$. For Sørensen-Dice we can apply $\frac{1}{(2/x)-2}$. For Dice we can apply $\frac{1}{2/x-1}$. For Goodman we can apply

$\frac{1}{(1/x)-1} + \frac{1}{2}$. □

In the following we use F to denote the number of failed tests, P to denote the number of passed tests and T to denote total number of tests; $T = F + P$.

PROPOSITION 7.3. *The Rogers and Tanimoto, Simple Matching, Hamann, Sokal, M1, Wong2, Euclid, Hamming, Manhattan and Lee metrics are all equivalent for ranking.*

PROOF. Simple Matching is equivalent to $(a_{ef} + a_{np})/T$. Applying $x \cdot T - P$ we obtain $a_{ef} - a_{ep}$ (Wong2), since $P = a_{ep} + a_{np}$. Hamann is equivalent to $(2a_{ef} + 2a_{np} - T)/T$ and applying $(x \cdot T + T)/2$ we get the same result. For Rogers and Tanimoto we can apply $\frac{2T}{(1/x)+1} - P$ to get the Simple Matching formula. For Sokal we can apply $\frac{T}{(2/x)-1}$ to get the same result. For M1 we can apply $\frac{1}{(1/x)+1}$ to obtain the Simple Matching formula. Euclid squared is simply the same as Hamming, Manhattan and Lee and applying x/T to Hamming we get Simple Matching. □

PROPOSITION 7.4. *The Scott and Rogot1 metrics are equivalent for ranking.*

PROOF. Taking the Scott formula, using $a_{nf} = F - a_{ef}$, $a_{np} = P - a_{ep}$, multiplying out and simplifying we obtain

$$\frac{-F^2 + (4P + 2F)a_{ef} - 2Fa_{ep} - 2a_{ef}a_{ep} - a_{ep}^2 - a_{ef}^2}{F^2 + 2PF + 2Pa_{ef} + 2Pa_{ep} - 2a_{ef}a_{ep} - a_{ep}^2 - a_{ef}^2}$$

Adding one we obtain

$$\frac{2PF + (6P + 2F)a_{ef} + (2P - 2F)a_{ep} - 4a_{ef}a_{ep} - 2a_{ep}^2 - 2a_{ef}^2}{F^2 + 2PF + 2Pa_{ef} + 2Pa_{ep} - 2a_{ef}a_{ep} - a_{ep}^2 - a_{ef}^2}$$

Taking the Rogot1 formula, using $a_{nf} = F - a_{ef}$, $2a_{np} = P - a_{ep}$, multiplying out and simplifying we obtain this result divided by two. □

PROPOSITION 7.5. *The CBI Increase and Tarantula metrics are both equivalent to $\frac{a_{ef}}{a_{ep}}$ for ranking.*

PROOF. The Tarantula metric is equivalent to $\frac{a_{ef}/F}{a_{ef}/F + a_{ep}/P}$. Applying $\frac{F}{P((1/x)-1)}$ we obtain $\frac{a_{ef}}{a_{ep}}$. CBI Increase is equal to $\frac{a_{ef}}{a_{ef}+a_{ep}} - \frac{F}{T}$. Applying $1/(1/(x + \frac{F}{T}) - 1)$ we obtain the same result. □

This proposition is of particular interest for two reasons. First, it illustrates another advantage of our more formal approach. Our initial implementation of the CBI Increase metric had slightly different performance to Tarantula. Having proved they should be identical the code was examined and a bug in our implementation of CBI Increase was found[1]. Second, the Tarantula metric does not perform particularly well — there are significantly better metrics. This suggests that there is a strong possibility of improving the performance of the CBI system by using a better metric. We are currently investigating this. Alternatively, the role of *Context*

[1]A special case to avoid division by zero returned a value that was not minimal in all cases.

may be particularly important and this could influence the way data on statement executions are best used for diagnosis.

PROPOSITION 7.6. *The Russell and Rao and Wong1 metrics are equivalent for ranking and if there is a single bug they result in the same ranking for the bug as the Binary metric.*

PROOF. The Russell and Rao metric equals a_{ef}/T; multiplying by T gives a_{ef} (which is Wong1). This is maximized for exactly those statements that are executed in all failed tests. There is at least one such statement, the buggy one, if there is a single bug and at least one failed test; if there are no failed tests all statements are ranked equally for all these metrics. The statements that are executed in all failed tests have $a_{nf} = 0$ and thus also maximise the Binary metric. So the same set of statements is maximally ranked in these metrics and includes the bug if there is a single bug. $\square$

Note the non-maximal rankings using Binary can differ from the those using Russell and Rao and Wong1, so for multiple bug programs the performance may differ. Our empirical results in Section 8, which include a small proportion of multiple bug programs, are almost identical for these metrics. O and O^p have a similar relationship, but can give multiple distinct ranks for statements with $a_{nf} = 0$:

PROPOSITION 7.7. *O and O^p give the same rankings to all statements with $a_{nf} = 0$, which includes the bug if there is a single bug.*

PROOF. If $a_{nf} = 0$, O is a_{np} and O^p is $F - a_{ep}/(P+1)$. Also, both metrics return higher values than for any statement with $a_{nf} > 0$. Applying $F - (x - P)/(P + 1)$ to O gives O^p. $\square$

7.3 Error detection accuracy

Error detection accuracy, q_e [Abreu et al. 2006], is defined as the proportion of failed tests in test cases where the buggy statement was executed. That is,

$$q_e = \frac{a_{ef}}{a_{ef} + a_{ep}}$$

It is an indication of how *consistent* a bug is, though it depends on the test set used. Some bugs always result in failure when they are executed (called *deterministic* bugs in [Liblit et al. 2005]) whereas some return the correct result for nearly all tests in which they are executed. We would expect less consistent bugs to be harder to diagnose. This has been studied empirically [Abreu et al. 2006; 2007; Lee et al. 2008]. Here we study it using our model.

Table V gives the performance of the metrics for 100 test cases and several ranges of q_e values. For this (and subsequent) tables, we use strict inequality for the lower bound and non-strict for the upper bound. For example, 0.9–1 means $0.9 < q_e \leq 1$. We generate multisets of paths as before and for each one we determine the q_e value. The second row of the tables gives the percentage of multisets for which the q_e value is in the range, which peaks around 0.5. Multisets where there are no failed tests are not in any of the ranges, so the total percentage is slightly less than 100.

Table V. Influence of error detection accuracy q_e with 100 tests for $ITE2_8$

q_e range	0.00-0.05	0.05-0.10	0.10-0.20	0.20-0.50	0.50-0.90	0.90-1.00
% of multisets	1.06	2.48	8.03	38.90	45.39	3.76
O, O^p	65.48	83.13	91.10	96.68	99.16	99.94
Wong3′	65.46	83.12	91.09	96.67	99.15	99.93
Wong3	65.46	83.12	91.09	96.67	99.15	99.93
Zoltar	65.48	83.13	91.09	96.67	99.15	99.94
M2	65.08	80.34	86.37	92.78	98.04	99.93
Kulczynski2	65.46	82.60	88.59	92.29	97.14	99.89
Russell etc	59.57	76.62	86.71	93.67	96.62	97.33
Overlap	59.57	76.61	86.70	93.66	96.60	97.24
Ochiai	63.11	75.64	80.77	88.32	96.48	99.89
Rogot2	62.39	74.44	79.23	86.79	95.94	99.89
HMean	62.39	74.44	79.23	86.79	95.94	99.89
GMean	62.01	73.82	78.75	86.57	95.79	99.88
AMean	61.65	73.22	78.28	86.35	95.65	99.87
Ample2	62.50	74.76	79.64	86.49	94.92	99.74
Jaccard etc	58.51	67.71	73.19	84.34	95.77	99.89
Ochiai2	58.37	67.28	71.76	81.78	94.61	99.87
Cohen	58.17	66.58	70.97	81.04	94.23	99.85
Tarantula etc	57.98	65.82	69.16	76.79	89.24	98.81
Fleiss	30.41	39.92	52.31	75.82	93.44	99.75
Scott etc	33.01	43.28	53.79	74.01	93.32	99.85
CBI Log	41.08	63.48	69.29	77.03	85.90	92.22
CBI Sqrt	37.04	62.45	69.12	76.71	83.17	88.09
Rogers etc	26.69	31.79	37.64	63.45	93.26	99.87
Ample	31.25	37.38	39.82	43.24	47.46	49.87

The performance of all metrics significantly increases as q_e increases. Although our optimality result does not apply to limited q_e ranges, the optimal metrics perform better than all other metrics in all the cases observed, and the margin between the best metrics and poorer ones is greatest for small q_e values. There are a few notable differences in relative performance compared with Table IV. First, despite good overall performance, Overlap and Russell etc perform relatively poorly for very low and very high q_e values. Second, for low q_e, Ample2 performs relatively well and Kulczynski2 performs better than M2. Third, the ordering of metrics with relatively poor overall performance varies more with different q_e ranges. Some, such as Rogers and Tanimoto perform well for very high q_e values.

7.4 The number of failed tests

The number of failed tests can also effect diagnosis performance, which is what we investigate next. Table VI gives performance of the metrics with 100 test cases and various ranges for the proportion of tests that fail. Due to the very small percentage of multisets, figures in the the last column may be inaccurate in the last two digits.

Overall, these results show similar patterns to Table V, with the optimal metrics performing best in all the cases. This is to be expected since q_e is related to the proportion of failures. However, it does show that some metrics, such as Tarantula and Ample, are less sensitive to the proportion of failures than others. Also, Tarantula and its equivalent CBI Increase and the variants CBI Log and CBI Sqrt all peak at around 0.1 to 0.5 rather than monotonically increasing. Performance of

Table VI. Influence of proportion of failed tests with 100 tests for $ITE2_8$

Failure range	0.00-0.05	0.05-0.10	0.10-0.20	0.20-0.50	0.50-0.90	0.90-1.00
% of multisets	6.24	11.02	26.54	49.27	6.56	0.002
O, O^p	86.86	94.23	96.79	98.36	99.10	99.37
Wong3′	86.85	94.22	96.79	98.35	99.09	99.17
Wong3	86.85	94.22	96.79	98.35	99.09	99.17
Zoltar	86.86	94.22	96.78	98.34	99.08	99.26
M2	86.34	92.11	93.91	95.64	97.41	99.16
Kulczynski2	86.76	93.17	94.14	94.67	96.05	98.77
Russell etc	74.36	88.76	93.76	96.82	98.30	98.95
Overlap	74.36	88.75	93.75	96.81	98.23	94.68
Ochiai	84.28	88.73	90.44	92.64	95.44	98.77
Rogot2	84.25	88.27	89.48	91.37	94.47	97.35
HMean	84.25	88.27	89.48	91.37	94.47	97.35
GMean	83.65	87.81	89.34	91.27	93.85	96.54
AMean	83.05	87.35	89.20	91.18	93.26	95.90
Ample2	84.60	88.96	89.96	90.38	90.11	85.53
Jaccard etc	80.73	84.34	86.82	90.58	94.83	98.77
Ochiai2	81.26	84.30	85.71	88.13	92.07	96.42
Cohen	80.34	83.35	85.04	87.79	91.47	95.98
Tarantula etc	79.87	82.01	82.46	82.47	81.74	73.29
Fleiss	66.19	73.92	79.77	85.10	88.45	90.48
Scott etc	67.72	73.92	78.51	84.26	90.36	96.50
CBI Log	75.57	82.44	82.95	79.71	74.89	69.95
CBI Sqrt	74.42	82.33	83.04	77.18	70.41	65.88
Rogers etc	58.41	62.60	68.95	81.07	92.77	98.50
Ample	42.30	44.48	44.98	45.19	45.05	42.77

Overlap and Ample2 also decreases slightly for high proportions of failures.

In previous work [Abreu et al. 2007] there is empirical evidence that around 5 failed test cases is sufficient for good performance of Ochiai, and increasing the number of passed tests beyond 20 results in only minor improvement in performance. We reproduce this experiment, using 4–6 failed tests, and all the metrics. The results are given in Table VII. For each number of tests we give the percentage of multisets with 4–6 failed tests (as expected, this decreased as the number of tests increases). The results are consistent with those of [Abreu et al. 2007] — most metrics show only a modest performance increase. The performance of a few of the poorer metrics actually decreases with more passed tests. As with Table VI, when the number of tests is large (and thus the proportion of failed tests is small) Ample2 performs well, Russell etc. and Overlap perform relatively poorly and Kulczynski2 performs better than M2.

7.5 Buggy code execution frequency

If buggy code is rarely executed bugs can also be hard to diagnose. Here we investigate the relationship between diagnostic performance and the proportion of tests that execute the buggy statement ($S4$). To our knowledge, this has not been studied previously. Note that we only consider cases in which there is a failed test case, so $S4$ must be executed at least once. The number of executions of the buggy statement, error detection accuracy and failure rate are related: the number of failures is the number of tests in which $S4$ is executed times q_e. That is, the

Table VII. Influence of passed tests with 4–6 failed tests for $ITE2_8$

Num. of tests	10	20	50	100	500
% of multisets	35.94	33.79	13.18	4.53	0.280
O, O^p	86.29	88.28	89.35	89.68	91.58
Wong3′	81.09	87.18	89.26	89.67	91.58
Wong3	81.09	87.18	89.26	89.67	91.58
Zoltar	84.71	87.85	89.30	89.67	91.58
M2	86.07	87.80	88.68	88.95	90.33
Kulczynski2	84.23	86.93	88.89	89.53	91.57
Russell etc	78.45	79.28	79.72	79.87	83.29
Overlap	75.02	78.83	79.70	79.86	83.29
Ochiai	83.13	84.88	85.94	86.29	87.39
Rogot2	80.58	83.51	85.47	86.25	87.88
HMean	80.61	83.51	85.47	86.25	87.88
GMean	80.54	83.52	85.07	85.54	86.68
AMean	80.21	83.58	84.62	84.85	85.64
Ample2	79.84	83.99	86.08	86.70	88.06
Jaccard etc	81.90	82.31	82.18	82.05	82.37
Ochiai2	78.05	80.58	82.09	82.61	83.53
Cohen	77.98	80.19	81.24	81.56	82.24
Tarantula etc	70.31	76.84	80.01	80.96	82.09
Fleiss	75.89	76.52	72.52	67.90	60.70
Scott etc	75.52	75.28	72.66	69.28	62.43
CBI Log	75.32	79.35	81.23	81.81	82.37
CBI Sqrt	72.68	77.70	80.33	81.11	82.13
Rogers etc	73.00	67.40	61.52	59.02	57.03
Ample	39.94	41.99	43.04	43.35	44.03

number of executions of $S4$ is the failure rate divided by q_e. The results are in Table VIII.

Perhaps surprisingly, metrics other than Russell etc., Overlap, the CBI variants and Ample perform extremely well when $S4$ is executed only a few times, and performance decreases with greater execution frequency. For O the decrease is only slight but it is substantial for the poorer metrics (particularly Fleiss). Wong3 (and Wong3′) performs worse than most metrics for a small number of $S4$ executions (though still extremely well). Russell etc. and Overlap exhibit the opposite trend — performance increasing as the frequency increases. For very high frequencies these metrics perform better than O. This is the only case we have discovered where O does not perform the best. When $S4$ is executed in a large proportion of the tests, the a_{np} component of the O formula is misleading. It is typically very small for $S4$ but larger for $S1$ and $S2$. There are not many test cases for which a_{nf} is zero for $S1$ or $S2$, but there are enough to reduce the performance of O below that of Russell etc. and also Overlap.

7.6 Comparison of metrics

It would be naive to extrapolate results from our simple model to diagnosis of real programs without further evidence. Nevertheless, our experiments suggest the following. The Optimal metrics are best overall, and are also quite robust. They perform best in nearly all the cases we have investigated. Zoltar performs almost as well as O in all cases and Wong3 (and Wong3′) performs similarly when the

Table VIII. Influence of buggy code execution frequency with 100 tests for $ITE2_8$

$S4$ exec range	0.00-0.05	0.05-0.10	0.10-0.20	0.20-0.50	0.50-0.90	0.90-1.00
% of multisets	0.063	0.432	3.74	46.51	48.49	0.389
O, O^p	99.89	99.75	99.42	97.92	95.54	94.39
Wong3′	99.86	99.73	99.41	97.91	95.54	94.35
Wong3	99.86	99.73	99.41	97.91	95.54	94.35
Zoltar	99.89	99.75	99.42	97.91	95.53	94.24
M2	99.89	99.75	99.42	97.55	90.92	77.20
Kulczynski2	99.89	99.74	99.37	97.20	90.52	75.18
Russell etc	66.53	77.61	85.87	92.74	95.59	96.94
Overlap	66.52	77.60	85.86	92.73	95.57	96.70
Ochiai	99.89	99.73	99.31	96.31	85.97	65.01
Rogot2	99.89	99.74	99.35	96.40	83.92	52.69
HMean	99.89	99.74	99.35	96.40	83.92	52.69
GMean	99.89	99.73	99.30	96.15	83.77	46.52
AMean	99.89	99.72	99.24	95.90	83.62	41.35
Ample2	99.89	99.75	99.42	96.95	82.43	30.21
Jaccard etc	99.89	99.72	99.21	95.27	81.43	55.96
Ochiai2	99.89	99.73	99.28	95.56	77.98	25.44
Cohen	99.89	99.72	99.19	94.91	77.44	31.91
Tarantula etc	99.86	99.65	98.95	93.26	70.65	18.67
Fleiss	99.89	99.75	99.41	94.31	68.12	16.85
Scott etc	99.89	99.72	99.16	93.38	67.93	19.62
CBI Log	78.13	92.50	96.70	91.94	68.26	18.47
CBI Sqrt	78.13	92.47	96.66	90.77	66.08	17.66
Rogers etc	99.84	99.52	98.37	88.60	60.56	30.54
Ample	49.94	49.88	49.71	48.48	41.22	15.11

numbers of tests is reasonably large. Although Russell etc. and Overlap are next in overall performance for larger numbers of tests and can perform better than O when the bug is executed in a large proportion of the tests, they are less robust. For small numbers of tests, q_e values and proportion of failed tests, their performance is significantly worse. In contrast, although M2 and Kulczynski2 perform somewhat worse than these metrics overall for large numbers of tests, they are more robust and our experiments suggest they may be better metrics to use in practice (and for small q_e and proportion of failed tests Kulczynski2 is the better of the two). Several of the other metrics also perform very well in some circumstances. However, even the ones with higher profiles such as Tarantula, Jaccard and Ochiai (the best of these) do not have particularly good overall performance, and when they perform very well they are still not as good as the best metrics. A few metrics perform particularly poorly.

The reason why Russell and Rao, Binary, Wong1 and Overlap behave somewhat differently from the other metrics can be understood by considering the two conditions for optimality. Recall metrics are optimal if they 1) increase as a_{np} increases when $a_{nf} = 0$ and 2) always have smaller values when $a_{nf} > 0$. When $a_{nf} = 0$ Russell etc. and Overlap are minimal but constant, rather than increasing in a_{np} (satisfying 2 but not 1). All other non-optimal metrics we consider are increasing in a_{np} when $a_{nf} = 0$ (this is straightforward to prove from the formulas) but do not always have smaller values when $a_{nf} > 0$ (satisfying 1 but not 2). Further insight into the relative performance of the metrics can be obtained by constructing formulas that are equivalent to the metrics for ranking purposes, but are easier to compare. Specifically, we can compare with formulas that are known to be optimal. In Table IX we provide formulas that (for ranking) are equivalent to a selection of the formulas given earlier.

PROPOSITION 7.8. *Table IX gives metrics that, for ranking, are equivalent to the definitions in Tables II and III, assuming a single bug and our* score *function in the case of Binary.*

PROOF. The first optimal metric equivalent formula is O^p, given earlier. The second optimal formula is equivalent to $a_{ef}/(a_{ep} + P \cdot F)$. If $a_{nf} = 0$ this equals $F/(a_{ep} + P \cdot F)$ which increases as a_{ep} decreases and is greater than the value of the metric for $a_{nf} > 0$ since the denominator is between $P \cdot F$ and $P \cdot (F + 1)$. Note that the term $P \cdot F$ can be replaced by any larger term. The rest are either given by the previous propositions or are straightforward. □

Table IX. Equivalent formulas for the ranking metrics used

Metric(s)	Equivalent Formula
Optimal (O^p)	$a_{ef} - \frac{1}{P+1} a_{ep}$
Russell etc	a_{ef}
Rogers etc	$a_{ef} - a_{ep}$
Ample2	$a_{ef} - \frac{F}{P} a_{ep}$
Ample	$\left\lvert a_{ef} - \frac{F}{P} a_{ep} \right\rvert$
Kulczynski2	$a_{ef} - \frac{F}{a_{ep}+a_{ef}} a_{ep}$
Optimal	$\log a_{ef} - \log(a_{ep} + P.F)$
Tarantula etc	$\log a_{ef} - \log a_{ep}$
Jaccard etc	$\log a_{ef} - \log(a_{ep} + F)$
M2	$\log a_{ef} - \log(a_{ep} + 2F + P)$
Ochiai	$\log a_{ef} - \frac{1}{2} \log(a_{ep} + a_{ef})$

O^p is similar to Wong3, but does not have three separate cases with different a_{ep} coefficients: one very small coefficient (its maximum size dependent on the number of passed tests) is optimal and the other cases just decrease performance in the model. Note that the smallest a_{ep} coefficient in Wong3 is also sub-optimal for more than 1000 tests. With larger numbers of test cases we see the performance of Wong3 again drops noticeably below that of O and Zoltar. For 5000 tests O and Zoltar have a percentage score of 99.93 compared with 99.91 for Wong3.

Comparing the O^p formula with those for Rogers etc. and Russell etc. we see that Russell etc. neglects the a_{ep} term whereas Rogers etc. gives it too much influence. For a large number of passed tests, this means the performance of Russell etc. approaches that of O^p (since the optimal coefficient of a_{ep} approaches zero). Conversely, Rogers etc. is closer to O^p for very small numbers of passed tests. For large numbers of tests Rogers etc. is one of the worst metrics because of its relatively large coefficient for a_{ep}. It is particularly bad when the failure rate, q_e or $S4$ are small (since a_{ep} is relatively large compared to a_{ef} in these cases).

The performance of Ample2 is particularly sensitive to the failure rate. For a single failed test it is almost as good as O^p (the a_{ep} coefficient is $1/(P+1)$) but when the failure rate is more than 50% it is even worse than Rogers etc. (the a_{ep} coefficient is more than 1). In practice, failure rates are typically quite small, helping the performance of Ample2. The Ample use of absolute value makes it less like O^p; this is what prompted us to experiment with Ample2, which indeed out-performs Ample. Kulczynski2 performs better than Ample2, particularly for a high number of failures, for the following reason. The Kulczynski2 a_{ep} coefficient is similar to that of Ample2 but the denominator is the number of tests in which the statement was executed, rather than P. For the buggy statement this will be between F and $F+P$ whereas for correct statements it will tend to be lower, lowering the rank of these statements.

From the formulas with logarithms it can be seen that M2 is better than Jaccard, which is better than Tarantula etc. due to the relative influence of a_{ep}. Jaccard gets close to optimal for very high failure rates and M2 gets close to optimal for both very high and very low failure rates. For Ochiai the use a the square root (leading to the factor of $\frac{1}{2}$) is helpful to performance (using a cube root et cetera would be even better as it would reduce the influence of a_{ep} even more).

We have not found a formula equivalent to Zoltar that is easily comparable with an optimal formula. When $a_{nf}=0$ the Zoltar metric is $a_{ef}/(a_{ef}+a_{ep})$, which is increasing in a_{np} but may be smaller than some cases when $a_{nf}>0$. For the buggy statement $S4$, the value is $F/(F+a_{ep}^{S4})$. Due to the large factor of 10000 it is rare for the value for $S1$ or $S2$ to be higher unless $a_{ep}=0$. Even when $a_{ep}=0$ we still need $a_{ef}/(a_{ef}+a_{nf}) > F/(F+a_{ep}^{S4})$, which requires a low value of a_{nf} and a high value of a_{ep}^{S4}. That is, a correct statement ($S1$ or $S2$) is used in no passed tests but many failed tests and the buggy statement is used in many passed tests. For example, with five tests, three of which are failed, $S4$ used in both passed tests and $S1$ used in two failed tests and no passed tests, the Zoltar metric for $S1$ is (just) larger than that for $S4$ (10/15 compared with 9/15). Such cases form only a tiny fraction of the multisets for larger numbers of tests.

7.7 Other models

We have conducted many experiments with other models. Here we briefly report some of the results, which may be expanded on in the future. Increasing the number of paths while retaining the same code increases the number of possible multisets of test cases. This has little effect on the relative performance of the different metrics but does increase absolute performance. This is because a greater proportion of the multisets have a relatively even distribution across the different paths (for example, the proportion of multisets that contain just one path becomes much smaller). Increasing the number of sources of “noise” (adding extra correct if-then-else statements) with a fixed number of paths decreases performance, as expected. Promisingly, the decrease we have observed is less than linear. Decreasing the proportion of paths through the buggy statement that lead to failure (thus decreasing the average q_e value) also decreases performance. In all the cases above, O appears to be optimal.

We have discovered some classes of models where O is not optimal according to

Table X. Description of Siemens Test Suite and Space

Program	Versions	LOC	Test Cases	Description
tcas	41	173	1608	altitude separation
schedule	9	410	2650	priority scheduler
*schedule*2	10	307	2710	priority scheduler
print_tokens	7	563	4130	lexical analyser
*print_tokens*2	10	508	4115	lexical analyser
tot_info	23	406	1052	information measure
replace	32	563	5542	pattern recognition
Space	32	9059	13585	array definition language

the definition we have given. If the buggy statement is (almost) always executed then metrics such as Russell etc. can perform better than O. In Table VI we have this result for just a subset of the multisets possible with $ITE2_8$, but by changing the model it can occur with the *total score*. For such models we believe it is appropriate to generalise our definition of optimality. Rather than fix a specific statement as being buggy, we could consider the possibility of each statement being buggy, and average over all these cases. For $ITE2_8$ the two definitions are equivalent due to the symmetry of the code. With such a revised definition O still appears optimal (for Russel etc. the performance is excellent when the frequently executed statement is buggy but poor for when the statement is correct).

For models with multiple bugs we see a divergence between the performance of O and O^p. O^p and the other good metrics other than O generally perform quite well, especially when most failures are caused by a single bug. In some cases Ochiai and even Jaccard perform better. None of the metrics examined is optimal for more than a very small number of tests. We have constructed optimal metrics for some cases. It may be that by constructing optimal metrics in other cases some patterns will emerge, allowing us to find optimal or near optimal metrics in a broad range of cases.

8. EMPIRICAL RESULTS

We use the Siemens test suite and Space benchmarks to perform empirical evaluation of how well the metrics perform. The Siemens test suite contains multiple versions of seven programs; Space has multiple versions of a single larger program. Table X lists the programs, the number of faulty versions of each program, number of lines of code, number of test cases and brief descriptions. The versions of the test suites we used were obtained from the Software Information Repository [Do et al. 2005]. The Siemens test suite is a particularly widely used benchmark for bug localization [Renieres and Reiss 2003; Wong et al. 2007; Abreu et al. 2006; 2007; Liu et al. 2005; Pytlik et al. 2003].

Program spectra were extracted using `gcov` (part of the `gcc` compiler suite) and Bash scripts. All experiments were carried out on a Pentium 4 PC running Ubuntu 6.06 and `gcc` version 4.2.1. There are six Siemens test suite programs that have runtime errors for some tests and `gcov` fails to produce any execution counts; we have obtained data from other researchers for these cases. There were also six (of an original 38) Space programs for which we failed to obtain data; we have excluded these programs from the benchmark set. `gcov` reports execution statistics for each line of source code, including lines that are empty or contain only preprocessor directives such as `#define`, comments, braces, et cetera. For one set of results (Tables XI and XII) we use the `gcc` statistics directly, rather than attempting to filter out lines such as these. For the others we ignore lines that are not executed in any tests.

We conducted experiments using the same metrics as previously reported in our $ITE2_8$ model. As well as figures for the entire Siemens test suite and Space, we report on various subsets. As in other papers, we report results for the (versions of the) individual different programs. We also consider the subset of programs that fail for at least one test case. Without failed test cases all of the metrics perform very poorly and this just adds noise to the results. Though potentially useful for debugging tools that use static analysis, these programs and test sets are not appropriate for dynamic diagnosis (there are no symptoms to diagnose). There are 130 Siemens test suite programs and 28 Space programs that fail at least one test case.

We also report on a subset of the suite that allows a better comparison with our model, and suits the main focus of this paper. We use the set of programs that fail a test case and have a single bug according to the following definition: a bug is a statement that, when executed, has unintended behaviour. This definition is motivated by programming language semantics and seems particularly apt for spectra-based diagnosis. Not all the Siemens test suite programs have a single bug according to this definition (though they are single bug programs according to other definitions). For example, some programs have a (single) `#define` that is wrong. The `#define` is never executed; it is only when a statement that uses the macro is executed that unintended behaviour occurs. There are typically several such statements. There are 122 "single bug" Siemens test suite programs and 15 such Space programs.

We provide experimental results using two established measures of diagnostic performance for the different ranking methods. The first is the percentage of the program that needs to be examined for a bug to be found. That is, the percentage of the code ranked higher than any bug. We call this the *rank percentage* as it corresponds to the rank of the buggy statement. We average the percentages over all programs in the test suite considered. Unlike the other performance measures we use, larger numbers mean *worse* performance. The second is whether diagnosis is successful (a bug can be found) by examining some fixed percentage of the program. We report the percentage of successful diagnoses over the test suite. For example, suppose we choose to examine at most 10% of the statements of each of the 122 single-bug Siemens test suite programs and for some metric the top ranked 10% of statements include the bug for 70 of the programs. We would report $70/122 \times 100 = 57.38$.

In the case where the buggy statement is tied in the ranking (the value of the metric for the buggy statement is the same as that for some other statements), we assume the bug can be found anywhere in that range, with uniform probability, and reflect this in our performance measures. For example, suppose the top 10% of statements are all ranked equally and include the bug. The rank percentage would be computed as 5%, as in [Abreu et al. 2007]. Successful diagnosis by examining

Table XI. Average Rank Percentages for programs of Siemens Test Suite

	Tcas	Sch	Sch2	PrT	PrT2	TInf	Repl
O^p	10.27	3.94	23.23	3.42	0.82	3.13	4.73
O	10.27	7.72	23.23	3.42	0.82	3.13	4.73
Zoltar	10.27	3.94	20.80	3.42	0.82	3.15	3.97
Kulczynski2	10.31	3.94	20.83	3.42	1.12	3.29	4.32
Wong3′	10.48	3.94	17.35	3.42	0.86	3.13	3.99
M2	10.64	1.64	23.19	4.33	2.04	4.61	4.54
Ochiai	11.01	1.69	23.58	6.23	3.73	5.53	4.95
Jaccard etc	11.13	1.75	26.05	8.38	5.57	6.55	6.29
Amean	11.18	5.22	23.91	7.32	5.12	9.38	5.29
Tarantula etc	11.15	1.83	26.05	8.94	5.73	7.10	6.49
Hmean	11.27	5.03	25.34	7.75	4.95	9.24	5.06
Gmean	11.28	5.03	25.14	7.85	4.95	9.32	5.09
Ample2	11.27	5.06	28.42	7.73	4.97	9.39	5.76
Cohen	11.15	5.30	26.05	8.79	5.73	10.10	6.44
CBI Log	11.83	7.29	26.05	8.89	5.73	10.28	6.29
CBI Sqrt	11.84	7.35	26.05	8.89	5.73	10.29	6.49
Rogot2	11.27	10.92	27.77	7.75	4.95	14.39	5.84
Russell etc	14.60	11.34	18.53	7.17	10.66	6.67	9.13
Binary	14.60	15.12	18.53	7.17	10.66	6.67	9.13
Overlap	14.60	11.34	18.53	7.17	10.66	6.70	9.14
Ochiai2	11.69	7.29	28.45	8.23	5.50	14.13	6.02
Ample	13.19	12.70	27.48	8.33	5.50	15.08	6.95
Wong3	18.17	17.82	29.14	3.42	0.86	9.30	11.34
Scott etc	73.81	71.25	86.14	44.89	38.62	74.97	52.57
Fleiss	73.93	71.25	86.14	44.84	38.19	75.01	54.88
Rogers etc	78.93	71.44	86.14	65.02	52.61	75.05	71.74

10% of the program is assured (we give a value of 1). However, if only 8% of the program is examined successful diagnosis is likely but not assured. We give a value of 0.8 to indicate the probability. We report the sum of these probabilities times 100 divided by the number of programs in the test suite. Alternative ways of measuring performance with ties are to report the worst case [Abreu et al. 2006] (performance of a metric can be increased according to this measure by adding a small random number) or both the best and worst cases [Wong et al. 2007] (which requires twice as many figures).

8.1 Rank percentages

Table XI shows the average of rank percentages, calculated using all lines of source code, for the different programs of the Siemens test suite using different metrics. Table XII shows average rank percentages for all programs, those with at least one failed test and "single bug" programs (in columns labelled "All", "Fail" and "1Bug", respectively) of the Siemens test suite, Space and combination of the two benchmark sets. Table XIII gives these percentages calculated using just lines of code that are executed. This can have a significant effect on relative performance and is what prompted us to introduce the Wong3′ metric (in several programs the Wong3 metric as defined in [Wong et al. 2007] ranks the buggy statement below code that is never executed in any test case). In [Wong et al. 2007], χsud [Telcordia Technologies, Inc. 1998] is used to extract the spectra and this apparently filters out some information

Table XII. Average Rank Percentages for Siemens Test Suite and Space

Benchmark	Siemens			Space			Combined		
Subset	All	Fail	1Bug	All	Fail	1Bug	All	Fail	1Bug
O^p	7.15	6.11	5.90	9.77	0.70	0.50	7.66	5.15	5.31
O	7.41	6.37	5.90	9.77	0.70	0.50	7.87	5.37	5.31
Zoltar	6.79	6.12	5.91	6.90	0.69	0.55	6.81	5.16	5.32
Kulczynski2	6.94	6.27	6.06	6.94	0.79	0.63	6.94	5.30	5.47
Wong3′	6.60	6.45	6.14	9.77	0.70	0.50	7.22	5.43	5.52
M2	7.46	6.80	6.81	6.91	0.76	0.58	7.35	5.73	6.13
Ochiai	8.10	7.45	7.47	6.97	0.82	0.67	7.88	6.28	6.73
Jaccard etc	9.08	8.45	8.68	7.24	1.13	1.06	8.72	7.15	7.85
Amean	9.33	8.70	8.88	7.21	1.09	0.93	8.92	7.35	8.01
Hmean	9.38	8.76	8.96	7.13	1.01	0.84	8.94	7.39	8.07
Gmean	9.40	8.77	8.97	7.13	1.00	0.83	8.96	7.39	8.08
Tarantula etc	9.28	8.65	8.89	7.81	1.78	1.88	8.99	7.43	8.12
Ample2	9.82	8.81	9.02	10.06	1.02	0.82	9.87	7.43	8.12
Cohen	10.02	9.40	9.69	7.40	1.31	1.26	9.51	7.97	8.77
CBI Log	10.36	9.75	10.11	3.83	1.73	1.82	9.09	8.33	9.20
CBI Sqrt	10.42	9.81	10.12	7.75	1.71	1.83	9.90	8.37	9.21
Rogot2	11.06	10.07	10.37	10.04	1.01	0.84	10.86	8.46	9.33
Ochiai2	11.06	10.46	10.74	7.70	1.66	1.64	10.40	8.90	9.74
Russell etc	11.27	10.68	10.51	11.39	5.88	5.35	11.29	9.83	9.95
Binary	11.53	10.94	10.51	11.39	5.88	5.35	11.50	10.04	9.95
Overlap	11.28	10.68	10.52	11.46	5.96	5.46	11.32	9.84	9.97
Wong3	13.68	12.74	12.36	9.77	0.70	0.50	12.92	10.61	11.06
Ample	12.22	12.01	12.22	3.55	2.62	2.28	10.53	10.35	11.13
Scott etc	65.42	65.28	65.67	21.04	13.58	18.37	56.76	56.12	60.49
Fleiss	65.99	65.85	66.27	21.03	13.57	18.47	57.22	56.59	61.04
Rogers etc	73.82	73.80	74.15	45.45	41.47	43.07	68.28	68.07	70.75

`gcov` reports, leading to different results. Our model program could be modified to include statements that are never executed, or debugging systems could *a priori* exclude such statements from consideration, as we have done in Table XIII (making Wong3 and Wong3′ equivalent).

The relative performance of the different metrics on the single bug subset of the combined test suite in Table XIII fits reasonably well with the overall results of our model, with the optimal metrics performing best (they perform best for all single bug subsets and all performance measures we present). The most significant difference is that Russell etc, Binary and Overlap perform more poorly than the overall performance in our model. However, the model predicts their relatively poor performance for low q_e and proportion of failed tests. In the Siemens test suite, the average q_e value is approximately 0.12 and the average proportion of failed tests is 0.035. The top six ranked metrics (treating O and O^p as the same metric and similarly Wong3 and Wong3′) in Table XIII are *identical* (modulo equal ranking) to those ranked according to the first column of Tables V and VI.

Performance of Russell etc, Binary and Overlap is significantly affected by statements that are executed in all test cases, such as initialisation code. These are always ranked equal-highest. Adding such a statement in our model program approximately halves their model performance. The simple score function we use is also kind to these metrics, especially Binary. If Binary doesn't rank the bug equal-

Table XIII. Average Rank Percentages — executed lines of code only

Benchmark	Siemens			Space			Combined		
Subset	All	Fail	1Bug	All	Fail	1Bug	All	Fail	1Bug
O^p	16.91	16.57	16.00	10.94	2.03	1.62	15.75	13.99	14.43
O	16.96	16.62	16.00	10.94	2.03	1.62	15.79	14.03	14.43
Zoltar	17.11	16.60	16.03	8.03	2.03	1.77	15.34	14.02	14.47
Kulczynski2	17.50	16.99	16.44	8.26	2.30	2.02	15.70	14.39	14.86
Wong3	17.75	17.42	16.60	5.07	2.03	1.62	15.28	14.69	14.96
M2	19.03	18.55	18.59	8.18	2.21	1.87	16.91	15.65	16.76
Ochiai	20.87	20.41	20.48	8.33	2.38	2.13	18.42	17.21	18.47
Amean	22.19	21.76	22.12	8.99	3.13	2.84	19.61	18.46	20.01
Hmean	22.31	21.88	22.34	8.79	2.90	2.61	19.67	18.52	20.18
Gmean	22.37	21.94	22.38	8.77	2.87	2.56	19.72	18.56	20.21
Ample2	22.31	22.05	22.52	11.74	2.94	2.55	20.25	18.66	20.33
Jaccard etc	23.54	23.12	23.79	9.07	3.23	3.20	20.72	19.60	21.54
Rogot2	23.81	23.58	24.15	11.69	2.90	2.61	21.45	19.92	21.79
Tarantula etc	24.15	23.74	24.44	11.01	5.44	6.23	21.59	20.50	22.45
Cohen	24.04	23.64	24.33	9.51	3.72	3.76	21.20	20.11	22.08
CBI Log	24.57	24.17	25.01	10.87	5.28	6.02	21.90	20.82	22.93
CBI Sqrt	24.72	24.32	25.06	10.79	5.18	5.98	22.00	20.93	22.97
Ochiai2	27.33	26.97	27.69	10.33	4.66	4.77	24.01	23.02	25.18
Binary	29.47	29.14	28.64	21.48	17.41	17.32	27.91	27.06	27.40
Russell etc	29.42	29.10	28.64	21.48	17.41	17.32	27.87	27.03	27.40
Overlap	29.45	29.12	28.67	21.82	17.80	17.94	27.96	27.11	27.50
Ample	31.63	31.17	31.71	7.67	7.33	6.63	26.95	26.95	28.96
Scott etc	41.07	41.10	42.25	20.11	12.51	15.84	36.98	36.03	39.36
Fleiss	41.39	41.43	42.55	20.08	12.48	16.12	37.23	36.30	39.66
Rogers etc	43.62	43.69	44.80	27.33	20.77	25.36	40.44	39.63	42.67

top it ranks it equal-bottom; other good metrics that fail to rank it equal-top will often rank it strictly above most correct statements but the score will be the same. Several other metrics (Fleiss, Scott etc, Rogot2 and Wong3) are also slightly lower in the ranking than in our model. For Fleiss and Scott etc. this is to be expected since they perform poorly for low q_e and proportion of failed tests. Kulczynski2 is slightly higher (it performs relatively well for low q_e and proportion of failed tests) and Ample performs rather better than predicted by the model. If the bug occurs in an if-then-else statement the Ample metric typically gives both the then and else statements the same reasonably high rank. Since this is still only a small fraction of the code (as opposed to half the code in our model) the performance overall is reasonable. Experiments using different models indicate than as the number of statements increases the difference in performance between Ample and the better metrics decreases.

Another interesting observation is there are 30 Siemens test suite programs where Russell etc. perform better than O, despite relatively poor performance overall. In all these cases, the buggy statement is executed in at least 82% of total test cases. This is consistent with our model (see Table VIII) — when $S4$ is executed in 0.50–0.90 of tests, Russell outperforms O and is the best of all metrics considered.

When multiple-bug programs are included, we see O^p still performs the best, but the performance of O declines more, as expected. Results for all programs in the test suites are influenced by the programs with no failed test cases. This can be

Table XIV. Percentage of successful diagnoses for "single bug" programs

Code Examined	1%	2%	4%	6%	8%	10%	20%	50%
O, O^p	17.09	29.61	47.07	56.74	60.38	63.76	74.12	91.15
Zoltar	16.36	29.61	46.90	56.38	60.38	63.09	74.12	91.15
Wong3	16.36	29.61	47.07	56.74	60.38	63.76	74.12	90.05
M2	15.55	28.04	42.75	53.24	56.20	58.69	70.63	88.86
Kulczynski2	14.87	28.12	45.15	55.54	58.87	61.51	73.09	91.07
Ample2	14.19	25.50	40.15	49.82	52.47	54.61	63.74	85.09
Ochiai	14.12	25.11	39.65	49.82	53.56	55.66	64.86	87.74
Rogot2	14.01	25.34	39.09	48.92	51.91	53.76	61.88	83.47
Hmean	14.01	25.34	39.09	49.65	52.64	54.49	63.70	85.66
Gmean	13.96	24.97	38.73	48.19	52.60	54.31	63.56	85.12
Amean	13.60	23.39	36.50	46.73	50.99	52.93	63.47	85.74
Ochiai2	13.57	22.13	36.04	42.93	48.78	51.84	58.96	78.80
Ample	13.40	21.85	32.45	38.07	42.84	45.49	56.38	74.50
Jaccard etc	13.20	21.61	34.49	41.47	46.19	49.23	62.31	85.13
Cohen	13.20	21.59	33.93	41.47	44.80	47.12	61.34	84.40
Fleiss	13.11	17.58	23.96	25.77	27.37	28.23	37.96	63.04
CBI Log	12.61	19.40	31.91	39.32	43.13	45.18	58.64	84.26
CBI Sqrt	11.93	19.27	31.20	39.01	43.13	45.18	58.82	84.18
Tarantula etc	11.38	19.33	31.52	39.58	44.03	46.39	59.68	84.40
Rogers etc	9.53	12.06	13.16	14.82	16.26	17.28	29.73	62.25
Scott etc	5.31	13.94	19.16	23.47	25.07	28.83	30.42	38.32
Russell etc	1.85	4.50	9.49	14.59	19.85	24.38	44.53	84.91
Binary	1.85	4.50	9.49	14.59	19.85	24.38	44.53	84.91
Overlap	1.82	4.47	9.43	14.47	19.71	24.21	44.30	84.77

seen particularly for Schedule (column two of Table XI), for which four out of nine versions pass all tests, and Space. The relative performance of metrics in these cases is quite different from the overall pattern.

8.2 Successful diagnosis percentages

Table XIV shows the percentage of single bug programs (in the combined benchmark set) that are successfully diagnosed as the percentage of the code examined increases; lines of code that are not executed are ignored. Again, the results fit well with our model for the smaller percentiles. The performance measure used with our model only considers the top-ranked statements — higher percentiles are ignored. Using average rank percentages arguably gives *too much* weight to high percentiles (where Wong3 performs worse than Kulczynski2, for example). There is little practical difference between a percentage ranking of 33% and 99% for the bug, since in both cases the programmer is almost sure to abandon this diagnosis method before the bug is found. A weighted average of rank percentages (giving more weight to lower percentages) would make a better practical measure of performance. We have experimented with such scoring functions with our model. The optimal metrics perform best in all cases examined. However, *proving* optimality for these more complex scoring functions is much more difficult than for the simple scoring function.

Table XV gives the corresponding results for the complete Siemens test suite, which allows a fairer comparison with diagnosis based on other techniques (albeit with some noise from programs with no failed test cases). The last line of the table

Table XV. Percentage of successful diagnoses for Siemens Test Suite

Code Examined	1%	2%	4%	6%	8%	10%	20%	50%
Zoltar	11.13	21.88	38.47	48.24	54.12	57.83	69.19	88.97
O	11.11	21.86	38.60	48.54	54.00	57.69	68.89	89.73
O^p	11.11	21.86	38.60	48.54	54.00	57.69	68.89	89.73
Wong3	10.36	21.86	38.60	48.54	54.00	57.69	68.89	87.84
Kulczynski2	9.61	20.37	36.59	47.26	52.43	56.15	68.13	88.89
M2	9.57	21.04	34.86	45.56	49.59	52.76	66.32	87.35
Ochiai	8.91	18.78	31.78	41.97	45.89	48.67	60.33	86.13
Ample2	8.90	19.13	33.61	43.40	47.03	49.33	58.87	84.53
Hmean	8.76	19.00	32.72	43.30	47.33	49.35	59.13	84.36
Gmean	8.76	18.63	32.34	41.78	46.53	49.16	58.99	83.80
Rogot2	8.75	18.98	32.66	42.45	46.46	48.44	56.94	82.85
Ample	8.52	17.40	26.55	31.96	37.80	40.62	51.18	71.27
Amean	8.38	16.98	30.26	41.02	44.81	47.14	58.89	84.45
Ochiai2	8.38	16.23	31.13	37.85	43.17	47.51	54.96	77.24
CBI Log	8.23	15.38	27.82	35.62	39.20	42.87	55.40	82.91
Jaccard etc	8.00	15.67	28.48	36.33	41.61	45.56	58.45	83.81
Cohen	8.00	15.67	28.48	36.33	40.18	44.12	57.43	83.05
Scott etc	7.99	13.28	16.80	18.43	22.35	23.80	34.47	63.31
Fleiss	7.99	12.53	17.56	19.19	20.83	22.29	34.09	63.15
Tarantula etc	7.85	15.37	27.80	36.33	40.18	44.12	56.47	83.05
CBI Sqrt	7.85	15.38	27.82	35.62	39.20	42.87	55.57	82.81
Rogers etc	5.63	7.73	8.64	10.10	11.57	13.19	27.07	63.15
Russell etc	1.62	4.15	8.86	13.62	18.67	22.90	41.52	81.90
Binary	1.62	4.15	8.86	13.62	18.67	22.90	41.52	81.90
Overlap	1.61	4.13	8.84	13.58	18.62	22.83	41.37	81.90
LingXiao	28.78	34.09	45.45	50.76	53.79	56.06	61.36	N/A
Sober	8.46	19.23	30.76	33.85	40.00	52.31	73.85	82.00
CBI	7.69	18.46	23.07	30.00	33.07	40.00	63.85	75.00
CT	4.65	N/A	N/A	N/A	N/A	27.00	38.00	60.00

gives figures for Cause Transition (CT) from [Cleve and Zeller 2005]. The previous three lines of the table are figures published in [Jiang and Su 2005] for their own system, SOBER [Liu et al. 2005], and CBI [Liblit et al. 2005]. Precise comparison is not possible because the way percentiles are computed is different: [Jiang and Su 2005] uses nodes in the program dependence graph (basic blocks), rather than lines of code that are executed. However, we can conclude that the system of [Jiang and Su 2005] performs better than the best spectral-based diagnosis for the very small percentiles: it ranks significantly more bugs in the top 1%. This is unsurprising as the analysis is much more sophisticated. However, from the eighth percentile upwards, the best spectral methods appear to perform best.

Surprisingly, the predicate-based CBI system seems to perform worse than spectral-based diagnosis using the Tarantula metric. As we have shown, the Tarantula metric is equivalent to our simplified version of the CBI metric. It may be that the actual CBI metric performs even more poorly than our simplified version, or the CBI system makes a poor choice of predicates to test, or the way the figures are calculated makes this comparison misleading (for example, basic blocks versus statements or the treatment of ties in the ranking).

9. RELATED WORK

In this section, we briefly review related work on software diagnosis. Program spectra were first introduced in the context of fixing year 2000 problems in software [Reps et al. 1997]. Various classes of program spectra were introduced: branch hit spectra, complete path spectra and execution trace spectra. In our study we concentrate on statement-level execution traces. Statement and basic block execution spectra have received most attention to date.

Tarantula [Jones et al. 2002; Jones and Harrold 2005] is normally credited as the first spectra-based debugger. It ranks statements in C programs according to what we refer to as the Tarantula metric. One of the novel features of Tarantula is the way the ranking is displayed. It is a graphical debugger which displays the program and uses colour to encode the ranking. A spectrum of colours from red through yellow to green is used; red corresponding the most highly ranked. Brightness is also used to indicate confidence in the ranking, using another metric (this has received much less attention as it is not designed to identify the statements that are most likely to be buggy). The Tarantula metric does not appear to work particularly well, but the Tarantula system could easily be modified to use another metric, as long as the values are scaled appropriately. For example, the rank percentage of each statement could be computed then mapped to a colour in a straightforward way.

Pinpoint [Chen et al. 2002] is a framework for root cause analysis in the J2EE platform, implemented for internet services such as web-mail. It uses data from web client request traces to diagnose exceptions generated due to failures in system components. The Jaccard metric is used for ranking, but others could be substituted.

Ample (Analyzing Method Patterns to Locate Errors) [Dallmeier et al. 2005] is an Eclipse plug-in for identifying buggy classes in Java software. It uses information on method call sequences from multiple passing runs and a single failing run. Ultimately a "weight" is computed for each class, which indicates the likelihood of it being buggy. In Abreu et al. [2006] the ranking method used in Ample is generalised to multiple failing runs (what we refer to as the Ample metric) and applied to blocks of statements in an imperative language; we use it for single statements.

The Ochiai metric was first used for software diagnosis in a comparison with Tarantula, Jaccard and (the generalised) Ample metrics [Abreu et al. 2006; 2007]. For the Siemens test suite, Ochiai was found to perform best (using rank percentages for basic blocks) and was also quite consistent. Jaccard, Tarantula then Ample followed in performance overall, though Ample performed well for some of the programs. Performance for different q_e values was also investigated, by adjusting the sets of test cases used for the different programs. In this paper we have reported similar results using the $ITE2_8$ model, Siemens test suite and Space (ranking statements and using two different performance measures). We have also shown there are several metrics that perform significantly better than Ochiai on single bug programs (and many that perform worse). We have explicitly focused on single bug programs but the results of Abreu et al. [2006] and Abreu et al. [2007] (for example) also have a strong bias towards single bug programs due to the choice of test suite. We are not aware of any work that gives a good comparison of spectral ranking metrics for diagnosing programs with multiple bugs.

The Zoltar metric [Gonzalez 2007] was developed as a modification of the Jaccard metric. A new term in the denominator was introduced to help distinguish the correct blocks from buggy ones. Their experiments showed superior performance compared to Ochiai [Abreu et al. 2006]. Our work confirms this result and shows Zoltar performs almost optimally for our single bug model program $ITE2_8$, and very close to our optimal metrics for other single bug programs.

Most recently, Wong et al. [2007] proposed and evaluated several metrics. We have evaluated what we refer to as Wong1–3. However, in Wong et al. [2007] there are actually three different versions of the Wong3 metric, with different coefficients for a_{ep} in the case where a_{ep} is large. They found the smallest coefficient (0.001) performed best, which is what we have used in our experiments. Our analysis of $ITE2_8$ has showed that even smaller coefficients are warranted when there are 1000 or more tests, and the other two cases for smaller a_{ep} ranges are best dropped. We have also introduced another variation, Wong3$'$, which can perform significantly better in cases where the spectra gathered include statements (or blocks) that are not executed in any test cases.

Naish [2008] uses probabilistic methods to improve the search strategy for declarative diagnosis of logic programming languages. Part of the algorithm used assigns a probability of being buggy to each clause of the program, based on (amongst other things) the total number of times each clause is used in the passed tests and a single failed test. This could be used for ranking. It is unclear how this is best adapted to the spectral approach we consider here, where there can be several failed tests and we don't distinguish single and multiple uses of a particular piece of code in a single test (the matrix has only binary numbers).

The CBI (Collaborative Bug Isolation) system [Liblit et al. 2005] is a predicate-based system, as discussed in section 3. One contribution of this system is a sampling method which can be used to gather data. The aim is to reduce runtime overheads to such an extent that they are acceptable in programs that are deployed. Although less data is produced for each run of the program, overall there can be many runs from many users across the world, resulting in more data overall. We are interested in trying different metrics for CBI. However, sampling in failed tests does not fit well the optimal metrics we have proposed since a_{nf} (or the equivalent data expressed using predicates) may be non-zero for a buggy statement that is executed in all failed tests but not sampled.

The SOBER system [Liu et al. 2005] also ranks predicates but uses a non-binary matrix and uses statistical methods, so a direct comparison with the spectral approach we have investigated is difficult. The number of times a predicate is true in each passed test forms a distribution; similarly for each failed test. Ranking is based on the *evaluation bias* which is a measure of difference between the two distributions for each predicate. For example, if the difference between the means of the two distributions is large and the standard deviations are small the evaluation bias is large and the predicate will be ranked highly.

The *Delta debugging* paradigm [Zeller 2001; Cleve and Zeller 2005; Zeller 2006] is very effective at diagnosing certain classes of bugs. The program is executed multiple times with different subsets of the bug-inducing input in order to find two very similar inputs, only one of which exhibits the bug symptom (this can take some time). A method is needed to determine if the bug symptom appears in a run. If the symptom is a program crash this is simple and in the Siemens test suite there is a correct version of each program so the correct output also can be ascertained. In general, human intervention is required. *Failure cause-effect chains* can then be used to provide a relatively concise explanation of the bug symptom. These are related to the program state (the values of the variables) at various program points. Methods that can simplify the test data that causes a bug symptom are likely to be useful for any diagnosis system.

LingXiao [Jiang and Su 2005] further explored cause-effect chains from isolated bug-related predicates using CBI [Liblit et al. 2005]. Their work is quite similar to delta debugging [Zeller 2001]; they try to locate not only failure but also cause-transition moments. The key difference of their work with the latter is they focus on instrumenting predicates of program using CBI. CBI [Liblit et al. 2005] is used to instrument a program and a list of suspicious predicates is generated. Next, machine learning approaches such as feature clustering and classification algorithms (support vector machine and random forest) are used to assign scores to these predicates. The control flow graph of the program is also used. Paths that connect different suspicious predicates are considered to adjust the ranking.

Although execution paths are an important consideration for the way we use our $ITE2_8$ model, they are not used in the spectral ranking methods we have examined. The a_{ij} values used in ranking metrics contain no path information. The matrix has some additional information. For example, for any set of test executions the matrix contains the information that statement $S1$ in $ITE2_8$ is executed in a test if and only if $S2$ is not executed (and for a large number of tests we might conclude this is not simply a coincidence). The results of Jiang and Su [2005] (reproduced in Table XV) suggest that using the control flow graph is an effective way of improving the higher part of the ranking. The best performing spectral ranking methods seem to find more bugs when a larger percentage of the code is examined. This may be because the machine learning algorithms used in Jiang and Su [2005] do not take advantage of the fact that nearly all Siemens test suite programs have a single bug.

10. FURTHER WORK

There are several ways that our work could be extended for single bug programs. First, metrics such as O could be proved optimal for a larger class of programs (not just $ITE2_8$). Second, optimality for different score functions could be investigated. Third, optimal ranking using the whole matrix and result vector rather than just the four a_{ij} values could be considered. We believe O is optimal within this broader class of functions for $ITE2_8$, but in general the matrix and result vector contains more information than just the a_{ij} values and this could potentially allow functions that perform better than O for some single bug programs. Fourth, predicate-based ranking could be investigated using our model-based approach. The matrix and result vector can capture all relevant information for predicates. Both theoretical and empirical avenues could be explored, such as determining optimal ranking methods and implementing them in a system such as CBI to evaluate them.

Clearly, extending the current work to multiple bug cases is of great interest.

The most obvious direction is to consider (classes of) models with more than one bug and find metrics that are optimal for these classes (possibly using different scoring function or using the whole matrix and result vector). Another possibility is to discover some way of finding a subset (or subsets) of the tests for which the failures are likely to be caused by a single bug. The single bug case gives us extra leverage, but this must be traded off against having a smaller number of tests and the possibility that there is not just a single bug. Though motivated differently, the work of Jones et al. Jones et al. [2007] on "parallel debugging" is promising. They use the Tarantula metric for ranking and present two techniques to cluster failed test cases. One uses similarity of the rankings produced, and an almost 30% reduction in total debugging effort (ignoring parallelism) was observed in experiments with versions of Space with eight bugs. Such experiments could be repeated using O^p rather than Tarantula. We would expect to see improved results. Using O^p, more aggressive clustering (producing smaller sets of tests on average) is likely to pay off, since O^p is more specialised to the single bug case, and this can help minimise the time taken to find all sources of failure when debugging in parallel.

Finally, it may be possible to apply our methodology to other domains, such as medicine. Each test run of the model program could correspond to a different person, the execution of a each statement corresponds to information about the person, such as having a particular gene, and the result vector indicates the people that have some disease. The choice of model program encodes *domain knowledge*. For example, more commonly occurring genes would be in more execution paths. Having constructed an appropriate model, different ranking metrics could be evaluated as we have done, and perhaps optimal metrics could be devised for that model. The best metrics could then used on actual medical data.

11. CONCLUSION

Software diagnosis using program spectra is an instance of the classification or clustering problem. The literature contains *many* metrics that can be used for ranking statements according to how likely they are to be buggy, and there are infinitely many more that could be used. Faced with such a choice it is unclear how metrics are best compared, which proposed metrics are best for a particular situation and whether there are other metrics, not yet proposed, that are substantially better. Our novel solution is to use a *model* which captures important features of the problem, and a method of evaluating the performance of ranking metrics for such a model. This allows analytical approaches to be used. Optimality of a metric can be defined, and classes of metrics can be discovered and proved optimal. Metrics can be compared by seeing how they deviate from optimality.

We have restricted attention to single bug programs in this paper, allowing us to experiment with the methodology in a relatively simple domain. The model for software diagnosis and evaluation measure we have used in this paper are extremely simple. There are just four statements and only the top-ranked statements are considered for performance. We have defined optimal ranking metrics that are much simpler than many of the metrics proposed. Furthermore they seem quite robust, performing better than all other metrics considered in a wide range of conditions, and allow a better understanding of the relative performance of other metrics with the model. There are two other metrics that come close to optimality. Both were proposed for the software diagnosis domain. We have also evaluated the performance of all metrics using the Siemens test suite and Space benchmarks. A small number of metrics are particularly sensitive to such things as statements that are never executed or always executed, which our model does not cover. However, overall the results from the model fit very well with results for single bug programs in the Siemens test suite and Space, with the optimal metrics performing best. We consider this to be a firm validation of our approach.

REFERENCES

Abreu, R., Zoeteweij, P., and van Gemund, A. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, 39–46.

Abreu, R., Zoeteweij, P., and van Gemund, A. 2007. On the Accuracy of Spectrum-based Fault Localization. *Testing: Academic and Industrial Conference Practice and Research Techniques-Mutation, 2007. TAICPART-Mutation 2007*, 89–98.

Anderberg, M. 1973. Cluster analysis for applications. *New York*.

Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. 2002. Pinpoint: problem determination in large, dynamic Internet services. *Dependable Systems and Networks, 2002. Proceedings. International Conference on*, 595–604.

Cleve, H. and Zeller, A. 2005. Locating causes of program failures. *Proceedings of the 27th international conference on Software engineering*, 342–351.

Cohen, J. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement 20,* 1, 37.

Dallmeier, V., Lindig, C., and Zeller, A. 2005. Lightweight bug localization with AMPLE. *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging*, 99–104.

Dice, L. 1945. Measures of the Amount of Ecologic Association Between Species. *Ecology 26,* 3, 297–302.

Do, H., Elbaum, S., and Rothermel, G. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering 10,* 4, 405–435.

Duarte, J., Santos, J., and Melo, L. 1999. Comparison of similarity coefficients based on RAPD markers in the common bean. *Genetics and Molecular Biology 22*, 427–432.

Dunham, M. 2002. *Data Mining: Introductory and Advanced Topics.* Prentice Hall, PTR Upper Saddle River, NJ, USA.

Everitt, B. 1978. *Graphical techniques for multivariate data.* North-Holland, New York.

Fleiss, J. 1965. Estimating the accuracy of dichotomous judgments. *Psychometrika 30,* 4, 469–479.

Gonzalez, A. 2007. Automatic Error Detection Techniques based on Dynamic Invariants. M.S. thesis, Delft University of Technology, The Netherlands.

Goodman, L. and Kruskal, W. 1954. Measures of association for cross classifications. *Journal of the American Statistical Association 49,* 268, 732–764.

Hamming, R. 1950. Error detecting and error correcting codes. *Bell System Technical Journal 29,* 2, 147–160.

Jaccard, P. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull. Soc. Vaudoise Sci. Nat 37*, 547–579.

Jiang, L. and Su, Z. 2005. Automatic isolation of cause-effect chains with machine learning. Tech. rep., Technical Report CSE-2005-32, University of California, Davis.

Jones, J. and Harrold, M. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 273–282.

JONES, J., HARROLD, M., AND BOWRING, J. 2007. Debugging in Parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis.* ACM Press New York, NY, USA, 16–26.

JONES, J., HARROLD, M., AND STASKO, J. 2002. Visualization of test information to assist fault localization. *Proceedings of the 24th international conference on Software engineering*, 467–477.

KRAUSE, E. 1973. Taxicab Geometry. *Mathematics Teacher 66,* 8, 695–706.

LEE, C. 1958. Some properties of nonbinary error-correcting codes. *IEEE Transactions on Information Theory 4,* 2, 77–82.

LEE, H., NAISH, L., AND RAMAMOHANARAO, K. 2008. Evaluation of metrics based on program spectra for software fault localization. *Submitted for publication.*

LIBLIT, B. 2004. Cooperative Bug Isolation. Ph.D. thesis, University of California.

LIBLIT, B., NAIK, M., ZHENG, A., AIKEN, A., AND JORDAN, M. 2005. Scalable statistical bug isolation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 15–26.

LIU, C., YAN, X., FEI, L., HAN, J., AND MIDKIFF, S. P. 2005. Sober: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes 30,* 5, 286–295.

LOURENÇO, F., LOBO, V., AND BAÇÃO, F. 2004. Binary-based similarity measures for categorical data and their application in Self-Organizing Maps.

MAXWELL, A. AND PILLINER, A. 1968. Deriving coefficients of reliability and agreement for ratings. *Br J Math Stat Psychol 21,* 1, 105–16.

MEYER, A., GARCIA, A., SOUZA, A., AND SOUZA JR, C. 2004. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L). *Genetics and Molecular Biology 27*, 83–91.

NAISH, L. 2008. Probabilistic declarative debugging. *Journal of Functional and Logic Programming 2008,* 1 (July).

OCHIAI, A. 1957. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bull. Jpn. Soc. Sci. Fish 22*, 526–530.

PYTLIK, B., RENIERIS, M., KRISHNAMURTHI, S., AND REISS, S. 2003. Automated Fault Localization Using Potential Invariants. *Arxiv preprint cs.SE/0310040*.

RENIERES, M. AND REISS, S. 2003. Fault localization with nearest neighbor queries. *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, 30–39.

REPS, T., BALL, T., DAS, M., AND LARUS, J. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, 432–449.

ROGERS, D. AND TANIMOTO, T. 1960. A Computer Program for Classifying Plants. *Science 132,* 3434, 1115–1118.

ROGOT, E. AND GOLDBERG, I. 1966. A proposed index for measuring agreement in test-retest studies. *J Chronic Dis 19,* 9, 991–1006.

RUSSEL, P. AND RAO, T. 1940. On habitat and association of species of anopheline larvae in south-eastern Madras. *J. Malar. Inst. India 3*, 153–178.

SCOTT, W. 1955. Reliability of Content Analysis: The Case of Nominal Scale Coding. *Public Opinion Quarterly 19,* 3, 321–325.

Telcordia Technologies, Inc. 1998. Telcordia software visualization and analysis toolsuite (χSuds). Users manual, Chapter 12.

WONG, W., QI, Y., ZHAO, L., AND CAI, K. 2007. Effective Fault Localization using Code Coverage. *Proceedings of the 31st Annual International Computer Software and Applications Conference-Vol. 1-(COMPSAC 2007)-Volume 01*, 449–456.

ZELLER, A. 2001. Automated debugging: are we close? *Computer 34,* 11, 26–31.

ZELLER, A. 2006. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann.

Received November 2008; revised February 2009; accepted XXX