Statements versus predicates in spectral bug localization

Lee Naish, Hua Jie Lee, Kotagiri Ramamohanarao
Computer Science and Software Engineering
University of Melbourne
Melbourne, Australia
Email: {lee, leehj, rao}@csse.unimelb.edu.au

***Abstract*—This paper[1] investigates the relationship between the use of predicate-based and statement-based program spectra for bug localization. Branch and path spectra are also considered. Although statement and predicate spectra can be based on the same raw data, the way the data is aggregated results in different information being lost. We propose a simple and cheap modification to the statement-based approach which retains strictly more information. This allows us to compare statement and predicate "metrics" (functions used to rank the statements, predicates or paths). We show that improved bug localization performance is possible using single-bug models and benchmarks.**

***Keywords*-bug localization, program spectra, statements, branches, predicates, paths**

I. Introduction

Bugs are be pervasive in software under development and tracking them down contributes greatly to the cost of software development. One of many useful sources of data to help diagnosis is the dynamic behavior 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 (known as program spectra [1]), such as the statements that are executed or predicates that are true, for each test case. If a certain statement is executed in most failed tests but few passed tests we may conclude it is likely to be buggy. Similarly, if some predicate such as $x < 0$ is true at a particular program point in most failed tests but few passed tests it may also help a programmer locate a bug. Generally the raw data is aggregated in some way to get data for each statement or predicate and some function (we use the term *metric*) is used to rank the statements or predicates according to how likely they are to be buggy (or associated with bugs). A programmer can then use this information to help find a bug. This paper primarily investigates the relationship between these statement-based and predicate-based approaches to bug localization. We make the following contributions:

- We show that although the raw data collected with both approaches is the same, the way that it is aggregated loses information. Different information is lost in the different approaches, with the statement-based approach losing more information in some sense.
- We present a simple cheap heuristic way of extracting some predicate spectra from statement spectra. This is practically important because there are many profiling tools available which provide statement spectra, whereas obtaining predicate information is often more difficult. Although our method is not completely accurate, it provides enough additional information to be useful and we discuss ways accuracy can be improved further.
- We show that retaining predicate spectra can theoretically improve performance of bug localization. We use a previously proposed model-based methodology where a particular statement-based metric O^p was proven optimal for locating bugs in a simple class of single-bug programs. Using a similar model we propose new predicate-based metrics which perform better than O^p and much better than previously proposed predicate-based metrics.
- We also report on practical experiments where the new metrics perform very well. Our experiments allow a fair comparison between several different metrics and show that previously proposed predicate-based metrics perform relatively poorly.

The rest of the paper is organized as follows. Section II provides the necessary background on spectra-based diagnosis. Section III shows how the raw data collected is the same with the two approaches but differs when the data is aggregated. Section IV shows how predicate spectra can be cheaply approximated using statement spectra. Sections V and VI give performance figures for a theoretical model and commonly used benchmark sets. Sections VII and VIII discuss results from the CBI and Holmes systems, including the use of branch and path spectra, and other related work. Section IX discussed further work and Section X concludes.

II. Background

All spectral methods use a set of tests, each classified as failed or passed. For statement spectra [2], [3], [4], [5], which we describe first, we gather data on whether

Table I
STATEMENT SPECTRA WITH TESTS $T_1 \ldots T_5$

	T_1	T_2	T_3	T_4	T_5	f	p	f_n	p_n
Statement_1	1	0	1	1	1	1	3	1	0
Statement_2	1	1	0	1	0	2	1	0	2
Statement_3	0	1	0	1	1	1	2	1	1
				⋮					
Test Result	1	1	0	0	0				

$F = 2,\ P = 3$

each statement is executed or not for each test. This can be represented as a binary matrix with a row for each statement and a column for each test; 1 means executed and 0 means not executed. For each statement, four numbers are ultimately produced. They are the numbers of failed and passed test cases in which the statement was executed (f and p) and failed and passed test cases in which the statement was not executed (f_n and p_n). Table I gives an example with five tests, the first two of which fail. Statement metrics (numeric functions used to rank the statements) are a measure of how similar the rows of the matrix are to the result vector, or how similar the set of failed tests is to the set of tests which execute the statement. Measures of similarity are important in classification and machine learning, not just debugging, and scores of different metrics have been proposed (see [5]). Most commonly they are defined in terms of these four values. However, due to our use of predicate spectra as well, it is simplest for us to define them using p, f and the total number of failed and passed tests, F and P, respectively (this is possible because $f_n = F - f$ and $p_n = P - p$).

Table II gives definitions of the statement metrics used here. Due to space limitations we focus on metrics which have been previously used for software diagnosis and introduce them very briefly ([5] provides more background). The Jaccard metric is the oldest and was originally used for classifying plants [6]; it has been used in the Pinpoint debugger [7]. The Russell (and Rao) metric [8] is also quite old; we include it because it is related to some metrics for predicate spectra. The Tarantula system [2] was the first spectra-based debugger. Ochiai, Jaccard, Ample and Tarantula metrics have been evaluated for diagnosis using the Siemens test suite [9]; the Ample metric was adapted from the AMPLE bug localization system [10]. The Zoltar metric [11] has been proposed for debugging as an enhancement to Jaccard, and Wong3 is the best of several metrics proposed for debugging in [4]. O^p is one of a class of metrics which are proved to be theoretically optimal for some single-bug programs [5] (we discuss this work further in Section V).

For predicate spectra [12], [13] we collect data on predicates, such as the conditions of if-then-else statements. As well as collecting data related to control flow (branch spectra), other predicates can also be instrumented. For example, whether the return value of a function is negative, zero or positive is often important, and predicates can be introduced to test this. Similarly, when a variable is assigned a value, predicates can be introduced to compare it to other related variables and constants of the same type. For each test it is determined if execution ever reaches that program point (the predicate is reached, or "observed") and whether the predicate is ever true. As with the statement-based approach, for each predicate we ultimately compute four integers: the numbers of failed and passed tests in which the predicate was observed (f' and p') and the numbers of failed and passed tests tests in which the predicate was true (f and p; we relate these to the f and p for statements in Section III). Predicate metrics are applied to these values in the same way as statement metrics in order to rank predicate according to how closely they are associated with a bug.

Table III gives the definitions of the predicate metrics we use. The first two rows are based on the CBI (Collaborative Bug Isolation) system [12]. The simplest metric used in CBI is Increase, defined in terms of Failure and Context. Context is the proportion of tests which fail, of all the tests which observe the predicate. Failure is the proportion of tests which fail, of the tests where the predicate is true. The difference betweeen these values (Increase) may ituitively indicate how closely the predicate being true is associated with a bug. CBI has also used two other metrics, Log and Sqrt, which are the harmonic mean of Increase and another value based on the proportion of failed tests for which the predicate is true (FP). One uses logarithms (Log) and the other square roots (Sqrt) to attempt to reduce the influence of FP. Failure is equivalent to Tarantula and FP is equivalent to Russell for ranking purposes [5].

The last row defines our three new metrics. The first, FPC, adds Context to Failure rather than subtracting it. Our experiments suggest this performs significantly better than Increase, indicating the intuition behind Increase has dubious merit. The next two, OFPC and O8FPC, are variations of OPC based on the understanding of optimal metrics for single bug programs. The optimal metrics discussed in [5] rank primarily on f and secondarily on $-p$ (or $1/p$). If we ignore f' and p' (or they have the same values for all predicates, so Context is constant) OFPC and O8FPC have the same property, and are therefore optimal in the same sense as O^p. However, they both give some weight to Context: the same weight as Failure in OFPC and one eight the weight of Failure in O8FPC. We currently have little theoretical understanding of the multiple bug case, so we cannot say whether the predicate-based approach is theoretically better than the statement-based approach for multiple bugs. Hence we make single bugs our intial focus in comparison and evaluation.

Table II
DEFINITIONS OF STATEMENT METRICS USED

Name	Formula	Name	Formula	Name	Formula
Jaccard	$\frac{f}{F+p}$	Tarantula	$\frac{\frac{f}{F}}{\frac{f}{F}+\frac{p}{P}}$	Ample	$\left\lvert \frac{f}{F} - \frac{p}{P} \right\rvert$
Ochiai	$\frac{f}{\sqrt{F*(f+p)}}$	Zoltar	$\frac{f}{F+p+\frac{10000F-f*p}{f}}$	Russell	$\frac{f}{F+P}$
Wong3	$f-h$, where $h = \begin{cases} p & \text{if } p \leq 2 \\ 2+0.1(p-2) & \text{if } 2 < p \leq 10 \\ 2.8+0.001(p-10) & \text{if } p > 10 \end{cases}$			O^p	$f - \frac{p}{P+1}$

Table III
DEFINITIONS OF PREDICATE METRICS USED

Name	Formula	Name	Formula	Name	Formula
Failure	$\frac{f}{f+p}$	Context	$\frac{f'}{f'+p'}$	Increase	$Failure - Context$
FP	$\frac{f}{F}$	Log	$\frac{2}{\frac{1}{Increase}+\frac{\log F}{\log f}}$	Sqrt	$\frac{2}{\frac{1}{Increase}+\frac{\sqrt{F}}{\sqrt{f}}}$
FPC	$Failure + Context$	OFPC	$3f + FPC$	O8FPC	$10f + 8Failure + Context$

III. RAW DATA AND AGGREGATED DATA

There is no essential difference in the raw data (the binary matrix) collected from predicates and statements. Consider the following three lines of code:

```
1:    if (C)
2:            T;
3:    else    E;
```

Instrumenting the three statements gathers identical information to instrumenting predicate C and its negation: both C and its negation are observed if and only if line 1 is executed, C is true if and only if line 2 is executed and the negation of C is true if and only if line 3 is executed. Note that in general it is possible that T and E are both executed (C and its negation are both true at some times) because the statement may be executed more than once in a test.

Given a predicate which is deemed worth instrumenting but not used in control flow, it is possible to add a dummy if-then-else which tests the predicate and has no-ops in the "then" and "else" branches. If for each predicate instrumented we have an if-then-else, the raw statement information contains all the predicate information (assuming we also know the structure of the program — which rows of the matrix correspond to which "then" and "else" branches *etc*). Similarly, if all conditions in a program which determine control flow are instrumented, then the raw predicate information contains all the statement information.

However, although the raw data is the same in both cases, the way it is aggregated to get four integers for each statement is different with the two approaches. For predicates, f and p are the same as the f and p for the corresponding "then" or "else" statement, but f' and p' are the same as the f and p for the corresponding "if" statement. Thus information from two *different* statements is combined in predicate spectra. For a given set of tests, F and P are fixed — they don't vary between statements. Thus, for a statement metric the only difference between different statements is the f and p values. There can be $F+1$ distinct f values (since $0 \leq f \leq F$) and $P+1$ distinct P values (since $0 \leq p \leq P$) so the number of distinct possible argument values for a statement metric is

$$\sum_{p=0}^{P}\sum_{f=0}^{F} 1 = (P+1)(F+1)$$

For predicates, $0 \leq p' \leq P$ and $0 \leq f' \leq F$ but, in addition, f and p can vary between statements: $0 \leq p \leq p'$ and $0 \leq f \leq f'$. The number of distinct values is

$$\sum_{p'=0}^{P}\sum_{f'=0}^{F}\sum_{p=0}^{p'}\sum_{f=0}^{f'} 1$$

Thus, compared to statement metrics, there are strictly more possible distinct values for the arguments to predicate metrics. For example, with $F = 5$ and $P = 15$ there are 96 possibilities for statement metrics and 2856 for predicate metrics.

The predicate approach to aggregation thus maintains more information, by combining information from different (though not completely independent) statements in applications of the metric. The statement approach allows us to know F and P, which cannot generally be determined with the predicate approach, so the aggregated predicate data does not contain all the information of the aggregated statement data. However, our results indicate that F and P are less important than the information obtained from pairs

of statements. Also, it is trivial to compute F and P in a predicate-based system and allow the metrics to use this information (in fact, Log and Sqrt in CBI use F).

It is possible to have a unified view of statement and predicate spectra by having the two "global" variables F and P (which are the same for all program points) as well as the four predicate spectra variables, f, p, f' and p'. Whether the latter are associated with predicates or statements is immaterial to the inner workings of a bug localization tool. Giving a certain rank to a predicate which is the condition of an if statement is equivalent to giving that rank to the statements in the "then" branch. Partly due to infrastructure we have previously developed, we associate the spectra with statements. Thus f' and p' for a statement in a "then" or "else" branch are the f and p, respectively, for the associated "if". For statements at the top level of a function, where there is no enclosing conditional, we say $f' = f$ and $p' = p$. This is equivalent to there being an enclosing conditional which always chooses this branch, and leads to reasonable default behavior. In the experiments we report on later we give performance for both statement and predicate metrics; the only difference is that the statement metrics do not use f' and p' in their definitions.

IV. Predicate spectra from statement spectra

Many profiling tools can be used to extract statement spectra easily (for example, we have used `gcov`, part of the `gcc` compiler suite). It is somewhat harder to find tools which produce predicate spectra. Even when available, it is often harder to link this data to source code, partly because we often put more than one predicate on a single line (when there is a condition which involves conjunction or disjunction), whereas we rarely put more than one statement on a single line. It can therefore be useful to combine statement spectra from different statements in order to reconstruct predicate spectra. To do this accurately requires knowledge of the program structure, otherwise it is not clear which pairs of statements should be combined. Writing and maintaining a tool which analyses program structure and combines it with statement spectra is a significant task. Combining *all* pairs of statements is simple but does not scale well (with N statements there are $O(N^2)$ pairs) and most of the pairs will be unrelated statements and of doubtful use.

The approach we suggest here is to just use pairs of *consecutive* statements, which can be done very easily, and use heuristics to infer the program structure and hence compute predicate information. Consider the code segment below:

```
1: if (C1) {
2:          S2;
3:          S3;
4: } else { S4;
5:          S5; }
6: S6;
```

Comparing the statement spectra for statement S_1 (line 1) and S_2 gives us the predicate information for C_1. Also, assuming statements S_2–S_5 do not make the control jump to outside this if statement (using `goto`, `break`, `return`, *etc*) the spectra for S_6 will be identical to S_1. Comparing the statement spectra for S_6 and S_5 will thus (usually) give us the predicate information for the negation of C_1. In general, if we can recognise the first statement in a "then" branch and the statement immediately following the end of an "else" branch we can reconstruct predicate spectra. We know that both the "then" and "else" branches will be executed no more than the if statement. The interesting case (where the predicate information is potentially useful) is when the "then" or "else" branches are executed strictly less than the if statement.

We use the following heuristic for recognising "then" and "else" branches and computing f'_i and p'_i (we use subscripts on the spectra variables here to denote statement numbers). If $f_i \leq f_{i-1}$, $p_i \leq p_{i-1}$ and either $f_i < f_{i-1}$ or $p_i < p_{i-1}$ then we assume statement i is the start of a then block; we set f'_i to f_{i-1} and p'_i to p_{i-1}. Otherwise, if $f_i \leq f_{i+1}$, $p_i \leq p_{i+1}$ and either $f_i < f_{i+1}$ or $p_i < p_{i+1}$ then we assume statement i is the end of an else block; we set f'_i to f_{i+1} and p'_i to p_{i+1}. Otherwise, we set f'_i to f_i and p'_i to p_i. Recognising "then" and "else" branches can be done more accurately if we have access to the whole matrix of raw data: we accurately check if the tests which executed one statement are a subset of the tests which execute the previous/next statement (the logic above approximates this). This has worse space complexity, but compromises are possible, where we store some extra information about each row of the matrix rather than the whole row.

Having determined which statements are at the start of a "then" block and the end of an "else" block, the spectra values for these statements should be propagated forwards (for "then") or backwards (for "else") to all statements in that block. Again, we use a heuristic to determine such statements. We assume any statement with the same f and p value is in the block, but stop scanning if we encounter a statement with strictly greater f or p value. If we encounter smaller f or p values we just keep scanning; this is so we can handle nested if-then-else statements between S_2 and S_3 or between S_4 and S_5, for example. Again, having access to the matrix of raw data (or a summary) can lead to greater precision. For example, if we have a hash of each row stored, we can scan for equal hash values rather equal f and p values, increasing precision considerably. The time complexity could also be improved by creating doubly linked lists of statements with the same hash (or f and p) values. The figures we provide here are for the simple algorithm without these improvements.

Algorithm 1 sketches the overall process. We apply this

algorithm for each statement, in order. If the statement has already had its f' (and p') computed we do nothing, to prevent values which are computed when scanning forwards from being overwritten.

```
// Compute f' and p' for the i^th statement, S_i,
// (and maybe others) given f and p values
PredSpectra(i):
begin
    if f'[i] not yet assigned then
        if S_i used in less tests than S_{i-1} then
            // Start of then block (probably)
            f'[i] = f[i-1];
            p'[i] = p[i-1];
            Scan(i, 1); // scan forwards
        end
        else if S_i used in less tests than S_{i+1} then
            // End of else block (probably)
            f'[i] = f[i+1];
            p'[i] = p[i+1];
            Scan(i, -1); // scan backwards
        end
        else
            f'[i] = f[i];
            p'[i] = p[i];
        end
    end
end
Scan(i, inc):
begin
    j = i + inc;
    while S_j exists and is not used in more tests than
    S_i do
        if S_j used in same tests as S_i then
            f'[j] = f[i];
            p'[j] = p[i];
        end
        j = j + inc;
    end
end
```

Algorithm 1: Computing f' and p' from f and p values

There are several ways in which Algorithm 1 can infer the program structure incorrectly, but in most cases the spectra produced are still correct. For example, consider the case where the "then" branch is used in all tests in which the if statement is executed but the *else* branch is not. Statements S_1 and S_2 are not recognised as a then branch but the default f' and p' values are what would be computed anyway. S_4 and S_5 are inferred to be a "then" branch and hence their f' and p' are the f and p values of S_3, respectively (which are the same f and p values as S_1, the if statement, as desired). We get a similarly correct result in cases where the "else" branch is always executed, or there is no else branch. While and for loops are indistinguishable from if statements since we only collect binary data, not execution counts. The main inaccuracy of Algorithm 1 is in the scanning process, where sometimes scanning proceeds to far and assigns f' and p' values it should not.

V. Theoretical performance

To assess performance in idealised conditions we adopt the model-based approach of [5]. We briefly review it here. The following very simple program (named ITE2) was used to model the debugging problem. Statement S_4 is buggy; the others are correct (the choice of S_4 is arbitrary due to symmetry but the fact there is a single bug is important).

```
if (C1) S1; else S2;
if (C2) S3; else S4;
```

S_4 is executed in half the tests on average and, of those, half fail on average (buggy code can behave correctly). The second if-then-else can be seen as the source of a "signal" in the data, indicating where the bug is. The fact that S_4 sometimes behaves correctly attenuates the signal. The first if-then-else is a source of "noise". Depending on the set of tests, executions of S_1 or S_2 may be correlated with the tests failing and thus be ranked equal or higher than S_4. A set of tests corresponds to a multiset of execution paths through the program, where faulty and correct executions of S_4 are considered distinct paths. Given a multiset of execution paths, program spectra for the statements S_1–S_4 can be generated, the statements ranked according to various metrics and the performance of the metrics evaluated for that particular set of tests (the best case is when S_4 ranked strictly above the other statements).

Given a number of tests, N, all possible multisets of N execution paths can be considered in order to evaluate the *overall* performance of each metric. For large N (or large numbers of paths if we use a program more complicated than ITE2) there are too many multisets to practically enumerate, but random sampling from a uniform distribution can be used to find approximate overall performance. We can also determine how overall performance is affected by various factors such as what proportion of tests fail, what proportion of tests execute the bug and how "consistent" the bug is (what proportion of the tests which execute the bug fail).

Although the ITE2 model program is very simple, it accurately predicts relative performance of metrics on "real" C programs seeded with single bugs. The O^p metric was proven to be in a class of metrics which are *optimal* for this model, with respect to a simple evaluation measure, for all numbers of tests (no metric can perform better overall). O^p also performed best on real benchmarks. Although other enhancements to statement-based spectral debugging have been made [14], a method which out-performs O^p in the single bug case has been elusive.

In the ITE2 model program, both if statements are always executed, so $Context$ is constant for a given set of tests. In such a case it can be shown that the Increase metric is equivalent to the Tarantula metric for ranking purposes (in general, Tarantula is equivalent to Failure) [5]: using the Tarantula metric with statement spectra for S_1 – S_4 produces identical rankings to using Increase (or Failure) with predicate spectra for C_1, C_2 and their negations. In order to compare the use of predicates and statements for spectral debugging in a more general setting (where Context can vary between different statements for a given set of tests) we used a model with two (symmetric) *nested* if-then-else statements, as follows:

```
if (C1) {
    if (C2) S3; else S4;
    S5;
} else
    S6;
if (C7) {
    if (C8) S9; else S10;
    S11;
} else
    S12;
S13;
```

Note that statements S_5, S_{11}, S_{13} and the if statement with condition C_7 will have the same statement spectra as a previous if statement; they enable us to compute predicate information from statement information of consecutive statements. In our experiments we picked one of the inner statements, S_3, to be buggy (note that it may not have the same Context value as S_9). We ranked all thirteen statements, including the if statements. Although O^p has not been proven to be optimal for such a model, it is reasonable to expect it is optimal and in our experiments no statement-based metric has performed better for this model.

The performance measure we use is *rank percentages*, used by various researchers [3], [4], [5]. This is the rank of the highest ranked buggy statement, expressed as a percentage of the program size. For example, if the program has 100 lines of code and the highest ranked bug is the tenth in the ranking, the rank percentage is 10% and the programmer needs to examine 10% of the program code in order to locate the bug (assuming they follow the ranking). Lower rank percentages thus mean better performance. If there are ties in the ranking (several statements, including the bug, have the same metric value) we take the mid-point. We use the average over all sets of tests. Table IV gives "ideal" results (where the structure of the code is known precisely) in average rank percentages for five, twenty, fifty and two hundred tests. We included "SLog" and "SSqrt" which are the same as SLog and SSqrt, respectively, except they use a constant value for *Context*. Recall that Tarantula is equivalent to Failure.

Number of tests	5	20	50	200
Predicate metrics				
O8FPC	11.21	8.14	7.74	7.69
OFPC	11.21	8.18	7.75	7.69
FPC	11.96	8.54	7.88	7.71
Log	21.64	10.78	9.04	8.22
Increase	22.16	11.29	9.16	8.22
Sqrt	29.95	11.34	9.10	8.24
Context	21.82	18.03	16.51	15.64
Statement metrics				
O^p	12.71	8.18	7.75	7.69
Zoltar	12.74	8.20	7.75	7.69
Ochiai	12.74	8.23	7.76	7.69
Jaccard	12.74	8.27	7.78	7.70
SLog	12.78	8.31	7.83	7.72
Ample	16.18	8.33	7.75	7.69
Wong3	15.31	8.44	7.75	7.69
SSqrt	13.19	8.47	7.85	7.76
Tarantula	13.19	8.57	7.88	7.71
Russell	33.52	30.51	28.06	26.96

Table IV
IDEAL RANK PERCENTAGES FOR METRICS WITH MODEL

Num. tests	5		20		50	
Heuristic	Then	Scan	Then	Scan	Then	Scan
O8FPC	11.21	12.21	8.14	8.14	7.74	7.74
OFPC	11.21	12.21	8.18	8.11	7.75	7.74
FPC	11.84	13.00	8.53	8.45	7.87	7.86
Log	21.37	13.32	10.82	12.43	9.06	10.03
Increase	22.06	14.22	11.38	13.00	9.18	10.16
Sqrt	30.47	14.02	11.42	12.86	9.13	10.10
Context	25.39	33.10	20.76	16.26	18.33	14.09

Table V
RANK PERCENTAGES USING HEURISTICS WITH MODEL

Table V gives results for the predicate metrics using heuristics to determine the program structure: "Then" means just the heuristic for detecting "then" and "else" statements and "Scan" means the additional scanning to determine the whole "then" or "else" block. Neither heuristic affects the statement metrics because they don't use f' or p'. Since there are only single statements in "then" and "else" the scanning is not necessary for this model, it just makes inference of the program structure less accurate, if anything.

We have done some experimentation with similar models to help validate these results and gain a deeper understanding, but more is desirable. As expected, the best of the statement metrics is O^p. However, the best of these metrics overall is the predicate metric O8FPC, followed by OFPC. This is the first time a spectral debugging method has been shown to be superior to O^p for single bug programs. Thus the additional information retained can increase performance. It is not yet known how much performance gain is possible, since we don't know what the optimal predicate metric is. The previously proposed predicate metrics (essentially based on Context, Tarantula and Russell) do not perform well and FPC is significantly better than Increase (it even beats O^p for small numbers of tests).

All the best metrics rank statement primarily on f and give a small negative weight to p (the condition for optimality given in [5]). Without the p component we have (the equivalent of) the Russel metric, which performs much worse. The weight given to f' and p' is comparable to that of p or (in the case of O8FPC) significantly smaller. Thus it seems that the most important information is captured by the statement spectra (or, equivalently, whether predicates are true). The information retained exclusively by predicate spectra (whether the predicate has been observed) is significantly less important but still a useful refinement.

For large numbers of tests the performance of the better metrics converges to 7.69 (1/13, which is the best possible value since there are thirteen statements). For Context, S_3 and S_4 are always equally ranked and the performance seems to converge (slowly) to 2/13. As the number of tests grow in our model, the results are more dominated by sets of tests which have a reasonably uniform coverage of different paths. Real tests suites have much less uniform coverage and the convergence is slow at best. The performance of Wong3 is also significantly affected by the number of tests. Of the three different cases in the definition of the metric (see Table II) the last gives best performance (it is very similar to O^p). With smaller numbers of tests the first two cases are used more, particularly with this model, in which the buggy statement is only executed in a quarter of the tests, on average.

The heuristic for determining "then" and "else" blocks works well for reasonable numbers of tests (the model includes extra statements to help the accuracy of this heuristic). Even for very small numbers of tests, the inaccuracy ends up not affecting overall performance at all for the better metrics and the poorer metrics are only affected a little. The heuristic for scanning blocks results in more performance variation, particularly for poorer metrics. As we mentioned, both heuristics can be made more accurate reasonably easily; we have implemented a version of scanning which uses the whole binary matrix and this results in a performance variation of less than 0.06 for all cases in Table V. For the better metrics, even when the simplest version of both heuristics are used, the performance is better than O^p.

VI. Practical performance

We now discuss performance on actual buggy programs. The main benchmark we use is a combination of the Siemens Test Suite (STS) and Unix (a collection of Unix utilities) [15], widely used benchmarks [5], [3], [2], [16]. They consist of multiple versions of several small C programs seeded with bugs, along with numerous test cases (1000-6000 for STS and 150–900 for Unix). We used the same "single bug" subset of this test suite as in [5]; there are 224 programs in total. In addition, we used the Concordance benchmark [17] consisting of 13 programs with "real" bugs (as opposed to deliberately seeded bugs); we excluded two which have multiple bugs. We used `gcc` version 4.2.1 and `gcov` to extract program spectra. This gives spectra for all lines of each program (including blank lines, *et cetera*); we ignored lines that were never executed when calculating average rank percentages. Table VI gives performance figures for the various metrics with both sets of benchmarks.

Metric	STS+Unix	Concordance
Predicate metrics		
OFPC	17.94	9.95
O8FPC	17.94	10.08
FPC	28.24	19.90
Context	30.18	20.00
Log	44.93	49.05
Increase	45.41	50.04
Sqrt	47.01	51.18
Statement metrics		
O^p	17.86	10.11
Wong3	18.19	10.15
Zoltar	18.23	10.11
Ochiai	21.63	11.19
Jaccard	23.64	17.68
SLog	26.23	19.59
SSqrt	26.77	20.42
Tarantula	27.09	20.03
Ample	30.16	27.53
Russell	30.02	21.03

Table VI
RANK PERCENTAGES WITH STS+UNIX AND CONCORDANCE

Unfortunately, the new predicate metrics were not able to out-perform O^p for STS and Unix. However, they perfomed best for Concordance, and for STS and Unix they performed better than all metrics other than O^p. With more accurate heuristics and possibly better predicate metrics there are good prospects of predicate metrics performing better than O^p overall for single-bug programs. As with the model, the previously proposed predicate metrics performed poorly. For Concordance their performance was indistinguishable from random ranking and for the other benchmark set they barely performed better. This confirms the suspicions raised in [5] that these metrics are poorly designed. In [5] the performance of CBI and various statement metrics was compared but it was unclear if the comparison was fair due to significantly different ways of measuring performance. Our experiments here only use predicates related to branches but it seems unlikely that relative performance of metrics varies greatly with the kinds of predicates chosen. Although the heuristics may affect performance of the predicate metrics to some degree, it seems clear that the previously proposed metrics perform more poorly than the comparable statement metrics with constant *Context* (SLog, SSqrt and Tarantula). FPC performs better than Increase and our two new predicates influenced by O^p perform significantly better still.

VII. Related Work — CBI and Holmes

The main author of the CBI system has subsequently contributed to the Holmes system [18]. Holmes uses *path* profiles (or spectra): sets of statements which form (part of) an execution path. Only acyclic paths through single functions are considered, since such information can be gathered relatively efficiently [19], but this still provides a significantly richer source of information than single statements or branches. The raw data for paths contains strictly more information than the raw data for statements (Table I). A variant of the predicate spectra methodology is used to compute four integers for each path, to be used for ranking. For each passed and failed test, a path can be executed (like a predicate being true, giving p and f) or observed/reached (meaning the first statement in the path is executed but not the whole path, giving p' and f'). The Log metric from CBI is used for ranking. Instead of producing the entire ranking, a smaller set of paths is output to indicate the most likely causes of bugs. The top-ranked path is always included. If it cannot explain all failed tests (because only some failed tests executed this path) the remaining failed tests are used to recompute the ranking of other paths and the top-most one is added to the set of likely bug indicators. This is repeated until all failures are explained.

In [18], there is an evaluation of the relative performance of path spectra, predicate spectra (in the style of CBI) and branch spectra (the subset of predicates used in control flow). Because the whole ranking is not produced, a different way of measuring performance is given. Starting with the bug indicators, a breadth-first search (BFS) of the program dependence graph is performed until the actual bug is found. The number of nodes in the graph examined, expressed as a percentage of the whole graph, is used as the performance indicator (each node corresponds to a basic block in the code). A graph of the number of bugs found in the benchmark set versus the percentage of the code examined is given. As may be expected, path spectra performed best, followed by predicates then branches. However, the absolute performance, particularly for branch spectra, is significantly lower than what we would expect.

When examining 50% of the code, around 9% of bugs were found using branch spectra, compared with 51% for predicates and 78% for paths [18]. The benchmark set was a combination of part of the Siemens test suite plus some larger programs. Even by randomly guessing bug locations we should be able to achieve 50% on average. In [20] it was reported that CBI found 75% of bugs in the Siemens test suite by examining 50% of the basic blocks in the code. In [5], by examining 50% of the executed lines of code using statement spectra (which should be very similar to branch spectra, as discussed here) 83% of bugs were found using the SLog metric and 90% were found using O^p (using $O8FPC$ and $OFPC$ and the heuristics described in this paper gives the same figure and using Log gives 54%)). In [18] it is suggested that the very poor performance for branch spectra may be due to the fact that some of the programs in the test suite has previously undergone extensive testing. This may be true, but here we propose some other possible reasons for the worse than expected performance.

In [18] there is no statement about how performance is measured in the case of ties. If ties are reported using the worst case scenario (where the bug is assumed to be the last node examined at that level of the BFS) then this will under-estimate the expected performance. Furthermore, as performance drops, the under-estimation will tend to worsen. When performance is very good, the bug will be close to the bug indicators in the graph, there will be few such nodes and therefore few nodes in the tie. When performance is worse, the distance will be greater and (generally) there will be far more nodes in the tie so gap between the average case and worst case will be larger. Thus the reported performance may be significantly worse than the actual average performance.

There is another reason related to the breadth-first search which may affect actual performance. The reason for using BFS to evaluate performance is because only a few of the bug indicators are reported, rather the ranked list containing all of them. This effectively discards much of the information collected dynamically. For example, it may be that there are several bug indicators with high metric values which explain all failed tests. The Holmes system would return just one of them to the user. If it does not indicate the actual bug location, the user has only static information to fall back on — the program dependence graph. The performance measure assumes the user will perform a BFS using this information. This *implicitly* gives a ranking to all nodes (the performance measure gives the rank percentage for this implicit ranking). Given that the second and subsequent ranked bug indicators have high metric values, one would presume this information is more valuable than the static information, yet Holmes discards it. Returning the whole ranking, or at least the ranking from bug indicators with high metrics values, may well lead to better performance.

VIII. Other Related Work

We now briefly review other related spectral debugging work. The SOBER system [13] is another important predicate-based spectral debugging system. It uses a non-binary matrix containing frequency counts (for example, how often a predicate evaluated to true in a test) rather than the "binarised" data we use here. The number of times a predicate is true in each passed test forms a weighting; similarly for each failed test. Ranking is based on the *evaluation bias* which is an established statistical measure of difference between the two weightings for each predicate. It is difficult to do a direct comparison with approaches which uses metrics over the (aggregated) binarised data.

The work which resulted in the Wong3 metric has been extended, and a considerably more elaborate metric has been proposed [21]. We have not evaluated it here, partly because it is so complicated and partly because some of our other experiments suggest it does not perform significantly better than Wong3 for single-bug programs [22]. In [14] a variation of the normal ranking method is proposed, for statement spectra. Each failed test is given a weight, dependent on the number of statements executed in the test, with smaller numbers of statements executed leading to higher weights. Tests with higher weights have more influence on f and p, which are real numbers rather than integers. The overall ranking is computed incrementally, with weights being adjusted at each stage. For single-bug programs, performance of optimal metrics is not affected. However, performance is improved for multiple-bug programs using the best known metrics. This weighted incremental approach could be adapted to predicate or path spectra.

IX. Further work

For systems which collect statement spectra, our study indicates potential for improving performance by implementing more accurate heuristics for determining program structure. Even if accuracy of heuristics remains a stumbling block, new predicate metrics such as those proposed here are worth incorporating into systems such as CBI and Holmes (which always determine the program structure accurately), since it seems performance can be significantly improved in this way. Gaining a better understanding of the relative performance of different systems and methods is also a priority — currently the literature contains unresolved inconsistencies.

It is also worth investigating other new predicate metrics. Previously proposed predicate metrics do not perform particularly well and we have only briefly experimented with a few new metrics. Statement metrics attempt to solve the same problem as as similarity metrics in many other areas (measuring similarity or distance between two sets or N-dimensional vectors) hence there is a huge body of relevant work on such metrics. Predicate metrics are different because they measure "similarity" between a pair of sets and a single set. It is harder to get a good intuition for constructing such metrics. The intuition behind Increase seems flawed, for example. Finding an optimal class of metrics, even in the single bug case, may be significantly harder than for statement metrics because we have more information and hence more degrees of freedom when designing metrics.

Multiple-bug performance should also be investigated. Optimal single-bug metrics are rather specialised and don't perform particularly well for programs seeded with two or more bugs [14]. We know of statement metrics which consistently perform well for multiple bugs (and reasonably well for single bugs), but it is unclear how they are best modified to make use of predicate information. We have performed some initial experiments on multiple-bug programs. We obtained similar poor performance for the previously proposed predicate metrics but improved the performance of some excellent statement metrics by adding a multiple of Context. Experiments on larger benchmarks are also desirable to see how the approach scales.

X. Conclusion

The predicate-based and statement-based approaches to spectral bug localization collect equivalent raw data from program executions. However, the way the data is aggregated differs. More information is retained by the predicate method than the statement method as it combines spectral information from pairs of program points. We have shown that this additional information can improve performance of bug localization. Collecting statement spectra is easier from a practical perspective due to the availability of various profiling tools. We have proposed a simple heuristic method which combines spectra from consecutive statements in order to reconstruct predicate information.

Experiments using a model-based approach indicate the heuristics are reasonably accurate and we have indicated how they can be improved further. We have evaluated several statement metrics and predicate metrics (which are also used for path spectra) using this these heuristics. This provided a reasonably fair comparison between these metrics. Although previously proposed predicate metrics perform poorly, we have proposed two new predicate metrics which perform very well. For a model single-bug program, they achieve better performance than the best known statement metric, O^p. This is theoretically significant because O^p is known to be the best possible statement metric for some models and is suspected to be the best for the model we chose to use here. The practical results with single-bug programs are not as positive, with performance of the best predicate metrics slightly worse than O^p for the largest benchmark set and slightly better than O^p for the smaller set. However, with better heuristics (or analysing program structure precisely) there is a good prospect of increasing performance. Furthermore, performance of existing systems which use predicate or path spectra is likely to be significantly improved by adopting metrics such as those proposed here. Finally, there is still room to develop even better predicate metrics.

References

[1] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT*. New York, USA: Springer-Verlag New York, Inc. New York, 1997, pp. 432–449.

[2] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," *Proceedings of the 20th ASE*, pp. 273–282, 2005.

[3] R. Abreu, P. Zoeteweij, and A. van Gemund, "An evaluation of similarity coefficients for software fault localization," *PRDC'06*, pp. 39–46, 2006.

[4] W. Wong, Y. Qi, L. Zhao, and K. Cai, "Effective Fault Localization using Code Coverage," *Proceedings of the 31st COMPSAC*, pp. 449–456, 2007.

[5] L. Naish, H. Lee, and K. Ramamohanarao, "A Model for Spectra-based Software Diagnosis," *Accepted for publication in TOSEM*, 2009.

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

[7] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," *Proceedings of the DSN*, pp. 595–604, 2002.

[8] P. Russel and T. Rao, "On habitat and association of species of anopheline larvae in south-eastern Madras," *J. Malar. Inst. India*, vol. 3, pp. 153–178, 1940.

[9] R. Abreu, P. Zoeteweij, and A. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *TAICPART-Mutation 2007*. Windsor,UK: IEEE Computer Society, 2007, pp. 89–98.

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

[11] A. Gonzalez, "Automatic Error Detection Techniques based on Dynamic Invariants," Master's thesis, Delft University of Technology, The Netherlands, 2007.

[12] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, "Scalable statistical bug isolation," *Proceedings of the 2005 ACM SIGPLAN*, vol. 40, no. 6, pp. 15–26, 2005.

[13] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[14] L. Naish, H. J. Lee, and K. Ramamohanarao, "Spectral debugging with weights and incremental ranking," in *16th Asia-Pacific Software Engineering Conference, APSEC 2009*. IEEE, December 2009, pp. 168–175.

[15] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[16] W. Wong, J. Horgan, S. London, and A. Mathur, "Effect of test set minimization on fault detection effectiveness," *Proceedings of the 17th ICSE*, pp. 41–50, 1995.

[17] S. Ali, J. Andrews, T. Dhandapani, and W. Wang, "Evaluating the Accuracy of Fault Localization Techniques," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 76–87.

[18] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 34–44.

[19] T. Ball and J. Larus, "Efficient path profiling," in *micro*. Published by the IEEE Computer Society, 1996, p. 46.

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

[21] W. Eric Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *Journal of Systems and Software*, 2009.

[22] H. J. Lee, L. Naish, and K. Ramamohanarao, "Effective Software Bug Localization Using Spectral Frequency Weighting Function," in *Proceedings of the 2010 34th Annual IEEE Computer Software and Applications Conference*. IEEE Computer Society, 2010, pp. 218–227.