

# In Quest of the Science in Statistical Fault Localization<sup>†</sup>

W. K. Chan

City University of Hong Kong  
Tat Chee Avenue  
Kowloon Tong, Hong Kong  
wkchan@cs.cityu.edu.hk

Yan Cai

City University of Hong Kong  
Tat Chee Avenue  
Kowloon Tong, Hong Kong  
yancai2@student.cityu.edu.hk

## Abstract

Many researchers employ various statistical methods to locate faults in faulty programs. Like other researchers, we sometimes have made mistakes in the quest of making statistical fault localization both practical and scientific. In this experience report, we reflect on our work done on this topic, organize our isolated experiences in the format of models and errors, and cast them in the context of statistics.

**Keywords:** statistical fault localization, mistakes, review, statistics.

## 1. Introduction

Program debugging aims to locate faults in faulty programs, repair them, and confirm the identified faults having been removed [39]. Over the past few decades, a significant progress in research has been achieved. Nonetheless, program debugging is still lengthy and labor-intensive. Together with testing, they consume at least 30% of the total development budget in a typical software development project [31]. Many researchers [1][11][24][26][27][28][32][43][45][55] have proposed or are investigating various automated or semi-automated methods with the aim of making this process more efficient, precise, and scalable. *Fault localization*, which means to pinpoint the locations of the faults in faulty software applications, is often deemed as a major bottleneck in this process. Diverse approaches such as traditional gdb<sup>1</sup>-like debugging facilities, program slicing [2][42][50], delta debugging [4][48], switching strategies [24][51], comparisons between a pair of peer executions [3][32][37] or among multiple ones [1][7][26][27][28][53], as well as the applications of formal analysis [10][44], to name a few, have been proposed.

In our investigation on this topic, we have made mistakes in the sake of making statistical fault localization techniques both practical and scientific. In this experience report, we extend our previous presentation [9] by revisiting, consolidating, and reflecting on the development process of our ideas in statistical approaches to fault localization. In particular, we focus our attention on the statistical aspect of this class of fault localization methods. Our observations on our previous ideas as well as the ideas presented in the work done by other authors are that there is still inadequate awareness of some well-known and general issues:

---

<sup>†</sup> This work is supported in part by General Research Grant of Research Grants Council of HKSAR (project no. 111410).

<sup>1</sup> [www.gnu.org/s/gdb](http://www.gnu.org/s/gdb)

- In Sections 4 and 5.1, we point out that many pieces of existing work do not address the issue of coincidental correctness [40] (which, an execution of a program may not always manifest into a failure even though a fault has been activated in the execution), even though this issue is “well-known” [34] in the software testing community that can be dated back to at least 1993. For instance, a fault may lie in a code library or fragment that is popularly executed by many executions. As pointed out by Denmat et al. [13], *Tarantula* [25], which relies on differencing the coverage achieved by a set of passed and failing executions, is intrinsically difficult to locate such a fault. We have further examined the 33 statement-level techniques summarized by Naish et al. [30], but could not observe that they are intrinsically able to locate such a fault either. Interestingly, many of the 33 techniques (and their variants) were proposed after 1993 and published in software engineering publication venues. It appears to us that many software engineering researchers (including us) recognize that coincidental correctness is a problem when using execution profiles with test verdict indicting no failure, and yet, there is inadequate awareness to address this problem in their fault localization techniques. Moreover, although techniques such as [40][53][54] have been developed to address this problem, yet we observe that later techniques may not learn from this experience.
- A technique may deliberately trade between precision and scalability [45]. In Section 5.2, we show that it is possible to work from a model-based perspective to eliminate a class of background noise inherent in such a tradeoff. Previous work has developed many similarity coefficients (see Table 1 of [55] for example) that some capture noise-reduction terms [45] in their similarity coefficients, yet subsequent work may not be aware of the significance of such noise-reduction terms, and resolves to develop a new similarity coefficient without them.
- In Section 5.3, this report revisits the needs of using distribution-free statistical inferences [55]. It further points out that it is not always possible for every feature instance (e.g., every executed statement in a program over a given test suite) to receive a sufficient number of samples to scientifically apply the same statistical inference technique with an assumed underlying or prior distribution on the samples. This experience report also argues that sometimes, the amount of samples for a subset of feature instances can be so inadequate that even applying a non-parametric approach is problematic. Without knowing whether an inference on every feature instances can be applied with confidence to draw a statistical assertion, a technique may generate a list of feature instances (e.g., statements) that are associated with some inference values with diverse degrees of confidence intervals, and yet some of the feature instances in the list may not be reliably used. If this is the case, suggesting the list to the users poses a huge risk on the reliability of the suggest locations of the faults. For instance, *SOBER* [28] and *DES* [56] did not validate whether each predicate may apply their “z-value” hypothesis testing formulas.

It is worth noting that the target of this experience report is not to concisely summarize the previously developed techniques or point out their imperfections. Rather, this report aims to raise the awareness of the research community about the scientific issues when developing techniques that solve practical problems. Hence, for brevity, in the sequel, this experience report only summarizes the intuitive ideas of individual work.

The rest of this report is organized as follows. Section 2 gives the preliminaries of statistical fault localization. It provides a basis for our discussion on how a statistical

instrument may affect the precision of a chosen fault localization idea that uses statistical inferences to suggest faulty positions in programs with confidence. Section 3 outlines a code-based example to ease us to discuss on various program debugging issues in the subsequent two sections. Section 4 explains the needs to develop statistical structural models to better understand fault localization for program debugging, which further leads us, in Section 5, to discuss on some issues about errors in the fault localization process. In particular, we show how such a model addresses the problem of coincidental correctness in Section 4, and we look at the errors that may be introduced in fault localization during the process of data collection in Section 5.1, data measurement in Section 5.2, and the application of statistical inference in Section 5.3. Finally, Section 6 concludes the report.

## 2. Statistical Fault Localization: Preliminaries

In this paper, we generally refer to the use of statistical methods as the major devices in tackling the fault localization problem as *statistical fault localization*. As explained by Liu et al. [28], SOBER [28], Tarantula [15], and CBI [27] require the collections of the statistics from executions. In [15], Eagan et al. also explain that Tarantula was developed in the background of having a large test suite. Because of the statistical nature, if there is inadequate amount of samples, a statistical test may draw a raw false negative conclusion due to inadequate statistical power. Hence, we argue that it is inappropriate to apply such a strategy on a set of data that contains an inadequate amount of samples such as merely having one failure sample and one non-failure sample. We firstly assume that we are given with a fairly large amount of samples, and later in Section 5.3, we revisit this assumption.

We firstly define a basic strategy to locate faults statistically, followed by discussing a few more subtle issues that exist in program debugging, which separate our discussion from a discussion on merely applying a statistical approach on a dataset. In a simplest form, a basic strategy coined as **BasicStrategy** to locate faults in a faulty program  $P$  relies on a dataset  $D$ , in which each element is an execution profile (e.g., [49]) about  $P$  that captures some information about a particular execution path and its test verdict.

A test verdict is to dictate whether the corresponding execution profile is earmarked as failed. We refer such an execution with a failed test verdict as a *failing* execution, and *non-failing* otherwise. In practice, a check on the program outputs or a detected violation of some assertions or invariant constraints can be used to determine such a test verdict. For instance, one may mark the test verdict of an execution as failed if a trustworthy failure report about the execution is available.

In many existing statistical fault localization techniques, various *program entities* such as statements [15][25], branches [53], predicates [27][28], path fragments [11][40], information flow [29], or their combinations [36] of a program exercised by an execution can be collected in an execution profile. To ease our subsequent discussion, we refer to each type of program entity (including its possible subtypes or supertypes) as a *feature*. In relation to this, every particular program entity in a program is called a *feature instance*. For instance, the statement  $s_9$  in Figure 1 can be regarded as a feature instance by a strategy.

Based on  $D$ , **BasicStrategy** may directly pick a similarity coefficient [22][30] or multiple ones [12][36] to measure the strength of co-occurrence between the presence of an object indicated by the test verdicts and the presence of an element indicated by the captured execution profiles. For instance, one may initialize **BasicStrategy** by choosing the Jaccard

index [21] as its similarity coefficient and the statement coverage achieved by each execution [15] as the samples for conducting statistical inference over  $D$ .

The application of a similarity coefficient may not be straightforward. For instance, the Jaccard index [21] is defined as  $Jaccard(X, Y) = |X \cap Y| \div |X \cup Y|$ , where both  $X$  and  $Y$  are sets. Suppose that the feature instances are statements. In essence, the Jaccard index measures the proportion of statements that are commonly executed by  $X$  and  $Y$ . The empirical results and discussions reported in [26][32] on a technique coined as *set-intersection* show that using such a strategy is significantly less effective to locate faults than the other techniques reported in the same experiments.

Some existing proposals such as [1][22] use an alternative formulation for the comparison for objects that their attributes are restricted to be binary. Such formulation can be clarified with the aid of a contingency table [46] as shown in

Table 1<sup>2</sup>, in which Object  $X$  refers to as the coverage achieved by all execution profiles on a particular feature instance, and Object  $Y$  refers to as the test verdicts of the executions on the same feature instance. In brief, the values  $a$ ,  $b$ ,  $c$ , and  $d$  refer to the total numbers of matched cases in the four possible combinations of whether or not a feature instance is covered and whether or not a corresponding execution profile is failing, respectively. In data mining [17], the *Jaccard coefficient* can be expressed as  $(b + c) \div (a + b + c)$ , which is the same as  $1 - a \div (a + b + c)$ . Hence, without the loss of generality, the Jaccard coefficient can be simply referred to as  $a \div (a + b + c)$ . For instance, the Jaccard technique described in [1] uses this reversed form.

According to Han et al. [17], this coefficient is said to be useful for *asymmetric* data, that is, under the assumption that the presence of data is more interesting than their absence. In fault localization, the presence of a program entity in an execution and the presence of failure associated with the same execution are more interesting than the opposite combination. The Jaccard coefficient accounts for this asymmetric property by excluding the component  $d$  from its formula. With this understanding, we firstly let  $s$  be  $a + b + c + d$ , and rewrite the Jaccard coefficient into  $(a/s) \div (a/s + b/s + c/s)$ . Each term in the Jaccard coefficient can be regarded as the probability of the occurrence of the corresponding event in the sense of frequency probability, and therefore, a composition of four such probabilities. Hence, an alternative view is that the Jaccard coefficient uses the parameters  $a$ ,  $b$ , and  $c$  to characterize  $D$ . We use this alternate view to link it to our discussion on distribution-free hypothesis testing in Section 5.3.<sup>3</sup>

---

<sup>2</sup> It is worth noting that contingency table is also known as *crosstab*, which has been directly used for statistical fault localization (e.g., [43]).

<sup>3</sup> For simplicity, in this paper, we do not go into the Bayesian probability theory to estimate and adjust the prior probabilities of event occurrences so that a technique may further be adapted to handle multiple faults in a program.

Table 1. Contingency table for binary data: Suppose that the feature instance in question is  $s$ . The terms  $a$ ,  $b$ ,  $c$ , and  $d$  refer to the number of executions that each covers  $s$  and is failing, does not cover  $s$  and is failing, covers  $s$  and is non-failing, and does not cover  $s$  and is non-failing, respectively.

|            |            | Object $X$ |             |         |
|------------|------------|------------|-------------|---------|
|            |            | Covered    | Not Covered | Sum     |
| Object $Y$ | Failed     | $A$        | $B$         | $a + b$ |
|            | Not Failed | $C$        | $D$         | $c + d$ |
|            | Sum        | $a + c$    | $b + d$     |         |

Once the above strategy computes the strengths of the above correlations, the next procedure is to organize (or prioritize) the corresponding feature instances with the aim of presenting the corresponding program entities that can help developers to locate faults most effectively first. In many empirical studies of statistical fault localization research (e.g., [24][26]), this procedure is *approximated* by a measurement on the relative rank of the first statement or predicate that is nearest to a fault position of each fault in a program. To make our paper more focused, the ranking part of statistical fault localization techniques is not within the scope of this experience report.

Nonetheless, if we want to present some empirical results, we are unavoidable to discuss on the ranking effectiveness of statistical fault localization techniques. We use the interpretation of *code examination effort* for this purpose.

In many validation experiments, the amount of code examination effort is measured via a metric known as *expense* [24]. Supposed that the feature instances used by a technique in an experiment are *statements*, then this metric (i.e., expense) is defined as the ratio of the executable statements received the same or higher ranks than an executable statement that is nearest to the fault position over the total number of executable statements in  $D$ . In other words, the higher this metric indicated, the better a technique in relation to the other techniques in the same experiment is. It is worth noting that the use of *expense* can only measure *one* particular aspect of the fault localization effectiveness of a technique. It is also still unclear whether this metric maintains a linear relationship with the original target of measuring *code examination effort*. Furthermore, a difference of, say, 10% in different ranges of code examination efforts to locate the same faults or the same amount of faults may not be directly comparable. Despite its limitations, in the rest of this article, we simply use this metric to stand for code examination effort when we discuss on empirical data.

Last, but not the least, we should be aware of the underlying inspiration of applying such a technique to fault localization for program debugging: A *program spectrum* is a *distribution of data* [16] on a feature instance such as path or statement derived from the execution profiles [33]. Raps et al. [33] pointed out that contrasting such a pair of path spectra generated by two versions of the same program (e.g., intuitively, two “closely related” programs both in terms of program structures and semantics) by a similarity coefficient provides insights on the program behaviors between the two versions to facilitate software maintenance. Harrolds et al. [16] generalized the concept of path spectra to program spectra, and empirically compared different types of program spectra and their individual correlations to program misbehavior in between a program and its modified program. Later,

Eagan et al. [15] combined the concept of program spectrum and fault localization to formulate *Taranulta*. The fault localization effectiveness of *Taranulta* was first reported in [25], which is probably the first empirical evidence in the public literature that contrasting two program spectra derived from a set of non-failing executions and a set of failing executions, respectively, of the *same* program can be valuable to locate faults in *the* program.

### 3. Running Example

Is it adequate to merely having a strong fault-failure correlation [54] between the execution information associated with a feature instance and the test verdicts? Let us consider the piece of code fragment shown in Figure 1 first.

In Figure 1, the program always crashes when any of its executions exercises the line *s9* due to the occurrence of a null pointer exception. Let us hypothesize that the root cause of this failure is at line *s3*, which is supposedly assigned a value from *pGiven*, but is mistakenly implemented to receive a null value<sup>4</sup>. This null value is kept by the variable *pA* at *s3*; and the corresponding infected state of this variable in an execution may not necessarily infect a memory location pointed by the variable *pB* at line *s6*, which is further referenced at line *s9*, triggering a failure as a result.

On the one hand, the fault is at *s3*, and yet some executions of the program may exercise *s3* only (e.g., skipping *s6* and *s9*). It means that some executions do not result in null pointer exceptions at *s9*. On the other hand, every execution that exercises *s9* must associate with a failure. The line *s9* strongly correlates with the failures indicated by *D*, but the line *s3* does not.

The prime utility of statistical fault localization is to pinpoint *s3*, which is the root cause of the failure. However, in the example, *s9* is determined to be more failure relevant than *s3* based on the relative strengths of the measured correlations. Perhaps, this “correlation inversion” scenario could not be perfectly solved.

```
s1.  int foo(int a, int b, POINT pGiven)
s2.  {
s3.      POINT pA = null; // fault
s4.      ...
s5.      if () {
s6.          POINT pB = pA;
s7.          ...
s8.          if () {
s9.              pB[3] = 0;
s10.             ...
s11.          }
s12.      }
s13. }
```

Figure 1. A simple program

---

<sup>4</sup> For instance, an alternative fault hypothesis is that the fault is due to an omission of statements right before *s9*, which misses to implement a guard of referencing a possible null value of *pB* before using the pointer at *s9* because *pGiven* can be a null value.

Let us further look at how an ordinary developer may debug this program. We recall that the class of failure is the null pointer exception. Starting from assessing  $s_9$ , a developer may follow the control flow and data flow of the program to find out that this problematic occurrence of null value is assigned at  $s_6$ , which is originated from  $s_3$ . If this is the case, the developer successfully locates the fault. Based on the finding, the developer can also explain the failures observed at  $s_9$ .

One may further deem such a correlation value as a statistical explanation of the observed failures in  $D$ . Hence, apart from merely increasing the correlation values of  $s_3$  by some means to maximize the above utility of statistical fault localization, another concern is to provide a scientific explanation to developers: Why and how is such a failure relevant value observed at  $s_9$  related to (or explain) another such value observed at  $s_3$ ?

We argue that a second utility of statistical fault localization is to provide a statistical model or framework that can help explain how a fault manifests into a failure of the faulty program as observed in  $D$ . The way may mimic how the above “ordinary developer” locates  $s_3$  from  $s_9$ , and explains the failures. Nonetheless, unlike the above developer, who locates the fault based on the unique fault signature of “null pointer exception”, such a model or framework should be general.

From the above example, readers may further observe that even though  $s_3$  is faulty, some infected states generated by executing  $s_3$  may not lead the program to misbehave. For instance, if the decision outcome of an execution at  $s_5$  is **false**, the above null pointer exceptions cannot be observed. This phenomenon is generally referred to as *coincidental correctness* [40], which affects the test verdict assignments in the execution profiles captured by  $D$ . For instance, some non-failing executions may contain many prolific data that are failure relevant but non-observable due to the presence of coincidental correctness.

In the above sense, there are errors in the construction process of  $D$ . Are there errors in executing the other parts of typical statistical fault localization strategies? In the next two sections, we are going to share our experience further and discuss on these issues.

#### **4. Modeling Fault-Failure Structure Statistically**

A dataset  $D$  can be initialized to contain much information such as the consecutive pairs of statements executed or a sub-path for each execution. Based on such “link” information, one may construct a structural model that describes the fault-failure correlations distributed among the program entities of a program concerned. With such a structure, a network-based inference on the structure (as we are going to describe in this section) can be used to alleviate the problem of locating faults that are popularly executed by the executions in  $D$ .

We first hypothesize that a measured fault-failure correlation value via a similarity coefficient represents a statistical fact presented in  $D$  about the observed failures. Although a strategy can apply any similarity coefficient in practice, yet under this hypothesis, we argue that it is preferential to apply those similarity coefficients that can be linked to the probability theory and have clear physical meanings.

For instance, in [53], the mean occurrence frequency of an edge for failing executions is contrasted with that of the same edge for non-failing executions in  $D$ . In [27], a variant of the former kind is compared to that in the whole set of executions, which is probably the first work that contributes to this kind of modeling concept in statistical fault localization for

program debugging. In essence, the ideas of these two pieces of work are to measure and *specify* the change in the probability of the same program entity occurred between the two sets of execution profiles used in their corresponding models. Because each specified change is a component in a model, we refer to such a change as an *infected state*.

Simply specifying the infected states of a model is insufficient because such states may associate with different flows of information along the execution paths in a program. To model such flows collectively, one may consider constructing a graph to denote how such the corresponding program entities are organized in the faulty program and how such infected states may be related to them.

For instance, we may take the set of such statistical facts (i.e., the observed infected states) on “links” as a starting point. We may connect an end node of one link and a start node of another link if the labels on these two nodes refer to the same basic blocks in the code of the faulty program that  $D$  bases on. If every link represents a control-based dependence of a program, then a resultant graph is a control flow graph, in which each edge is superimposed with the above-mentioned statistical facts [53]. Similarly, if every link is a definition-use association of a program, it becomes a data flow graph. Other superimposed dependence graphs can be constructed similarly (e.g., see Baah et al. [5] for an alternate formulation).

There are challenges in using such a graph for fault localization. Let us consider a simple example that such a graph  $G$  comprises of three nodes  $\{a, b, c\}$  and three directed edges  $\{(a, b), (b, c), (c, a)\}$ . Suppose also that the three edges are earmarked with non-identical values and they represent control flow edges.

One way to “execute”  $G$  statistically is to “follow the edges” to visit its nodes in turn. Similar to how an error in an execution may propagate from a program state to the other program states in the execution, the error on one edge may propagate from one edge to another edge when a program execution is simulated on  $G$ . Hence, the *measured* statistical fact of every such node should represent the sum of the original statistical facts of the node and the fractional original statistical facts of the nodes propagated via the graph and reached this node. In other words, if one simply uses the measured statistical fact on a node (or edge) as if it were *the* original statistical fact for the node (or edge), this is an error in such approximation.

To iron out the fault-failure correlations of individual nodes more precisely, we apportion the measured statistical facts of one node to all those nodes that are directly connected by the incoming edges of the former node, which is analogous to rewinding the above “execution” on  $G$  [53][54]. In the example, the graph  $G$  is cyclic. Hence, at the end, the proportion of the statistical fact of a node apportioned to the other nodes reaches the node itself. Moreover, the node also receives the proportions from the other two nodes as well. Through Gaussian elimination [41] or its approximation such as dynamic programming, finally, all the three nodes on  $G$  receive the same fault-failure correlation values. Readers may observe that such a propagation method is (1) not only capable of re-distributing the fault-failure correlations among nodes in a network-based model (2) but also uncover the next level of “facts” for further statistical analysis. In general, such a network-based motif is a general directed graph, and the correlation values after propagation may be non-identical. Intuitively, the set of equations generated via such a propagation model can help model how a failure observed at a



node (e.g., statement  $s_9$  in Figure 1) can be backtracked to another node (e.g.,  $s_6$  in Figure 1) statistically.

However, for program debugging, the situation can be more complicated. An execution may crash at a particular program state. A consequence is that one should not simply follow the edges to apportion the statistical facts without considering whether such a propagation carries a precise physical meaning [54].

Table 2. Proportion of faults located by different statistical fault localization techniques (taken from [54])

| Code Emanation Effort | Proportion of faults located |             |            |          |          |            |              |                |         |                |
|-----------------------|------------------------------|-------------|------------|----------|----------|------------|--------------|----------------|---------|----------------|
|                       | Tarantula [26]               | Jaccard [1] | Ochiai [1] | SBI [47] | CBI [27] | SOBER [28] | DES-CBI [56] | DES-SOBER [56] | CP [53] | BlockRank [54] |
| 5%                    | 0.30                         | 0.30        | 0.30       | 0.30     | 0.12     | 0.01       | 0.07         | 0.05           | 0.39    | 0.39           |
| 10%                   | 0.40                         | 0.41        | 0.41       | 0.41     | 0.23     | 0.10       | 0.16         | 0.10           | 0.47    | 0.49           |
| 20%                   | 0.66                         | 0.66        | 0.66       | 0.66     | 0.29     | 0.24       | 0.27         | 0.22           | 0.65    | 0.67           |
| mean                  | 0.20                         | 0.20        | 0.20       | 0.20     | 0.41     | 0.43       | 0.43         | 0.43           | 0.18    | 0.17           |
| stdev                 | 0.24                         | 0.24        | 0.24       | 0.24     | 0.28     | 0.24       | 0.27         | 0.24           | 0.21    | 0.20           |

Table 2 shows an empirical comparison between two network-based techniques (i.e., CP and BlockRank)<sup>5</sup> and some other techniques on a suite of four medium-scale subjects (*flex*, *grep*, *gzip*, and *sed*) to locate faults in 110 single-fault versions with one suite of test cases for each subject. All of them were downloaded from SIR [14]. Each column of the table shows a technique, and each row shows the proportion of faults located by the corresponding technique with a certain threshold of code examination effort (see Section II for the definition) as indicated as the corresponding row heading. The bottom two rows of the table show the mean code examination effort (*mean*) and their standard deviations (*stdev*) to locate all the faults in the experiment by each technique. For instance, with 5% code examination effort, these two network-based techniques can be at least 0.09 (i.e.,  $0.39 - 0.30$ ) in absolute term (or 30% in relative term) more effective than the other techniques in the same row to locate the faults from these subjects.

Intuitively, as some errors have been alleviated, such a network-based technique can be more predictable. This intuition is consistent with the observations in the bottom row that the standard derivations on code examination effort achieved by the two techniques to locate a fault are smaller than the corresponding values of the other techniques in the table.

A statistical model for fault-failure propagation may be more complicated if a graph  $G$  represents multiple kinds of dependencies in a program or a family of programs. For instance, for graphs representing concurrent programs or definition-use associations for

<sup>5</sup> We note that the essence of CP and BlockRank is to factor in the influences of all the nodes and edges in the same graph on individual fault predictor location, yet some readers may prefer to view that these techniques work on a graph-based motif rather than individual (and standalone) fault predictors. Hence, one may view them as graph-based techniques. We accept both viewpoints.

memory locations, multiple edges may be exercised at each execution step. A further generalization or adaptation of the above propagation models is necessary to handle such cases. Moreover, in Table 2, the overall fault localization effectiveness for different techniques may become almost indistinguishable after certain ranges of code examination efforts. It indicates that some faults seem to be more effective to be detected by strategies that are not network-based. They suggest that there are some error dimensions that the propagation model has not captured but it may interference with the propagation strategies. What are they? In the next section, we are going to reveal some of them based on our experiences.

## 5. Thriving on Errors

In this section, we firstly point out some observations or misconceptions that many researchers may have recognized, but as we are going to explain, many statistical fault localization studies have overlooked them. Then, we share our experiences in addressing them.

### 5.1 Data

We firstly look at a problem in the construction process of the dataset  $D$ .

**Observation 1:** Activating a fault in an execution does not imply observing a failure.

This observation is well-known in the software testing community for decades [34]. The implication of **Observation 1** is that some non-failing execution may have activated some faults as well. Nonetheless, the location of a fault cannot be known before it is located. Intuitively, such errors in  $D$  are hard to avoid.

In the “ordinary developer” example presented in Section 3, the developer applied some knowledge about “null pointer exception” to track the flow of a program to locate  $s_3$  as the root cause of an observed failure at  $s_9$ . This kind of knowledge can be codified as fault patterns. For instance, in FindBugs [18], some static fault patterns have been characterized and used in the tool to detect anomalies (e.g., the null pointer bugs [19]). Similarly, many patterns of concurrency faults [38] have been used in numerous static or dynamic concurrency fault detectors.

An execution profile may contain sub-paths. If this is the case, such a sub-path may match against some fault patterns. In [40], we use a heuristics to utilize this concept for statistical fault localization: if there is a match, then the execution fragment is kept; otherwise, the fragment can be discarded. Both the comprehensiveness of the suite of fault patterns and how well the suite of fault patterns is coupled with real faults are important. These questions remain to be answered. It appears to us that the situation is analogous to the first-order mutants in mutation testing that killing them may effectively expose the actual faults or higher-order mutants of the same program.

Previous research such as [32] has shown that a non-failing execution trace that is *nearest* to a failing execution trace can help to locate faults. In relation to the idea mentioned in the last paragraph, we have developed a suite of 12 fault patterns for this purpose [40]. By applying the whole suite of patterns to *every* execution profile, a fragment of an execution may be able to precisely match with a particular fault pattern. If this is the case, the part of the execution profiles for this particular execution fragment is treated as a new execution

profile, and the latter is added to a constructing dataset  $D'$  (initially empty) with a test verdict the same as that of the original execution profile.

In this way,  $D'$  is by design to contain no execution fragment that cannot be matched with at least one fault pattern; and at the same time, an execution trace that can match with multiple fault patterns will generate multiple execution profiles in  $D'$ . These two types of actions correspond to cleansing  $D$  and refining  $D$ , respectively. Hence, in statistical inference, the execution fragments can be compared more reliably with respect to the set of fault patterns.

If  $D$  incurs the problem of coincidental correctness and contains many samples, intuitively, both failing executions and non-failing executions that can match the same fault pattern at the same position in the program may exist. The corresponding pairs of execution fragments in  $D'$  (i.e., after the application of fault patterns) should be the nearest (which is in fact identical with respect to the concerned fault pattern) because they match the same pattern at the same position. At first sight, such a pair of execution fragments achieves the same coverage statistics with respect to a fault pattern, and hence, provides little hints to locate faults. At a closer look, suppose that a failed execution in  $D$  manifests into multiple execution fragments  $D'$ , and they contain a fault. This type of coverage refinement may enhance the amount of samples for failure scenarios (i.e., the component  $a$  in Table 1), potentially improving the probability to locate the fault in the corresponding code fragment with respect to the fault patterns (and their correlated higher-order fault pattern instances and actual faults).

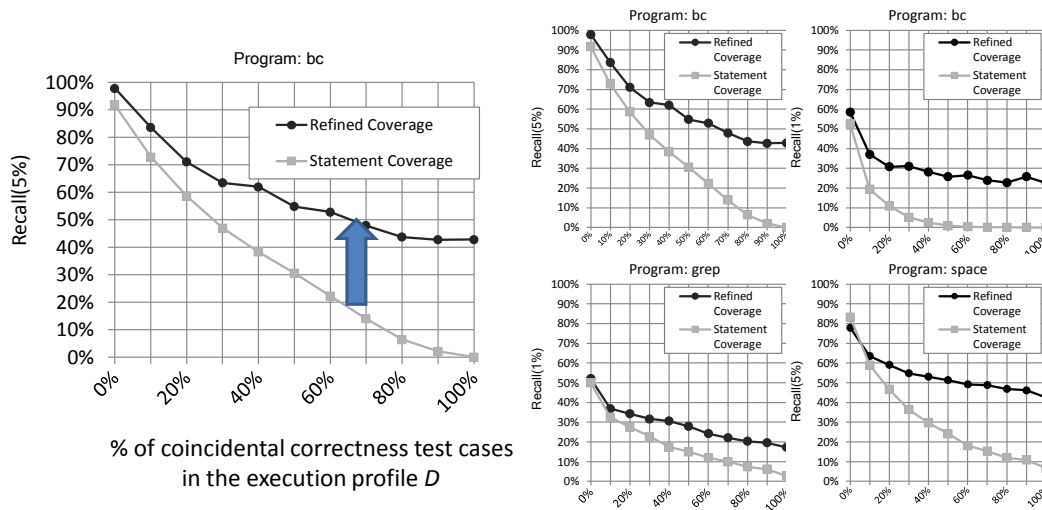


Figure 2. The change in fault localization effectiveness between before applying the coverage refinement patterns on top of Tarantula and after applying it (adapted from [40]): In each plot of the figure,  $\text{Recall}(k\%)$  refers to the proportion of faults located with  $k\%$  code examination effort. Moreover, the statement coverage achieved by each test case is used as an execution profile in  $D$ . The lines labeled as “Statement Coverage” and “Refined Coverage” are the fault localization effectiveness achieved by Tarantula using  $D$  and  $D'$ , respectively, on each program (bc, grep, and space are shown in the figure) with 1000 mutants that a failure of each mutant can be detected by at least one test case.

Figure 2 shows the results of applying the whole suite of fault patterns on the datasets for three medium-scale programs (`bc`, `grep`, and `space`). The subjects were downloaded from SIR [14]. Although the improvement in effectiveness measured as the proportion of faults located within the same amount of code examination effort varies, and yet one still observes that, by cleansing and refining  $D$ , on average, a statistical fault localization technique can be more accurate as the amount of confidential correctness in  $D$  increases.

Such a pair of execution fragments has been shown to be useful when it is used alone [32] or statistically [40]. Nonetheless, it is certainly that in some scenarios, the above-mentioned cleansing and refinement strategy may remove some execution profiles that can succinctly and effectively used by a fault localization strategy to locate faults. It appears that there are some deeper reasons to be uncovered. It is interesting to explore further by the research community.

## 5.2 Similarity Measurement

We now turn our focus on the measurement process on the features that a strategy applies.

*Observation 2:* Given a feature, measuring the correlation between  $D$  and each instance of most features is inaccurate irrespective to the similarity coefficient used.

Take Figure 1 for example. If we merely measure the strength of the fault-failure correlation between the execution information in  $D$  for  $s_9$  and the test verdicts, we have ignored the fact that an execution cannot merely execute  $s_9$ . Rather, every such execution may go through some other feature instances such as  $s_3$  and  $s_6$ .

Hence, each test verdict of an execution profile is probably associated with a set of feature instances rather than the one being measured only. In other words, the samples captured in  $D$  with respect to individual feature instances such as individual statements being measured *are not independent* to one another.

In terms of probability theory, they are simply not independent events. If this is the case, mathematicians use conditional probability to handle such cases. In this connection, if we merely measure the direct fault-failure correlation for a feature instance, there are *white noises* in the background, probably in every such measurement. Echoing the message in Section 5.1, researchers may also develop some active strategies to reduce the adverse effects of such issues on fault localization effectiveness.

Let us consider the following formulation to ease us to relate our discussion to the issues on similarity coefficient:

Let  $A(e)$  be the probability of an execution with an observed failure and it executes  $e$ ,  
 $B(e)$  be the probability of an execution without observed failure and it executes  $e$ ,  
and,  
 $C(e)$  be the probability of execution executes  $e$ .

Then, we can compute  $T(e)$  as  $(A(e) - C(e)) - (B(e) - C(e))$ , which can be further simplified into  $A(e) - B(e)$ .

The terms  $A(e)$  and  $B(e)$  as well as their variants have been directly used as the similarity coefficients in many existing fault localization techniques. Examples include Jaccard [1], Tarantula [26], and their variants.

The term  $A(e) - C(e)$  represents how much the probability of observing  $e$  has been changed from the (noisy) background  $C(e)$  to the set of execution profiles that each is earmarked as failed.  $C(e)$  can be considered as the white noise observed in  $D$ , and this idea of characterization has been proposed in [27].

Opposite to the interpretation of the term  $A(e) - C(e)$ , the term  $B(e) - C(e)$  represents how much the probability of observing  $e$  has been changed from the (noisy) background  $C(e)$  to the set of execution profiles that each is *not* earmarked as failed. Hence, if the term  $A(e) - C(e)$  represents the evidences of observing a failure on top of the white noise, then the term  $B(e) - C(e)$  represents the evidences of observing no failure on top of the background with identical white noise.

The difference between these two terms ( $A(e) - C(e)$  and  $B(e) - C(e)$ ) thus represents the evidence of changing from observing no failure to observing a failure in the presence of (noisy) background. Interestingly, this difference  $T(e)$  can be simplified into  $A(e) - B(e)$ , which is independent to the background  $C(e)$ . In CP [53] and Minus [45], we use such noise-reduction formulas. Table 3 shows the empirical results [45] of Minus, which is initialized on top of Tarantula [15], on a suite of three medium-scale Java programs (jtopas, xml-security, and ant) with 86 versions with real faults using statements as the feature instances for measurement. The row and column headings of the table can be interpreted similarly as Table 2. Readers may compare the columns for Minus and Tarantula to observe the overall effect on adding such a term in a fault localization similarity coefficient.

Table 3. Effect of Having Noise Reduction Terms (adapted from [45]).

| Code Examination Effort | Proportion of faults located |             |            |          |                |
|-------------------------|------------------------------|-------------|------------|----------|----------------|
|                         | Minus [45]                   | Jaccard [1] | Ochiai [1] | SBI [47] | Tarantula [26] |
| 5%                      | <b>0.1034</b>                | 0.1264      | 0.1264     | 0.0690   | 0.1034         |
| 10%                     | <b>0.2644</b>                | 0.2644      | 0.2644     | 0.1494   | 0.2184         |
| 20%                     | <b>0.3448</b>                | 0.3448      | 0.3448     | 0.2759   | 0.3333         |
| mean                    | <b>0.4539</b>                | 0.4687      | 0.4673     | 0.5830   | 0.4739         |
| stdev                   | <b>0.3517</b>                | 0.3628      | 0.3624     | 0.3837   | 0.3537         |

The above measurement of fault-failure correlations is also affected by the chosen feature and where in the program the feature instances are measured. In some cases, the locations to measure such feature instances introduce imprecision, rendering such measurements imprecise.

For instance, in [56], we observe that the computation taken by a program to compute a decision value of a decision statement at the source-code level is subject to the complexity of the decision expression. In C, C++, or Java source code, there are statements expressing short-circuit logics. The above kind of measurement on such a statement is not precise

enough. By not mixing up different evaluation sequences (which is free from short-circuit expressions) [56], one can obtain data at a finer granularity for every execution profile.

The empirical study reported in [56] uses four medium-scale programs (*flex*, *grep*, *gzip*, and *sed*) with 110 faulty versions and the seven Siemens suite with 126 faulty versions as subjects [14] to validate this idea. The result shows that the improvement on using finer coverage granularity for predicate-based statistical fault localization may vary significantly from a subject to another subject, and sometimes do not improve the original techniques. We have reflected on this finding, which lead us to study the issue that we are going to discuss in the next sub-section.

In some intermediate-representation levels of a program, there is no room for short-circuit logics to take any effect. One may still improve the measurement precision by applying the concept of evaluation sequence to construct a sequence of atomic predicates (or a subpath) as a feature instance. For instance, in [45], the code listing of a program is proposed to be partitioned according to such feature instances statically. Moreover, we used *Minus* with all such sequences as the set of feature instances to have developed a technique known as *MKBC* [45]. Figure 3 shows the empirical result [45] of *MKBC* in the same experiment as what we have described to produce Table 3. In the table, *MKBC* is 10% more effective than the other techniques in the same plot consistently across a majority range of code examination effort.

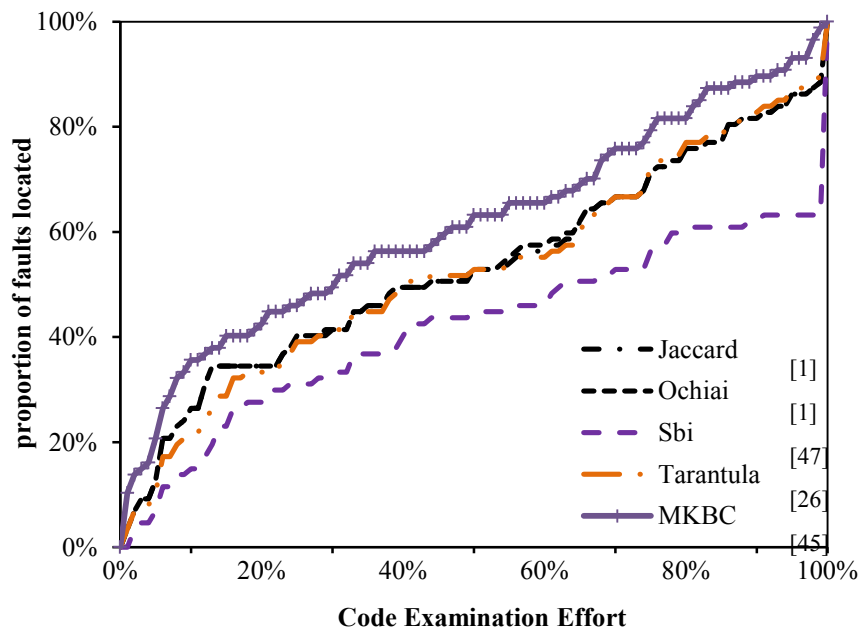


Figure 3. Effect on having more accurate feature measurement (adapted from [45])

Last, but not the least, we recall that we have discussed the Jaccard coefficient in Section 2. A coefficient similar to the Jaccard coefficient is affected by the sample size [22]. This finding shows that if a technique compares the coefficient values provided by such a coefficient on two feature instances, the relative size of the samples for the two feature instances should be considered. In theory, the size can be non-identical. Baah et al. [6] recognize the problem of potential presence of cofounding factor in measurement, and propose a methodology to address such a problem.

### 5.3 Statistical Inference

In this section, we study on the use of statistical inference in the measurement of similarity. In statistics, broadly speaking, there are parametric and non-parametric tests. Non-parametric tests are distribution free tests, and tend to be more robust, but their power to distinguish distributions is weaker than parametric tests if both classes are appropriate to the same problem [57] (and see [35], on page 34, for a further discussion). Hence, it is well receptive that one should use parametric tests whenever possible. Liu et al. [28] contribute to propose the first parametric hypothesis testing approach to fault localization to debug programs, and develop a technique known as **SOBER**, which is stated in [28] to be based on normal distribution. Nonetheless, there are still some complications.

In  $D$ , some feature instances, such as a particular statement like  $s_9$  in Figure 1, may contain too few samples to apply a parametric test. One idea to address this problem is to add more samples to  $D$  until every such feature instance has an adequate amount of samples. Nonetheless, this kind of idea does not solve the problem entirely.

**Observation 3:** Using a large amount of data does not imply that a distribution with known parameters such as a normal distribution (i.e., the samples are drawn from a Gaussian population) can be assumed.

Consider the line  $s_6$  in Figure 1. In general, whether this statement can be covered in an execution of a program does not always follow a prior distribution. Furthermore, there is no guaranty that different feature instances in the same program must follow the same prior distribution. Rather, the concerned distribution depends on how a decision expression is implemented at  $s_5$  with respect to its input domain, and how this particular input domain for  $s_5$  is sampled with respect to  $D$ . In general, controlling  $D$  to attain a desirable distribution for every feature instance in a program is very difficult. One of the primary obstacles is the reachability problem encountered in test generation, which is in general undecidable. If it can be done, the distribution can be represented by some parameters of a target distribution, and hence, these parameters can further be used in a fault localization strategy properly. In theory, one may enlarge  $D$ ; in practice, one needs more test cases with definite test verdicts. However, unless the test oracle problem in related to the concerned program has been addressed at a low cost, it does not look like a promising direction.

As it appears difficult to control the composition of  $D$  so that a parametric test can be reliably applied, in [20][52], we have further testified the normality assumption using the Siemens suite [14], which has been popularly used in many testing and fault localization research experiments. We executed each version over its corresponding test pool in entirety, which contains thousands of test cases. We then measured the degree of normality of the distributions of *evaluation bias* [28] of each predicate, which is the proportion of the total frequencies that the predicate has been evaluated to be true. We used the Jarque-Bera test [23] as the normality test. As shown in Table 4, the empirical results [52] indicate that in many cases, the distributions of evaluation bias for individual predicates *cannot* be reasonably considered as normal distributions.

The empirical result seems suggesting the follows: Intuitively, using thousands of test cases even for a small-scale program such as the Siemens programs for the testing purpose is already unrealistically large. Nonetheless, for a scientific application of a parametric

statistical fault localization strategy to address the problem in statistical fault localization, such a quantity is still inadequate.

Table 4. Normality Test on the distribution of evaluation bias (adapted from [52])

| Range of $p$ -value for normality test           | > 0.90 | > 0.50 | > 0.10 | > 0.05 | > 0.01 |
|--|--------|--------|--------|--------|--------|
| Proportion of the most fault-relevant predicates | 0.55   | 0.55   | 0.55   | 0.61   | 0.67   |
| Proportion of all predicates                     | 0.56   | 0.59   | 0.60   | 0.62   | 0.78   |

Apparently, Observation 3 contradicts to the well-known Central Limit Theorem [8] that provides a scientific background to ensure parametric tests work properly in the presence of large amount of samples *even if* the population is non-Gaussian. In fact, for program debugging, it does not.

So far, we have assumed that  $D$  contains many elements. In some cases, only a few samples for some feature instances exist even though the cardinality of  $D$  is large. For instance, in  $D$ , some statements of a program may only be barely executed, and hence, only a limited amount of execution profiles in  $D$  may cover these statements. In statistics, when the number of samples to apply a statistical test is too small such as less than 20, it is more appropriate to use a non-parametric test. An implication is that for a statistical fault localization technique such as **Jaccard** that uses whether a particular program entity (e.g., statement) is covered by an execution profile that is earmarked as failed, a preliminary guideline is that the technique should have at least 20 such profiles for every such program entity so that the technique can be applied scientifically.

Moreover, if  $D$  cannot be asserted to follow an assumed distribution for every (failure relevant) feature instance such as a normal distribution, the use of a parameter for the assumed distribution (e.g., mean values for normal distributions) is *risky* from the statistics point of view. It is primarily because of the effect size for a statistical tests being inadequate, which makes the Type II error in statistical tests as the source of such risks. On the other hand, nonparametric tests can still be applied even though the sample size is not small. Indeed, our experiment reported in [55] shows that a non-parametric technique tends to improve its effectiveness as more samples are available, whereas its parametric counterpart does not.

Hence, in [55], we have argued that in general, it may be more robust (i.e., bearing less risk) for statistical fault localization techniques to use a non-parametric approach to statistical inference to draw a conclusion on whether the difference can be meaningful statistically. We have experimented to compare the fault localization effectiveness of two popular and widely used non-parametric hypothesis tests (Wilcoxon and Mann-Whiney tests) with 33 statement-level strategies (for brevity, we refer readers to [55] for the discussion of these techniques) on the program **space** downloaded from SIR [14]. We used 28 of the available faulty versions because our platform cannot successfully compile the other 10 versions or the test pool in the benchmark cannot reveal any failure on these versions. The results are shown in Table 5 in which columns are techniques and rows are the number of faults located by spending a



certain amount of code examination effort as indicated by the corresponding row heading. We observe from the table that the two non-parametric strategies can be more effective.

Table 5. Comparison of non-parametric test (highlighted) to other strategies (adapted from [55])

| Code examination effort | Code examination effort |           |               |      |             |             |                 |        |                 |       |    |    |                   |         |              |        |        |         |           |        |       |       |          |              |    |  |  |  |
|-------------------------|-------------------------|-----------|---------------|------|-------------|-------------|-----------------|--------|-----------------|-------|----|----|-------------------|---------|--------------|--------|--------|---------|-----------|--------|-------|-------|----------|--------------|----|--|--|--|
|                         | Jaccard                 | Anderberg | Sorensen-Dice | Dice | Kulczynski1 | Kulczynski2 | Russell and Rao | Hamman | Simple Matching | Sokal | M1 | M2 | Rogers & Tanimoto | Goodman | Manning etc. | Euclid | Ochiai | Overlap | Tarantula | Zoltar | Ample | Wong1 | Wilcoxon | Mann-Whitney |    |  |  |  |
| 1%                      | 13                      | 0         | 13            | 0    | 13          | 0           | 0               | 0      | 7               | 6     | 7  | 14 | 2                 | 13      | 6            | 6      | 13     | 0       | 9         | 0      | 13    | 0     | 16       | 18           |    |  |  |  |
| 2%                      | 16                      | 0         | 16            | 1    | 15          | 0           | 0               | 0      | 8               | 7     | 7  | 18 | 3                 | 16      | 7            | 7      | 16     | 0       | 14        | 0      | 15    | 0     | 21       | 20           |    |  |  |  |
| 5%                      | 21                      | 0         | 21            | 1    | 21          | 0           | 0               | 0      | 13              | 10    | 10 | 23 | 3                 | 21      | 10           | 10     | 22     | 0       | 17        | 1      | 19    | 0     | 23       | 22           |    |  |  |  |
| 10%                     | 23                      | 0         | 23            | 1    | 23          | 0           | 0               | 0      | 15              | 14    | 14 | 24 | 9                 | 23      | 14           | 14     | 24     | 1       | 21        | 1      | 22    | 0     | 24       | 23           |    |  |  |  |
| 20%                     | 27                      | 5         | 27            | 6    | 27          | 5           | 5               | 0      | 20              | 18    | 18 | 24 | 13                | 27      | 18           | 18     | 26     | 1       | 26        | 2      | 24    | 5     | 26       | 25           |    |  |  |  |
| 50%                     | 28                      | 26        | 28            | 26   | 28          | 26          | 26              | 20     | 21              | 21    | 21 | 26 | 20                | 28      | 21           | 21     | 27     | 2       | 28        | 16     | 27    | 26    | 28       | 28           |    |  |  |  |
| 100%                    | 28                      | 28        | 28            | 28   | 28          | 28          | 28              | 28     | 28              | 28    | 28 | 28 | 28                | 28      | 28           | 28     | 28     | 28      | 28        | 28     | 28    | 28    | 28       | 28           | 28 |  |  |  |

| Using Non-Parametric Test is More Scientific |    | Wong2 | Wong3 | Ochiai2 | Geometric Mean | Harmonic Mean | Arithmetic Mean | Cohen | Scott | Fleiss | Rogot1 | Rogot2 | Wilcoxon | Mann-Whitney |
|--|----|-------|-------|---------|----------------|---------------|-----------------|-------|-------|--------|--------|--------|----------|--------------|
| 1  | 14 | 11    | 0     | 0       | 0              | 0             | 0               | 13    | 2     | 0      | 7      | 16     | 18       |              |
| 3  | 19 | 15    | 0     | 0       | 0              | 0             | 0               | 15    | 8     | 1      | 8      | 21     | 20       |              |
| 5  | 23 | 20    | 0     | 0       | 0              | 0             | 0               | 20    | 13    | 1      | 16     | 23     | 22       |              |
| 8  | 24 | 24    | 0     | 0       | 0              | 0             | 0               | 21    | 16    | 1      | 21     | 24     | 23       |              |
| 9  | 24 | 25    | 1     | 1       | 1              | 1             | 1               | 23    | 19    | 5      | 23     | 26     | 25       |              |
| 11   | 26 | 27    | 19    | 19      | 19             | 19            | 19              | 23    | 23    | 24     | 25     | 28     | 28       |              |
| 28   | 28 | 28    | 28    | 28      | 28             | 28            | 28              | 28    | 28    | 28     | 28     | 28     | 28       | 28           |

Program: SPACE. (# of faults located)

A problem of nonparametric tests in relation to parametric tests is that the former tests are not powerful in the sense that the produced  $p$ -values tend to be quite large, and hence it is more difficult to tell that the difference is statistically significant. This adverse effect is linked to the ranking part of statistical fault localization techniques, which may use the  $p$ -values as the indicators to prioritize program entities and presented them to their users. On the other hand, if a  $p$ -value which is not small enough and the corresponding feature instance is still presented in a ranked list as a suggestion to developers, the list would contain entries that associate diverse degree of confidence values. An entry will a low confidence but is ranked high may mislead the developers. Similarly, in the aforementioned similarity coefficient measurement sections, if the fault-failure correlation of a feature instance is only mild (e.g., 0.5), but a technique still present the feature instance to the developers simply because the feature instance is ranked high in the ranking procedure, the developer may also be misled.

We have only scratched the surface of using a non-parametric idea for fault localization for program debugging. As we have mentioned in Section 2, sometimes, a feature instance may come with one failing sample in  $D$ . In such a case, even a non-parametric approach cannot be reliably applied in a standalone manner. A further investigation on scientific methods to address this problem is needed to be well done.

As a final note, there are proposals (e.g., [7]) to use machine learning approaches that train classifiers to locate and categorize faults based on program spectra in a dataset. A simple and direct application to existing learning algorithms (Bayesian or not) ignores to validate whether the dataset can provide a diagnosis result with sufficient *confidence* associated with their classification strategies. Maximizing the diagnosis power over a dataset for statistical fault localization does not solve this fundamental issue either.

## 6. Concluding Remarks

In this experience report, we have summarized and re-interpreted some of the isolated work that we have investigated in the past few years. The main message is two-fold. First, we have argued that a fault localization strategy should model a faulty program statistically as an approach to understanding faults, fault activation, error propagation, and failure observation in a chained manner and in a statistical way. Second, during a fault localization process, a fault localization strategy should address the errors in multiple dimensions such as data, similarity coefficient, and statistical inference.

We have also reviewed the basic ideas of some selected strategies and elaborated their rationales. We have described selective mistakes that the science part of a technique could be improved in the last two sections. We have made such mistakes ourselves in our own studies. We hope that through this occasion, we could share our first-hand experiences with readers, and raise the awareness of the research community not to overlook the importance of developing sound techniques.

There are some obvious challenges for statistical fault localization in general. For instance, a program with about 10,000 lines of code is usually regarded as small. However, it is not quite effective if a developer still needs to debug the program by focusing their attention to the first 100 lines of code suggested by a technique. At the same time, we have presented some empirical results of different fault localization techniques in this report. Readers must have observed that these techniques cannot consistently locate faults within 1% code examination effort or a small number of lines of codes, say 3 lines, that the developers may feel comfortable to use such a technique in practice. Both accuracy and scalability of this class of fault localization techniques are remained to be improved significantly.

Another issue is about a fundamental assumption about a statistical approach. A developer may require debugging a program when there is only one failing case. In such circumstances, some feature instances may not accumulate sufficient samples in the dataset to conduct reliable statistical inference even one chooses a nonparametric approach. It appears that some mechanisms to control the involved reliability should be developed. Moreover, some faults may be elusive and may not be related to whether a particular executable statement has been implemented wrongly. For instance, there are parameter configuration errors in software applications or errors in the compiler to translate mistakenly the source code into a wrong object code. It is also unclear to us how to relate the precise position of omission faults to a similarity coefficient, although we have proposed to use fault patterns.

Fortunately, the challenge in fault localization is not restrictive to software engineering. We may learn a lot of from the fault localization studies conducted in the other fields such as artificial intelligence, data mining, and networks and systems.

## 7. References

- [1] Abreu, R., Zoetewij, P., and van Gemund, A. J. C. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.
- [2] Agrawal, H., and Horgan, J.R. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI 1990)*, pages 246–256, 1990.
- [3] Agrawal, H., Horgan, J., Lodon, S., and Wong, W. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering (ISSRE 1995)*, pages 143–151, 1995.

- [4] Artho, C. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer*, 13(3):223–246, 2011.
- [5] Baah, G. K., Podgurski, A., and Harrold, M. J. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA '08)*, pages 189–200, 2008.
- [6] Baah, G. K., Podgurski, A., and Harrold, M. J. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis (ISSTA 2010)*, pages 73–84, 2010.
- [7] Briand, L.C., Labiche, Y., and Liu, X. Using machine learning to support debugging with tarantula. In *Proceedings of the 18<sup>th</sup> International Symposium on Software Reliability (ISSRE 2007)*, pages 137–146, 2007.
- [8] Casella, G., and Berger, R. *Statistical Inference*, the Second Edition, Duxbury, 2001.
- [9] Chan, W.K. Statistical fault localization: challenges and achievement. Keynote address. *The Second International Workshop on Program Debugging (IWPD 2011) in conjunction with the 35<sup>th</sup> Annual IEEE Conference on Computer Software and Applications (COMPSAC 2011)*, Munich, 18–21 July 2011.
- [10] Ceillier, P., Ducasse, S., Ferre, S., and Ridoux, O. Formal concept analysis enhances fault localization in software. In *Proceedings of the 4<sup>th</sup> International Conference on Formal Concept Analysis (ICFCA 2008)*, pages 273–288, LNCS 4933, Springer, 2008.
- [11] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A.V., and Vaswani, K. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 34–44, 2009.
- [12] Debroy, V., and Wong, W.E. On the consensus-based application of fault localization techniques. To appear in *Proceedings of the 2<sup>nd</sup> International Workshop on Program Debugging (IWPD 2011) in conjunction with the 34<sup>th</sup> Annual IEEE Computer Software and Applications Conference (COMPSAC 2011)*, volume 2, 2011.
- [13] Denmat, T., Ducassé, M., and Ridoux, O. Data mining and cross-checking of execution traces: a re-interpretation of Jones, Harrold and Stasko test information visualization. In *Proceedings of the 20<sup>th</sup> IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 396–399, 2005.
- [14] Do, H., Elbaum, S.G., and Rothermel, G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.
- [15] Eagan J., Harrold, M.J., Jones, J.A., and Stasko, J. Technical Note: Visually Encoding Program Test Information to Find Faults in Software. In *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS 2001)*, pages 33–36, 2011.
- [16] Harrold, M.J., Rothermel, G., Wu, R., and Yi, L. An empirical investigation of program spectra. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE 1998)*, pages 83–90, 1998.
- [17] Han, J., and Kamber, M. *Data Mining: Concepts and Techniques, the 2<sup>nd</sup> Edition*. The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor Morgan Kaufmann Publishers, March 2006. ISBN 1-55860-901-6.
- [18] Hovermeyer, D., and Pugh, W. Finding bugs is easy. In *Companion to the 19th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 132–136, 2004.
- [19] Hovermeyer, D., Spacco, J., and Pugh, W. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of the 6<sup>th</sup> ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, pages 13–19, 2005.
- [20] Hu, P., Zhang, Z., Chan, W.K., and Tse, T.H. Fault localization with non-parametric program behavior model. In *Proceedings of the 8<sup>th</sup> International Conference on Quality Software (QSIC 2008)*, pages 385–395, 2008.
- [21] Jaccard, P. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Socit Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

- [22] Jackson AA, Somers KM, Harvey HH. Similarity coefficients: measures for co-occurrence and association or simply measures of occurrence?. *American Naturalist*, 133:436–453, 1989.
- [23] Jarque, C.M., Bera, A.K. Efficient tests for normality, homoscedasticity and serial independence of regression residuals: Monte Carlo evidence. *Economics Letters*, 7(4):313–318, 1981.
- [24] Jeffrey, D., Gupta, N., and Gupta R. Fault localization using value replacement. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 167–178, 2008.
- [25] Jones, J. A., and Harrold, M.J., and Stasko, J. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 467–477, 2002.
- [26] Jones, J. A., and Harrold, M.J., Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE 2005)*, pages 273–282, 2005.
- [27] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 15–26, 2005.
- [28] Liu, C., Fei, L., Yan, X., Midkiff, S. P., Han, J. Statistical debugging: a hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10), 831–848, 2006.
- [29] Masri, W. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability*, 20:121–147, 2010.
- [30] Naish, L, Lee, H. JI, and Ramamohanarao, K. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 20(3): Article 11 (August 2011), 32 pages, 2011. <http://doi.acm.org/10.1145/2000791.2000795>
- [31] National Institute of Standards & Technology. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3, May 2002.
- [32] Renieris M., and Reiss, S.P. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 30–39, 2003.
- [33] Reps, T., Ball, T., Das, M., and Larus, J The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC 1997/FSE-5)*, pages 432–449, 1997.
- [34] Richardson, D. J., and Thompson, M.C. An analysis of test data selection criteria using the REPLAY model of fault detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, 1993.
- [35] Siegel, S., and Catellan, N.J. *Nonparametric Statistics for the Behavioral Sciences*, 2<sup>nd</sup> Edition. New York, NY. McGraw-Hill, 1988.
- [36] Santelices, R., Jones, J.A., Yu, Y., Harrold, M.J. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31<sup>st</sup> International Conference on Software Engineering (ICSE 2009)*, pages 55–66, 2009.
- [37] Sumner, W.N., Bao, T., Zhang, X. Selecting peers for execution comparison. *Proceedings of the 20th international symposium on Software testing and analysis (ISSTA 2011)*, pages 309–319, 2011.
- [38] Vaziri, M., Tip, F., and Dolby, J. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of 33rd Annual Symposium on Principles of Programming Language (POPL 2006)*, pages 334–345, 2006.
- [39] Vessey, I. Expertise in debugging computer programs: an analysis of the content of verbal protocols. *IEEE Transactions on Systems, an, and Cybernetics*, 16(5):621–637, 1986.
- [40] Wang, X., Cheung, S. C., Chan, W. K., and Zhang, Z. Taming coincidental correctness: coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 45–55, 2009.
- [41] Wilkinson, J.H. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1988.
- [42] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, 10 (4): 352–357, 1984.

- [43] Wong E., Wei, T., Qi, Y., and Zhao, L. A Crosstab-based Statistical Method for Effective Fault Localization. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 42–51, 2008.
- [44] Wotawa, F., Stumptner, M., and Mayer, W. Model-based debugging or how to diagnose programs automatically. In *Proceedings of the 15<sup>th</sup> International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2002)*, Developments in Applied Artificial Intelligence, pages 243-257, LNCS 2358/2002, Springer, 2002.
- [45] Xu, J., Z. Zhang, W.K. Chan, T.H. Tse, and S. Li. A Dynamic Fault Localization Technique with Noise Reduction for Java Programs. In *Proceedings of the 11<sup>th</sup> International Conference on Quality Software (QSIC 2011)*, pages 11–20, 2011.
- [46] Yates, F. Contingency tables involving small numbers and the  $\chi^2$  test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217–235, 1934.
- [47] Yu, Y., Jones, J.A., and Harrold, M.J.. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 201–210, 2008.
- [48] A. Zeller. Yesterday, my program worked. Today, it does not. Why?. In *Proceedings of ESEC/FSE-7 Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*, pages 253–267, 1999.
- [49] Zhao, Q., Sim, J.E., Wong, W.F., and Rudolph, L. DEP: detailed execution profile. In *Proceedings of the 15<sup>th</sup> International Conference on Parallel Architectures and Compilation Techniques (PACT 2006)*, pages 154–163, 2006.
- [50] Zhang, X., Gupta, N., and Gupta, R. Pruning dynamic slices with confidences. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 169–180, 2008.
- [51] Zhang, X., Gupta, N., and Gupta, R. Locating faults through automated predicate switching. In *Proceedings of the 28<sup>th</sup> International Conference on Software Engineering (ICSE 2006)*, pages 271–281, 2006.
- [52] Zhang, Z., Chan, W.K., Tse, T.H., Hu P., and Wang, X. Is non-parametric hypothesis testing model robust for statistical fault localization?, *Information and Software Technology*, 51 (11):1573–1585, 2009.
- [53] Zhang, Z., Chan, W.K., Tse, T.H., Jiang, B., and Wang, X. Capturing propagation of infected program states, *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC 2009/FSE-17)*, pages 43–52, 2009.
- [54] Zhang, Z., Chan, W.K., Tse, T.H., and Jiang, B. Precise Propagation of Fault-Failure Correlations in Program Flow Graphs. To appear in *Proceedings of the 35<sup>th</sup> Annual International Computer Software and Applications Conference (COMPSAC 2011)*, 10 pages, 2011.
- [55] Zhang, Z., Chan, W.K., Tse, T.H., Yu, Y.T., and Hu, P. Non-parametric statistical fault localization. *Journal of Systems and Software*, 84(6):885–905, 2011.
- [56] Zhang, Z., Jiang, B., Chan, W.K., Tse, T.H., and Wang, X. Fault localization through evaluation sequences. *Journal of Systems and Software*, 83 (2): 174–187, 2010.
- [57] Zolman, J.F. Experimental design and statistical inference. Oxford University Press, New York, 1993.