


# Probabilistic Fault Localisation

David Landsberg<sup>1</sup>, Hana Chockler<sup>2</sup>, and Daniel Kroening<sup>1</sup>

<sup>1</sup> Department of Computer Science, University of Oxford, Oxford, UK  
`david.landsberg@linacre.ox.ac.uk`

<sup>2</sup> Department of Informatics, King's College London, London, UK

**Abstract.** Efficient fault localisation is becoming increasingly important as software grows in size and complexity. In this paper we present a new formal framework, denoted *probabilistic fault localisation* (PFL), and compare it to the established framework of spectrum based fault localisation (SBFL). We formally prove that PFL satisfies some desirable properties which SBFL does not, empirically demonstrate that PFL is significantly more effective at finding faults than all known SBFL measures in large scale experimentation, and show PFL has comparable efficiency. Results show that the user investigates 37% more code (and finds a fault immediately in 27% fewer cases) when using the best performing SBFL measures, compared to the PFL framework. Furthermore, we show that it is theoretically impossible to design strictly rational SBFL measures that outperform PFL techniques on a large set of benchmarks.

**Keywords:** Fault localisation · Spectrum based fault localisation · Triage and debug technologies

## 1 Introduction

Faulty software is estimated to cost 60 billion dollars to the US economy per year [1] and has been single-handedly responsible for major newsworthy catastrophes<sup>1</sup>. This problem is exacerbated by the fact that debugging (defined as the process of finding and fixing a fault) is complex and time consuming – estimated to consume 50–60% of the time a programmer spends in the maintenance and development cycle [2]. Consequently, the development of effective and efficient methods for software fault localisation has the potential to greatly reduce costs, wasted programmer time and the possibility of catastrophe.

In this paper we advance the state of the art in lightweight fault localisation by building on research in spectrum-based fault localisation (SBFL). In SBFL, a *measure* is used to determine the degree of suspiciousness each line of code is with respect to being faulty, where this degree is defined as a function of the number of passing/failing traces that do/do not cover that code. SBFL is one of the most prominent areas of software fault localisation research, has recently

---

<sup>1</sup> <https://www.newscientist.com/gallery/software-faults/>.

been estimated to make up 35% of published work in the field [3] and has been consistently demonstrated to be effective and efficient at finding faults [4–21].

However, so far there have not been many formal properties about the general problem of fault localisation which SBFL measures have been shown to satisfy, representing a potential theoretical shortcoming of the approach. Although properties that measures should satisfy a priori (such as strict rationality [22]) have been discussed, and measures that solve fault localisation sub-problems have been presented (such as single bug optimal measures [15]), there is not yet a SBFL measure that solves the problem of fault localisation for all benchmarks. Indeed, recently Yoo et al. have established theoretical results which show that a “best” performing suspicious measure does not exist [23]. In light of this, the SBFL literature has favoured developing measures with good experimental performance as opposed to developing them according to a priori requirements. This has facilitated a culture of borrowing measures from other domains [11, 15, 16], manually tweaking measures [13, 17], or using machine learning methods [19, 20, 24]. Thus, there remains the challenge of developing new, better performing and comparably efficient methods that can satisfy key properties of fault localisation. Our contributions in this paper are as follows:

- We introduce and motivate a new formal framework denoted *Probabilistic Fault Localisation* (PFL), which can leverage any SBFL measure.
- We formally prove that PFL satisfies desirable formal properties which SBFL does not.
- We demonstrate that PFL techniques are substantially and statistically significantly more effective (using  $p = 0.01$ ) than all known (200) SBFL measures at finding faults on what, to our knowledge, is the largest scale experimental comparison in software fault localisation to date.
- We show that it is theoretically impossible to define strictly rational SBFL measures that can outperform given PFL techniques on many of our benchmarks.
- We demonstrate that PFL maintains efficiency comparable to SBFL.

The rest of the paper is organised as follows. In Sect. 2 we present the formal preliminaries common to the approaches discussed in this paper and in Sect. 3 introduce a small illustrative example of SBFL. In Sect. 4, we introduce and motivate the formal theory underlying the PFL approach and formally prove it satisfies desirable fault localisation properties which SBFL does not. Section 5 presents our experimental comparison of PFL techniques against SBFL measures. Finally, we present related work and general conclusions.

## 2 Preliminaries

In this section we summarise the formal apparatus common to the approaches in this paper.

We model each *program* as an ordered set  $\mathbf{P} = \langle C_1, \dots, C_n \rangle$ . Intuitively, each  $C_i$  can be thought of as a program entity, event, or proposition, which is executed, occurs, or is true if a corresponding program entity is covered in a given

execution. A program entity (or component) can be thought of as a program statement, branch, path, or block of code [7, 25]. A component is called *complex* if it is the union of other components, and *atomic* otherwise. In practice,  $\mathbf{P}$  is modelled as a set of atomic components in order to reduce overhead [4–21].

We model each *test suite* as an ordered set of test cases  $\mathbf{T} = \langle t_1, \dots, t_m \rangle$ . Each test case  $t_i$  is a Boolean vector of length  $|\mathbf{P}|$  such that  $t_k = \langle c_1^k, \dots, c_{|\mathbf{P}|}^k \rangle$ , and where we have  $c_i^k = \top$  if  $C_i$  is covered/occurs/is true in  $t_k$  and  $\perp$  otherwise. We also use 1 and 0 for  $\top$  and  $\perp$  respectively.  $C_{|\mathbf{P}|}$  can be thought of as the event of the error (denoted by  $E$ ), where  $c_{|\mathbf{P}|}^k = e^k = \top$  if the test case *fails* and  $\perp$  if it *passes*. Intuitively, each test case records the coverage details of a given execution, and is failing/passing if that execution violates/satisfies a given specification, where a specification is a logically contingent proposition stated in some formal language.

Each test suite may be partitioned  $\mathbf{T} = \mathbf{F} \cup \mathbf{P}$ , where  $\mathbf{F}$  and  $\mathbf{P}$  are the set of failing and passing test cases respectively. By convention each test suite is ordered such that the failing traces appear before the passing. In general, we assume that every failing test case covers at least one component, and that every component is covered by at least one failing test case. We may represent a test suite with an  $m \times n$  *coverage matrix*, in which the  $k$ -th row of the  $i$ -th column represents whether  $C_i$  occurred in  $t_k$ . An example of a coverage matrix is given in Fig. 2.

For each  $C_i \in \mathbf{P}$  we can construct its *program spectrum* using a test suite. A program spectrum is defined as a vector of four elements  $\langle a_{ef}^i, a_{nf}^i, a_{ep}^i, a_{np}^i \rangle$ , where  $a_{ef}^i$  is the number of failing test cases in  $\mathbf{T}$  that cover  $C_i$ ,  $a_{nf}^i$  is the number of failing test cases in  $\mathbf{T}$  that do not cover  $C_i$ ,  $a_{ep}^i$  is the number of passing test cases in  $\mathbf{T}$  that cover  $C_i$  and  $a_{np}^i$  is the number of passing test cases in  $\mathbf{T}$  that do not cover  $C_i$ . Probabilistic expressions may be defined as a function of program spectra as follows. We identify  $P(C_i \cap E)$ ,  $P(\overline{C_i} \cap E)$ ,  $P(C_i \cap \overline{E})$  and  $P(\overline{C_i} \cap \overline{E})$  with  $\frac{a_{ef}^i}{|\mathbf{T}|}$ ,  $\frac{a_{nf}^i}{|\mathbf{T}|}$ ,  $\frac{a_{ep}^i}{|\mathbf{T}|}$  and  $\frac{a_{np}^i}{|\mathbf{T}|}$  respectively. Using definitions from probabilistic calculus [26], we may then identify many measures with a probabilistic expression.

A *suspiciousness measure*  $w$  maps a program entity to a real number as a function of its spectrum [15], where this number is called the program entity's degree of suspiciousness. The higher the degree the more suspicious the program entity  $C_i$  is assumed to be with respect to being a fault. In practical SBFL the components in the program are investigated in descending order of suspiciousness until a fault is found. Prominent measures include Zoltar =  $\frac{a_{ef}^i}{a_{ef}^i + a_{nf}^i + a_{ep}^i + k}$  where  $k = \frac{10000 a_{nf}^i a_{ep}^i}{a_{ef}^i}$  [15], Kulczynski2 =  $\frac{1}{2}(P(E|C_i) + P(C_i|E))$  [15], Ochiai =  $P(C_i \cap E) / \sqrt{P(\overline{C_i})P(E)}$  [4], and Positive predictive power (PPV) =  $P(E|C_i)$  [6]. PPV is equivalent to the Tarantula measure [27].

Some suspiciousness measures are informally known as measures of *causal strength* [11]. Measures of causal strength are designed to measure the propensity of an event in causing a given effect. Any measure can be proposed as

a measure of causal strength. Historically such measures have been developed around the premise that causes raise the probability of their effects. Prominent measures include Lewis =  $P(E|C_i)/P(E|\neg C_i)$ , Fitelson =  $P(E|C_i)/P(E)$ , Suppes =  $P(E|C_i) - P(E|\neg C_i)$ , and Eels =  $P(E|C_i)/P(E)$  (see [11]).

Formally, a suspiciousness measure  $w$  is *rational* if and only if for all  $c > 0$  (1)  $w(a_{ef}, a_{nf}, a_{ep}, a_{np}) \leq w(a_{ef} + c, a_{nf} - c, a_{ep}, a_{np})$ , and (2)  $w(a_{ef}, a_{nf}, a_{ep} + c, a_{np} - c) \leq w(a_{ef}, a_{nf}, a_{ep}, a_{np})$ . The property of *strict rationality* is defined by replacing  $\leq$  with  $<$  in the latter two conditions. Roughly speaking, a measure is rational/strictly-rational if more failing traces covering a component make it more suspicious, and more passing traces make it less suspicious – conforming to our intuition of suspiciousness. Many suspiciousness measures have been shown to satisfy strict rationality, at least when  $a_{ef}, a_{ep} > 1$  [15, 22]. Naish et al. argue that it is reasonable to restrict the SBFL approach to rational measures [28].

We now discuss established methods for evaluating the performance of a suspiciousness measure. First, there are *wasted effort* scores (or W-scores). W-scores estimate the percentage of non-faulty components a user will look through until a fault is found. Best case, worst case, and average case W-scores have been defined [4, 11, 13, 15]. Where  $w$  is a measure,  $b$  is a fault with the highest degree of suspiciousness, and  $f$  is the number of faults which are equally suspicious to the most suspicious fault, we use the following definitions:  $best(w) = \frac{|\{x|w(x) > w(b)\}|}{|P|-1} 100$ ,  $worst(w) = \frac{|\{x|m(x) \geq w(b)\}-1|}{|P|-1} 100$ ,  $average(w) = best(w) + \frac{worst(w) - best(w)}{f+1}$ . We use *avg* W-scores. Second, there are *absolute scores* (or A-scores) [29]. A-scores measure whether a given measure found a fault after inspecting  $n$  non-faulty components [29]. Thus, for a given  $n$  a suspiciousness measure receives 100% if the user found a fault after investigating  $n$  non-faulty components, otherwise it received 0%. We use  $n = 0$ . A suspiciousness measure performs well if it has low mean W-scores and a high mean A-scores.

Finally, Naish et al. define the *unavoidable costs* of any strictly rational measure. These are the scores that the best performing strictly rational measure can possibly receive [28]. To determine this score, one constructs an ordered list with the property that for every component,  $C_i$  is ranked higher than a given fault  $C_j$  just in case every strictly rational measure would rank  $C_i$  higher than  $C_j$ . The W/A-scores of this list are the unavoidable cost W/A-scores. Unavoidable cost scores estimate the upper bound limit for the performance of the SBFL approach in general (see [28] for details).

### 3 Example

We present a small example to illustrate SBFL. Consider the C program **min-max.c** in Fig. 1 (from [30]). The program is formally modelled as the following set of program entities  $P = \langle C_1, C_2, C_3, C_4, E \rangle$ , where  $E$  models the event in which the specification `assert(least <= most)` is violated. The program fails to always satisfy this specification. The explanation for the failure is the fault at  $C_3$ , which should be an assignment to `least` instead of `most`. We collected coverage data from ten test cases to form our test suite  $T = \langle t_1, \dots, t_{10} \rangle$ . The

```

int main() {
    int input1, input2, input3;
    int least = input1;
    int most = input1;

    if (most < input2)
        most = input2; // C1
    if (most < input3)
        most = input3; // C2
    if (least > input2)
        most = input2; // C3 (fault)
    if (least > input3)
        least = input3; // C4

    assert(least <= most); // E
}
    
```

Fig. 1. minmax.c

	$C_1$	$C_2$	$C_3$	$C_4$	$E$
$t_1$	0	1	1	0	1
$t_2$	0	0	1	1	1
$t_3$	0	0	1	0	1
$t_4$	1	0	0	0	0
$t_5$	0	1	0	0	0
$t_6$	0	0	0	1	0
$t_7$	0	0	1	1	0
$t_8$	0	0	0	0	0
$t_9$	1	0	0	1	0
$t_{10}$	1	1	0	0	0

Fig. 2. Coverage matrix

coverage matrix for these test cases is given in Fig. 2. Three of the test cases fail and seven pass. We compute the program spectrum for each component using the coverage matrix. For example, the program spectrum for  $C_3$  is  $\langle 3, 0, 1, 6 \rangle$ .

To illustrate an instance of SBFL we use the suspiciousness measure Wong-2 =  $a_{ef}^i - a_{ep}^i$  [13]. The user inspects the program in decreasing order of suspiciousness until a fault is found.  $C_3$  is inspected first with a suspiciousness of 2 and thereby a fault is found immediately. The example illustrates that SBFL measures can be successfully employed as heuristics for fault localisation, but that the formal connection to fault localisation could potentially be improved.

## 4 Estimating Fault Probability

In this section, we introduce assumptions to generate our estimation of fault probability and then prove this estimation satisfies important properties that are not satisfied by any SBFL measure.

We begin as follows. We introduce a probability function  $P$  the domain of which is a set of propositions. To define the set of propositions, we first define two sets of atomic propositions  $\mathbf{H} = \{h_i | C_i \in \mathbf{P}\}$  and  $\mathbf{C} = \{h_i^k | C_i \in \mathbf{P} \wedge t_k \in \mathbf{T}\}$ . Intuitively,  $\mathbf{H}$  is a set of fault hypotheses, where  $h_i$  expresses the hypothesis that  $C_i$  is faulty, and  $\mathbf{C}$  is a set of causal hypotheses, where  $h_i^k$  expresses the hypothesis that  $C_i$  was the cause of the error  $E$  in execution  $t_k$ . The set of propositions is then defined inductively as follows. For each  $p, q \in \mathbf{H} \cup \mathbf{C}$ ,  $p$  and  $q$  are propositions. If  $p$  and  $q$  are propositions, then  $p \wedge q$ ,  $p \vee q$ ,  $\neg p$  are propositions. We also assume the following standard properties of probability [26]. For each proposition  $p$  and  $q$ :  $P(p) = 1$  if  $p = \top$ .  $P(p) = 0$  if  $p = \perp$ .  $P(p \vee q) = P(p) + P(q) - P(p \wedge q)$ .  $P(\neg p) = 1 - P(p)$ .  $P(p|q) = P(p \wedge q)/P(q)$ .

We now present assumptions **A1-7** which are designed to be plausible in any probability space induced by a test suite  $\mathbf{T}$  for a faulty program  $\mathbf{P}$ .

**A1.** For all  $h_i \in \mathbf{H}$ ,  $h_i = \bigvee_{k=1}^{|\mathbf{T}|} h_i^k$ .

This states that  $C_i$  is faulty just in case  $C_i$  was the cause of the error  $E$  in some execution of the program.

**A2.** For all  $t_k \in \mathbf{F}$ ,  $\bigvee_{i=1}^{|\mathbf{P}|} h_i^k = \top$ .

This states that for every failing trace, there is some component  $C_i \in \mathbf{P}$  which caused the error  $E$  in that trace. In other words, if an error occurred then something must have caused it. For all  $h_i^k \in \mathbf{C}$  we also have the following

**A3.** if  $h_i^k = \top$  then  $C_i \neq E$ .

**A4.** if  $h_i^k = \top$  then  $c_i^k = \top$  and  $e^k = \top$ .

These assumptions state that if  $C_i$  was the cause of  $E$  in  $t_k$ , then  $C_i$  must have been a different event to  $E$  (**A3**), and  $C_i$  and  $E$  must have actually occurred (**A4**). These two assumptions have been described as fundamental properties about causation [31]. For all  $h_i^k, h_j^k \in \mathbf{C}$

**A5.** if  $C_i \neq C_j$  then  $h_i^k \wedge h_j^k = \perp$ .

This states that no two events could have both been the cause of the error in a given trace. In other words, different causal hypotheses for the same trace are mutually exclusive. The rationale for this is that the intended meaning of  $h_i^k$  is  $C_i$  was *the* cause of  $E$  in  $t_k$ , and as *the* implies uniqueness, no two events could have been *the* cause. In general, any union of events may be said to be *the* cause so long as that union is in  $\mathbf{P}$ . For all  $h_i^k \in \mathbf{C}$  and every sample  $\mathbf{S} \subseteq \mathbf{T} - \{t_k\}$

**A6.**  $P(h_i^k | \bigvee_{t_n \in \mathbf{S}} h_i^n) = P(h_i^k)$ .

This states that the probability that  $C_i$  was the cause in one trace is not affected by whether it was in some others. In other words, whether it was the cause in one is statistically independent of whether it was in others. Here, we assume that our probabilities describe objective chance, and that the causal properties of each execution is determined by the properties of the events in that execution alone, and therefore cannot affect the causal properties of other executions. Independence principles are well established in probability theory [26].

In light of the above assumptions we may define  $c(t_k) = \{C_i | C_i \in \mathbf{P} \wedge c_i^k = e^k = \top \wedge C_i \neq E\}$  as the set of candidate causes of  $E$  in  $t_k$ . Following this, for some measure  $w$ , and all  $C_i, C_j \in c(t_k)$ , we assume

**A7.**  $P(h_i^k)/P(h_j^k) = w(C_i)/w(C_j)$ .

Here, we assume  $w$  measures the propensity of a given event to cause the error (and is thus motivated as a measure of causal strength as described in the preliminaries). Accordingly, the assumption states that the relative likelihood that one event caused the error over another, is directly proportional to their propensities to do so. In general, any suspiciousness measure  $w$  from the SBFL literature may be proposed as a measure of causal strength, and thus there is great room for experimentation over the definition of  $w$ . One formal proviso is that measures be re-scaled so  $w(C_i) > 0$  if  $a_{e,f}^i > 0$  (this avoids divisions by zero). We use the notation PFL- $w$  when measure  $w$  is being used.

We now show that the assumptions **A1-7** (henceforth PFL assumptions) imply Eqs. (1), (2) and (3) (henceforth PFL equations). The PFL equations can be used to determine the probability that a given component  $C_i$  is faulty. For all  $h_i^k \in \mathbf{C}$

$$P(h_i) = P\left(\bigvee_{n=1}^{|\mathbf{T}|} h_i^n\right) \quad (1)$$

$$P\left(\bigvee_{j=n}^{|\mathbf{T}|} h_i^j\right) = P(h_i^n) + P\left(\bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j\right) - P(h_i^n)P\left(\bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j\right) \quad (2)$$

$$P(h_i^k) = \begin{cases} \frac{w(C_i)}{\sum_{C_j \in c(t_k)} w(C_j)} & \text{if } C_i \in c(t_k) \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

**Proposition 1.** *The PFL assumptions imply the PFL equations.*

*Proof.* We first show Eq. (1).  $h_i = \bigvee_{k=1}^{|\mathbf{T}|} h_i^k$  (by **A1**). Thus  $P(h_i) = P\left(\bigvee_{k=1}^{|\mathbf{T}|} h_i^k\right)$  (by Leibniz's law). We now show Eq. (2). The definition of disjunction states  $P\left(\bigvee_{j=n}^{|\mathbf{T}|} h_i^j\right) = P(h_i^n) + P\left(\bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j\right) - P(h_i^n \wedge \bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j)$ . It remains to show  $P(h_i^n \wedge \bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j) = P(h_i^n)P\left(\bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j\right)$ .  $P(h_i^n \wedge \bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j)$  is equal to  $P(h_i^n | \bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j)P\left(\bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j\right)$  (by probabilistic calculus). This is equal to  $P(h_i^n)P\left(\bigvee_{j=n+1}^{|\mathbf{T}|} h_i^j\right)$  (by **A6**).

We now show Eq. (3). We have two cases to consider:  $C_i \in c(t_k)$  and  $C_i \notin c(t_k)$ . Assume  $C_i \in c(t_k)$ . We may assume  $t_k$  is ordered such that  $\bigwedge_{i=1}^n c_i^k = \top$ ,  $\bigwedge_{i=n+1}^{|P|-1} c_i^k = \perp$  and  $c_{|P|}^k = e^k = \top$  (such that  $c(t_k) = \{C_1, \dots, C_n\}$ ). Now, for all  $C_i, C_j \in c(t_k)$   $P(h_i^k)/P(h_j^k) = w(C_i)/w(C_j)$  (by **A7**). Thus, for  $C_i, C_j \in c(t_k)$   $w(C_i)/P(h_i^k) = w(C_j)/P(h_j^k)$  (as  $x/y = w/z \equiv z/y = w/x$ ). So,  $w(C_1)/P(h_1^k) = w(C_2)/P(h_2^k) = \dots = w(C_n)/P(h_n^k)$ . Thus, there is some  $c$  such that for all  $C_i \in c(t_k)$ ,  $c = w(C_i)/P(h_i^k)$  (by the last result). Equivalently, there is some  $c$  such that for all  $C_i \in c(t_k)$ ,  $P(h_i^k) = w(C_i)/c$ . To complete the proof it remains to prove  $c = \sum_{C_j \in c(t_k)} w(C_j) \cdot \bigvee_{i=1}^{|P|} h_i^k = \top$  (by **A2**). But,  $\bigvee_{i=n+1}^{|P|-1} h_i^k = \perp$  (by **A4**), and  $h_{|P|}^k = \perp$  (by **A3**). Thus,  $\bigvee_{i=1}^n h_i^k = \top$  (by  $\vee$ -elimination). So,  $P\left(\bigvee_{i=1}^n h_i^k\right) = 1$  (by probabilistic calculus). Thus,  $\sum_{i=1}^n P(h_i^k) = 1$  (by probabilistic calculus

and **A5**). So,  $\sum_{i=1}^n (w(C_i)/c) = 1$ . Thus,  $(\sum_{i=1}^n w(C_i))/c = 1$ . Equivalently,  $\sum_{i=1}^n w(C_i) = c$ . So,  $\sum_{C_i \in c(t_k)} w(C_i) = c$  (by def. of  $c(t_k)$  above). We now do the second condition. Assume  $C_i \notin c(t_k)$ . Then  $\neg(c_i^k = e^k = \top \wedge C_i \neq E)$  (by def. of  $c(t_k)$ ). Thus  $c_i^k = \perp$  or  $e^k = \perp$  or  $C_i = E$ . If  $C_i \neq E$ , then  $P(h_i^k) = 0$  (by **A3**). If  $c_i^k = \perp$ , then  $P(h_i^k) = 0$  (by **A4**). If  $e^k = \perp$ , then  $P(h_i^k) = 0$  (by **A4**). Thus, if  $C_i \notin c(t_k)$ , then  $P(h_i^k) = 0$ .

To use the PFL equations, it remains for the user to choose a measure  $w$  for **A7**. One proposal is  $w(C_i) = P(E|C_i)$  (the PPV measure [11]) or  $P(E|C_i)/P(E)$  (the Fitelson measure of causal strength [32]). For the purposes of defining  $P(h_i^k)$  both proposals are equivalent (observe  $P(h_i^k)/P(h_j^k) = P(E/C_i)/P(E/C_i) = (P(E/C_i)/P(E))/(P(E/C_j)/P(E))$  using **A7**).

The proposal captures three potentially plausible intuitions about causal likelihood. Firstly, it captures an intuition that the more something raises the probability of the error, the more likely it is to be the cause of it (to see this, observe we have  $P(h_i^k)/P(h_j^k) = P(E|C_i)/P(E|C_j)$  using **A7**). Secondly, it captures an intuition that events which do not affect the error's likelihood are equally unlikely to have caused it (to see this, assume both  $C_i$  and  $C_j$  are independent of  $E$  i.e.  $P(E) = P(E|C_i)$  and  $P(E) = P(E|C_j)$ , then it follows  $P(h_i^k) = P(h_j^k)$  using **A7**). Thirdly, a plausible estimate of  $w(C_i)$  as a measure of  $C_i$ 's causal strength is the probability that  $C_i$  causes  $E$  given  $C_i$ , and  $P(E|C_i)$  accordingly provides an upper bound for this estimate. In our running example PFL-PPV returns  $P(h_2) = 0.00$ ,  $P(h_2) = 0.31$ ,  $P(h_3) = 1.00$ , and  $P(h_4) = 0.25$ , which correctly identifies the correct hypothesis with the most probable one.

Finally, given a test suite  $\mathbf{T}$  and measure  $w$ , an algorithm to find a single fault in a program  $\mathbf{P}$  is as follows. Step one, find  $\max_{h_i \in \mathbf{H}} (P(h_i))$  by computing the value of  $P(h_i)$  for each  $h_i \in \mathbf{H}$  using the PFL equations. If the most probable hypothesis represents a fault in the program, the procedure stops. Otherwise,  $h_j$  is removed from the set of candidates by setting  $c_j^k = \perp$  for each  $t_k$ , and return to step one. We call this the PFL algorithm. A property of this algorithm is that yet to be investigated components can change in fault likelihood at each iteration.

We now identify desirable formal properties which we prove the PFL equations satisfies, but no SBFL suspiciousness measure can.

**Definition 1. Fault Likelihood Properties.** For all  $C_i, C_j \in \mathbf{P}$ , where  $C_i \neq C_j$ , we define the following:

1. *Base case.* If there is some failing trace which only covers  $C_i$ , but this property does not hold of  $C_j$ , then  $C_i$  is more suspicious than  $C_j$ .
2. *Extended case.* Let  $\mathbf{T}_1$  be a test suite in which all failing traces cover more than one component, and let  $\mathbf{T}_2$  be identical to  $\mathbf{T}_1$  except  $c_i^k = 1$  and  $c_j^k = 1$  in  $\mathbf{T}_1$  and  $c_i^k = 1$  and  $c_j^k = 0$  in  $\mathbf{T}_2$ , then the suspiciousness of  $C_i$  in  $\mathbf{T}_2$  is more than its suspiciousness in  $\mathbf{T}_1$ .

These properties capture the intuition that the fewer covered entities there are in a failing trace, the fewer places there are for the fault to "hide", and so



the a priori likelihood that a given covered entity is faulty must increase. Upper bounds for this increase is established by the base case – if a failing trace only covers a single component then that component must be faulty. We now formally establish that the PFL equations, but no SBFL measure, satisfies these properties.

**Proposition 2.** *The PFL equations satisfies the fault likelihood properties.*

*Proof.* We first prove the base property. We first show that if there is some failing trace  $t_k$  which only covers  $C_i$ , then nothing is more suspicious than it. Let  $t_1$  be a failing trace which only covers  $C_i$ . Then  $P(h_i^1) = \frac{w(C_i)}{w(C_i)} = 1$  (by Eq. (3)). Letting  $n$  abbreviate  $P(\bigvee_{j=2}^{|\mathbf{T}_1|} h_i^j)$ , we then have  $P(\bigvee_{k=1}^{|\mathbf{T}_1|} h_i^k) = (1+n) - (1n) = 1$  (by Eq. (2)). So  $P(h_i) = 1$  (by Eq. (1)). Thus, nothing can be more suspicious than  $C_i$ . We now show that if there is no failing trace which only covers  $C_j$ , then  $C_j$  must be less suspicious than  $C_i$ . Assume the antecedent, then for each  $t_k$  we have  $P(h_i^k) = \frac{w(C_i)}{w(C_i) + \dots + w(C_k)} < 1$  (by Eq. (3)). Thus  $P(\bigvee_{k=1}^{|\mathbf{T}_1|} h_i^k) < 1$  (by Eqs. (2) and (3)). Thus  $P(h_j) < 1$  (by Eq. (1)). Thus  $P(h_j) < P(h_i)$ , which means  $C_i$  is more suspicious than  $C_j$ .

We now prove the extended property. Let  $\mathbf{T}_1$  be a test suite in which all failing traces cover more than one component, and let  $\mathbf{T}_2$  be identical to  $\mathbf{T}_1$  except  $c_i^1 = 1$  and  $c_j^1 = 1$  in  $\mathbf{T}_1$  and  $c_i^1 = 1$  and  $c_j^1 = 0$  in  $\mathbf{T}_2$ . Let  $n$  abbreviate  $P(\bigvee_{m=2}^{|\mathbf{T}_1|} h_i^m)$   $P(h_i) = P(h_i^1) + n - (P(h_i^1)n)$  (by Eqs. (1) and (2)). It remains to first show that  $P(h_i^1)$  is greater in  $\mathbf{T}_2$ , and secondly show  $n$  has the same value for both test suites where  $n < 1$ . For the former, let  $P(h_i^1) = \frac{w(C_i)}{w(C_i) + \dots + x + \dots + w(C_{|c(t_k)|})}$  for both test suites (using Eq. (3)), where we let  $x = w(C_j)$  for  $\mathbf{T}_1$  (where  $w(C_j) > 0$ ), and  $x = 0$  for  $\mathbf{T}_2$  (as  $c_j^k \notin c(t_k)$  for  $\mathbf{T}_2$ ). So, the equation for  $P(h_i^1)$  is greater in  $\mathbf{T}_2$ . To show the latter, we observe that for all  $1 < m \leq |\mathbf{T}_1|$  we have  $P(h_i^m) < 1$  (by assumption each  $t_m \in \mathbf{F} \subseteq \mathbf{T}_1$  covers at least 2 components) and that  $P(h_i^m)$  is the same in both  $\mathbf{T}_1, \mathbf{T}_2$ , thus  $n < 1$  (by Eq. (2)) and  $n$  has the same value for both.

**Proposition 3.** *No SBFL measure satisfies either property.*

*Proof.* To show that no suspiciousness measure  $w$  satisfies the base property, we show that for any  $w$  we can construct a test suite in which (1) there is a failing trace which only covers  $C_i$ , (2) there is some  $C_j$  such that there is no failing trace which only covers it, and (3)  $w(C_i) = w(C_j)$ . A simple example is as follows. Let  $\mathbf{P} = \langle C_1, C_2, C_3, E \rangle$  and  $\mathbf{T} = \langle \langle 1, 1, 1, 1 \rangle, \langle 0, 1, 1, 1 \rangle, \langle 1, 0, 0, 1 \rangle \rangle$ . Thus the spectrum for  $C_1$  and  $C_2$  is  $\langle 2, 0, 0, 0 \rangle$ , and so  $w(C_i) = w(C_j)$ .

To show that no suspiciousness measure  $w$  satisfies the extended property, we show that for any  $w$  we can construct a pair of test suites  $\mathbf{T}_1$  and  $\mathbf{T}_2$  which are otherwise identical except (1)  $c_i^k = 1$  and  $c_j^k = 1$  in  $\mathbf{T}_1$  (2)  $c_i^k = 1$  and  $c_j^k = 0$  in  $\mathbf{T}_2$ , and (3)  $w(C_i) = w(C_j)$ . The simplest example is as follows. Let  $\mathbf{P} = \langle C_1, C_2, E \rangle$  and  $\mathbf{T}_1 = \langle \langle 1, 1, 1 \rangle \rangle$  and  $\mathbf{T}_2 = \langle \langle 1, 0, 1 \rangle \rangle$ . Thus the spectrum for  $C_1$  is  $\langle 1, 0, 0, 0 \rangle$  in both cases, and so  $w(C_i) = w(C_j)$ .

The proof of the last proposition suggests that there are large classes of test suites in which SBFL measures violate the properties. SBFL measures do not have

the resources to satisfy the properties because each  $C_i$ 's suspiciousness is only a function of its program spectrum, which itself is only a function of the  $i$ -th column of a coverage matrix.

## 5 Experimentation

In this section we discuss our experimental setup and our results. The aim of the experiment is to compare the performance of the PFL algorithm against SBFL measures at the practical task of finding a fault in large faulty programs.

### 5.1 Setup

We use the Steimann test suite in our experiments, described in Table 1 [33].  $M$  is the number of methods,  $UT$  the number of units under test,  $b$  the number of blocks of code,  $UT/b$  the mean number of  $UT$ s per block,  $t$  the number of test cases. The last column gives the number of program versions with 1/2/4/8/16/32 faults respectively. The average number of covered components that were faulty for all 1/2/4/8/16/32 fault benchmarks was found to be 1.00/1.92/3.63/6.71/11.81/20.02 respectively (7.52 on average). Steimann's test suite is the only test suite found to represent large programs with a large range of faults and a large number of program versions. For more about the suite see [33]. The suite came with a program that generated the coverage matrices (see [33]).

We used blocks as our atomic program entity, and only considered atomic blocks for all methods compared. A block corresponds to a maximal set of executable statements with the same traces covering them. This correspondence provides a natural grouping, as the degree of suspiciousness of lines of code is the same as the block to which they belong, and does not effect the fault localisation process from the user's point of view. In the majority of cases blocks represented a continuous chunk of the program and were similar in size – the average size of these blocks are reported in the  $UT/b$  column of Table 1, and can often be quite large.

**Table 1.** Benchmarks

Benchmark	$M$	$UT$	$b$	$UT/b$	$t$	1/2/4/8/16/32v
Daikon 4.6.4	14387	1936	48	40	157	353/1000/1000/...
Eventbus 1.4	859	338	68	5	91	577/1000/1000/...
Jaxen 1.1.5	1689	961	228	4	695	600/1000/1000/...
Jester 1.37b	378	152	25	6	64	411/1000/1000/...
Jexel 1.0.0b13	242	150	48	3	335	537/1000/1000/...
JParsec 2.0	1011	893	240	4	510	598/1000/1000/...
AC Codec 1.3	265	229	57	4	188	543/1000/1000/...
AC Lang 3.0	5373	2075	78	27	1666	599/1000/1000/...
Eclipse.Draw2d 3.4.2	3231	878	74	12	89	570/1000/1000/...
HTML Parser 1.6	1925	785	148	5	600	599/1000/1000/...

We now discuss techniques compared. We include all known SBFL techniques; which includes the 157 measures in [11], which itself includes 30 measures from the studies of Naish [15] and the 40 measures Lo [16]. We include the 30 genetic measures of Yoo [20], the Dstar measures [17] and the 6 “combination” measures of Kim et al. [21]. This brings the number of SBFL measures to almost 200, which to our knowledge is the largest comparison of SBFL measures to date. We now discuss the PFL techniques. We used the weighted model PFL- $w$ , and used the PPV, Ochiai, Kulczynski2, and Suppes measures (see preliminaries) as values for  $w$ . Not all measures could be tested because PFL techniques take slightly longer to run. To our knowledge SBFL and PFL techniques are the only ones which can feasibly scale to our experiments. Techniques which take 10 min on average to localise a fault in one program version would take almost a year to complete the experiment.

We evaluated the effectiveness of a technique using *avg* W-scores and A-scores (see preliminaries). We define higher level W/A-scores as follows. For each  $n \in \{1, 2, 4, 8, 16, 32\}$  a basic score for the  $n$ -fault versions of a given benchmark is the mean of the scores for all versions of that benchmark with  $n$ -faults. The score for the  $n$ -fault versions is the mean of the ten basic scores for the  $n$ -fault versions. The AVG score is the mean of the 60 basic scores. We used *Wilcoxon rank-sum tests* to determine to whether a technique’s 60 W/A basic scores were statistically significantly better than another (using  $p = 0.01$ ). To provide a *lower bound* for SBFL performance, we included scores for the Random measure (defined as a measure which outputs a random number). To provide an *upper bound*, we computed the unavoidable costs for W/A-scores (discussed in Sect. 2).

## 5.2 Results

We begin with overall results. Zoltar was the SBFL measure with the highest AVG W-score of 2.59. PFL-PPV improved on this score with a AVG W-score of 1.88. Thus, the user has to investigate 37.77% more code when using the best SBFL measure. Klosgen was the SBFL measure with the highest AVG A-score of 55.2. (PFL-PPV) improved on this score with a AVG A-score of 76.2. Thus, the user finds a fault immediately 27.56% less frequently using the next best SBFL measure. Both PFL-PPV’s W/A 60 scores were a statistically significant improvement over the next best performing SBFL measures using  $p = 0.01$ . Thus, the PFL approach was a substantial and significant improvement at localising faults.

We now discuss unavoidable cost (UC) scores. UC’s AVG W/A-scores were 76.45 and 1.38 respectively. PFL-PPV outperformed UC’s W-scores at 30/60 benchmarks, and UC’s A-scores at 24/60 benchmarks. It is thus theoretically impossible to design a strictly rational SBFL measure that can outperform PFL-PPV on many benchmarks.

To get an impression for the overall range of performance, we present the following additional AVG scores. Kulczynski2, Ochiai, Suppes, PPV (Tarantula), Random had AVG W-scores of 2.69, 2.97, 3.60, 3.88, 19.77, and AVG A-scores of 52.65, 52.47, 52.47, 49.73, 12.97 respectively. When PFL- $w$  was used in conjunction with the first three measures (scaled [0,1]), the techniques had AVG

W-scores of 2.31, 2.05, 2.09 and AVG A-scores of 70.57, 72.55, 72.30. Thus, all our PFL- $w$  approaches outperformed all SBFL measures regardless of choice of  $w$ . This suggests that the PFL- $w$  framework is more responsible for the improvement of fault localisation effectiveness than the choice of weight  $w$ .

We now discuss how measures behave as more faults are introduced into a program. In Figs. 3 and 4 we graphically compare a range of techniques. Firstly, we represented the unavoidable cost scores to show how PFL-PPV approximates (and in some cases exceeds) the idealised upper bounds for performance of strictly rational SBFL measures. Secondly, we represented Zoltar as it was the SBFL measure which the best W-scores. Thirdly, we represented Tarantula (equivalent to PPV) to show how using PFL-PPV improves performance. Each column represents a technique's score for the  $n$ -fault versions of that suite, with the key shade representing the value of  $n$  (for example, the W-score for PFL-PPV at the 2-fault benchmarks is 1.94).

We observed the following trends: In general, the more faults there were in a program the better an SBFL measure's W-scores, but the worse that measure's A-scores. A proposed explanation for this is that the more faults there were in

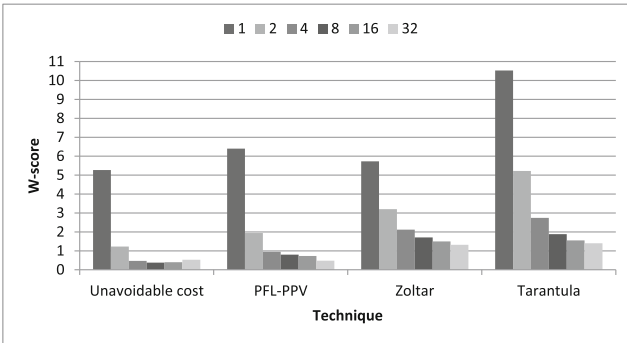


Fig. 3. W-scores for selected techniques

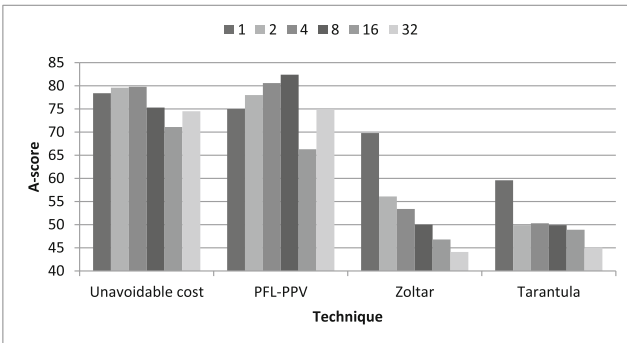
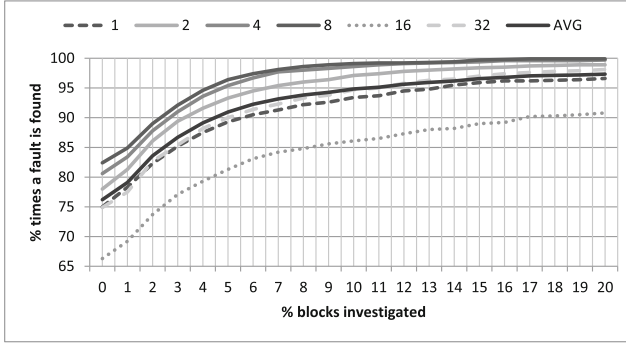


Fig. 4. A-scores for selected techniques



**Fig. 5.** PFL-PPV performance

a program, the more likely it was to find a fault early (due to increased luck – thus improving W-scores), but the less likely to find a fault immediately (due to increased noise – thus worsening A-scores). These trends were noticed in all of our SBFL measures, of which Zoltar and Tarantula are examples. By contrast, a negative trend for the A-scores was not noticed for our variants of PFL- $w$ , which demonstrated a superior ability to deal with noise introduced by multiple faults.

We now discuss Fig. 5. For each set of  $n$ -fault benchmarks, if  $y\%$  of the program versions received a W-score of  $\leq x\%$ , a point was plotted on that graph at  $(x, y)$ . The mean (AVG) of the 6 graphs is also plotted. The figure demonstrates that if we limit fault localisation to only 10% of the blocks, on AVG we would expect to find a fault 95% of the time using PFL-PPV. An outlier is that PFL-PPV does slightly worse on the 16-fault benchmarks. In general, the graph confirms the conclusion that PFL-PPV’s performance is not substantially worsened by the number of faults in the program.

We now discuss time efficiency. In our implementation it took under a second to find the most suspicious component in SBFL/PFL procedures. The complete PFL procedure (as per the algorithm in Sect. 4), took an average of 6.16s (with potential for optimisation) – thus establishing PFL’s negligible overhead.

In summary, PFL approaches substantially, and statistically significantly improve over the best performing SBFL approaches in our large multiple fault programs, and are comparably efficient. Furthermore, they outperform theoretically optimum performance of SBFL measures on a large class of benchmarks.

We briefly report results (AVG scores) in additional experiments which were of much lower quality and size. We generated 500+ 1/2/3/4 fault versions using the methodology and 10 SIR benchmarks of [34]. 80% of the faults were covered by all failing traces which made it less challenging for our techniques in terms of noise. The highest scoring SBFL measure was Kulczynski2 (K2) (W-score 8.76, A-score 33.75). PFL-Suppees, PFL-K2, PFL-Ochiai came second (W-scores 9.66, 10.27, 10.40, and A-scores 25.73, 24.43, 28.43 respectively). K2’s scores were not statistically significantly better. The BARINEL tool (see [34]) came 43rd overall and was not competitive. PFL-PPV came 6th after SBFL measures (W-score 11.39, A-score 29.15). The experiments confirm PFL as high performing.

## 6 Related Work

The most prominent lightweight approach to software fault localisation is SBFL [3], and provides the theoretical groundwork for the PFL approach. Research is driven by the development of new measures and experimentally comparing them on benchmarks [4–21]. Causal measures were introduced to SBFL in [11]. Threats to the value of empirical studies of SBFL is studied in [33]. Theoretical results include proving potentially desirable formal properties of measures and finding equivalence proofs for classes of measures [10, 11, 15, 22, 35]. Yoo et al. have established theoretical results that show that a “best” performing suspicious measure for SBFL does not exist [23], and thus there remains the problem of finding formal properties for lightweight techniques to exploit. We have tried to address this problem in this paper.

A prominent probabilistic approach is BARINEL, which differs to PFL insofar as it uses Bayesian methods to generate likelihoods of given hypotheses [34], and a minimal hitting set algorithm STACCATO to generate hypotheses. Their approach is designed for the simultaneous fault localisation of sets of multiple faults, and were only scalable to our additional experiments. Other heavyweight techniques are similarly unscalable [30, 36–41], which emphasises the importance of developing lightweight techniques such as PFL/SBFL.

In general, SBFL methods have been successfully used in the following applications. Firstly, in semi-automated fault localisation in which users inspect code in descending order of suspiciousness [29]. Secondly, in fully-automated fault localisation subroutines within algorithms which inductively synthesise (such as CEGIS [42]) or repair programs (such as GENPROG [43]). Thirdly, as a substitute for heavyweight methods which cannot scale to large programs [30, 36, 37]. Fourthly, as a technique combined with other methods [21, 24, 44–50]. In general, PFL may be used as a substitute for SBFL measures in all these applications. For a major recent survey we defer to Wong et al. [3].

## 7 Conclusions

In this paper we have presented a new formal framework which we call PFL, and compared it to SBFL in terms of (1) desirable theoretical properties, (2) its effectiveness at fault localisation and (3) its efficiency. Regarding (1), the PFL equations were formally proven to satisfy desirable fault likelihood properties which SBFL measures could not. Regarding (2), PFL-PPV was shown to substantially and statistically significantly (using  $p = 0.01$ ) outperform all known SBFL measures at W and A-scores in what is to our knowledge the largest scale experimental comparison in software fault localisation to date. We found that the user has to investigate over 37.77% more blocks of code (and finds a fault immediately 27.56% less frequently) than PFL-PPV when using the best SBFL measures. Furthermore, we show that for a third/quarter of our benchmarks it is theoretically impossible to design strictly rational SBFL measures which outperforms PFL-PPV’s W/A-scores respectively. Regarding (3), we found that the

PFL approach maintains a comparably negligible overhead to SBFL. Thus, our results suggest the PFL framework has theoretical and practical advantages over SBFL.

For future work, we would like to find additional suspiciousness measures for use with PFL-*w*. Secondly, we would like find a method to determine upper bound scores for PFL performance (similar to Naish's unavoidable costs). Thirdly, we would like to implement PFL in an easy to use tool for engineers.

## References

1. Zhivich, M., Cunningham, R.K.: The real cost of software errors. *IEEE Secur. Priv.* **7**(2), 87–90 (2009)
2. Collofello, J.S., Woodfield, S.N.: Evaluating the effectiveness of reliability-assurance techniques. *J. Syst. Softw.* **9**(3), 745–770 (1989)
3. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. *IEEE Trans. Softw. Eng.* (99) (2016)
4. Abreu, R., Zoetewij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: TAICPART-MUTATION, pp. 89–98. IEEE (2007)
5. Briand, L.C., Labiche, Y., Liu, X.: Using machine learning to support debugging with Tarantula. In: ISSRE, pp. 137–146 (2007)
6. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: ASE, pp. 273–282. ACM (2005)
7. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. *SIGPLAN Not.* **40**(6), 15–26 (2005)
8. Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.P.: Statistical debugging: a hypothesis testing-based approach. *IEEE Trans. Softw. Eng.* **32**(10), 831–848 (2006)
9. Zhang, Z., Chan, W.K., Tse, T.H., Jiang, B., Wang, X.: Capturing propagation of infected program states. In: ESEC/FSE, pp. 43–52. ACM (2009)
10. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM TSEM* **22**(4), 31:1–31:40 (2013)
11. Landsberg, D., Chockler, H., Kroening, D., Lewis, M.: Evaluation of measures for statistical fault localisation and an optimising scheme. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 115–129. Springer, Heidelberg (2015). doi:[10.1007/978-3-662-46675-9\\_8](https://doi.org/10.1007/978-3-662-46675-9_8)
12. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39 (2003)
13. Wong, W.E., Qi, Y., Zhao, L., Cai, K.Y.: Effective fault localization using code coverage. In: COMPSAC, pp. 449–456 (2007)
14. Pytlik, B., Renieris, M., Krishnamurthi, S., Reiss, S.: Automated fault localization using potential invariants. Arxiv preprint cs.SE/0310040 (2003)
15. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* **20**(3), 1–11 (2011)
16. Lucia, L., Lo, D., Jiang, L., Thung, F., Budi, A.: Extended comprehensive study of association measures for fault localization. *J. Softw. Evol. Process* **26**(2), 172–219 (2014)
17. Wong, W., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. *IEEE Trans. Reliab.* **63**(1), 290–308 (2014)

18. Wong, W.E., Debroy, V., Choi, B.: A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.* **83**(2), 188–208 (2010)
19. Wong, W.E., Debroy, V., Golden, R., Xu, X., Thuraisingham, B.M.: Effective software fault localization using an RBF neural network. *IEEE Trans. Reliab.* **61**(1), 149–169 (2012)
20. Yoo, S.: Evolving human competitive spectra-based fault localisation techniques. In: Fraser, G., Teixeira de Souza, J. (eds.) *SSBSE 2012*. LNCS, vol. 7515, pp. 244–258. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-33119-0\\_18](https://doi.org/10.1007/978-3-642-33119-0_18)
21. Kim, J., Park, J., Lee, E.: A new hybrid algorithm for software fault localization. In: *IMCOM*, pp. 50:1–50:8. ACM (2015)
22. Naish, L., Lee, H.J.: Duals in spectral fault localization. In: *Australian Conference on Software Engineering (ASWEC)*, pp. 51–59. IEEE (2013)
23. Yoo, S., Xie, X., Kuo, F., Chen, T., Harman, M.: No pot of gold at the end of program spectrum rainbow: greatest risk evaluation formula does not exist. Department of Computer Science, UCL (2014)
24. Xuan, J., Monperrus, M.: Learning to combine multiple ranking metrics for fault localization. In: *ICSME* (2014)
25. Wong, W.E., Qi, Y.: Effective program debugging based on execution slices and inter-block data dependency. *JSS* **79**(7), 891–903 (2006)
26. Keynes, J.M.: *A Treatise on Probability*. Dover Publications, New York (1921)
27. Baah, G.K., Podgurski, A., Harrold, M.J.: Causal inference for statistical fault localization. In: *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 73–84. ACM (2010)
28. Naish, L., Lee, H.J., Ramamohanarao, K.: Spectral debugging: how much better can we do? In: *Australasian Computer Science Conference (ACSC)*, pp. 99–106 (2012)
29. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: *ISSTA*, pp. 199–209 (2011)
30. Groce, A.: Error explanation with distance metrics. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 108–122. Springer, Heidelberg (2004). doi:[10.1007/978-3-540-24730-2\\_8](https://doi.org/10.1007/978-3-540-24730-2_8)
31. Lewis, D.: Causation. *J. Philos.* **70**(17), 556–567 (1974)
32. Fitelson, B., Hitchcock, C.: Probabilistic measures of causal strength. In: Illari, P.M., Russo, F., Williamson, J. (eds.) *Causality in the Sciences*. Oxford University Press, Oxford (2011)
33. Steimann, F., Frenkel, M., Abreu, R.: Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: *ISSTA*, pp. 314–324 (2013)
34. Abreu, R., Zoetewij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: *ASE*, pp. 88–99 (2009)
35. Debroy, V., Wong, W.E.: On the equivalence of certain fault localization techniques. In: *SAC*, pp. 1457–1463 (2011)
36. Mayer, W., Stumptner, M.: Evaluating models for model-based debugging. In: *ASE*, pp. 128–137 (2008)
37. Yilmaz, C., Williams, C.: An automated model-based debugging approach. In: *ASE*, pp. 174–183. ACM (2007)
38. Zeller, A.: Isolating cause-effect chains from computer programs. In: *SIGSOFT/FSE*, pp. 1–10. ACM (2002)
39. Jeffrey, D., Gupta, N., Gupta, R.: Effective and efficient localization of multiple faults using value replacement. In: *ICSM*, pp. 221–230. IEEE (2009)



40. Zhang, X., Gupta, N., Gupta, R.: Locating faults through automated predicate switching. In: ICSE, pp. 272–281. ACM (2006)
41. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005). doi:[10.1007/11513988\\_23](https://doi.org/10.1007/11513988_23)
42. Jha, S., Seshia, S.A.: Are there good mistakes? A theoretical analysis of CEGIS. In: 3rd Workshop on Synthesis (SYNT), pp. 84–99, July 2014
43. Goues, C.L., Nguyen, T., Forrest, S., Weimer, W.: GenProg: a generic method for automatic software repair. *IEEE Trans. Softw. Eng.* **1**, 54–72 (2012)
44. Baudry, B., Fleurey, F., Le Traon, Y.: Improving test suites for efficient fault localization. In: ICSE, pp. 82–91. ACM (2006)
45. Gopinath, D., Zaeem, R.N., Khurshid, S.: Improving the effectiveness of spectrum-based fault localization using specifications. In: ASE, pp. 40–49 (2012)
46. Abreu, R., Mayer, W., Stumptner, M., van Gemund, A.J.C.: Refining spectrum-based fault localization rankings. In: SAC 2009, pp. 409–414 (2009)
47. Debroy, V., Wong, W.E.: A consensus-based strategy to improve the quality of fault localization. *Softw. Pract. Exp.* **43**(8), 989–1011 (2013)
48. Lucia, L. D., Xia, X.: Fusion fault localizers. In: Automated Software Engineering (ASE), pp. 127–138. ACM (2014)
49. Debroy, V., Wong, W.E.: On the consensus-based application of fault localization techniques. In: 2011 IEEE 35th Annual COMPSACW, pp. 506–511, July 2011
50. Ju, X., Jiang, S., Chen, X., Wang, X., Zhang, Y., Cao, H.: HSFal: effective fault localization using hybrid spectrum of full and execution slices. *JSS* **90**, 3–17 (2014)