# Evaluating Fault Localization Techniques with Bug Signatures and Joined Predicates

Roman Milewski[1], Simon Heiden[1] and Lars Grunske[1]

[1]Software Engineering, Humboldt-Universität zu Berlin, Germany

**Abstract**

Predicate-based fault localization techniques have an advantage over other approaches by not only determining the location of a fault but also potentially giving the developer additional information to understand it. In this paper, we evaluate the accuracy of predicate-based bug signatures based on the Defects4J benchmark. Additionally, we try to improve the predicate-based approach by extending it with joined predicates, a technique for combining multiple predicates, to extract even more information. To validate our results, we compare our approaches with established spectrum-based fault localization methods.

**Keywords**

SBFL, predicate-based fault localization, bug signatures

## 1. Introduction

Searching for bugs, debugging, and fixing bugs are important parts of the software development cycle. A lot of time and effort is spent on minimizing the amount of bugs in a program, but this is usually a costly and difficult process [1].

While some bugs can be found by static analysis, e.g., compilers fail to compile a program, a lot of bugs only manifest under special conditions or are simply not detectable at all by compilers or static analysis [2]. For those bugs, the only detection approach is dynamic analysis, e.g., extracting information about the program during execution. Basic methods for findings bugs include the usage of print statements to extract information about the state of variables or the path taken through the program. More advanced methods include using a debugger or slicing [3]. All these methods give the programmer more information about the program states which lead to the bug, or reduce the amount of code that needs to be examined, thus helping the programmer to identify the faulty code.

There have been multiple approaches to automate parts of the fault localization process. A popular approach, shared by most techniques, is collecting data during correct and faulty executions of the program code and then analyzing it to produce information about the possible bug locations [4]. Tarantula, one of the first automated fault localization techniques, compares the executed lines in failed and successful program executions and assigns lines executed by more failed runs a higher score [5]. There exist more advanced statistics-based approaches

*29th international Workshop on Concurrency, Specification and Programming (CS&P'21)*

milewskr@hu-berlin.de (R. Milewski); heiden@informatik.hu-berlin.de (S. Heiden); grunske@informatik.hu-berlin.de (L. Grunske)



that use, e.g., path spectra [6, 7], def-use pairs [8, 9], information-flow pairs [9], dependence chains [10], (potential) invariants [11, 12, 13, 14, 15, 16], predicates [17, 18, 19, 20], predicate pairs [21] or method call sequences [22]. Many of these techniques provide the user with more information than only the potential locations of the bug and, thus, provide more context to help the user understand the cause of the bug.

One leading the way to providing this context is statistical debugging, a technique introduced by Liblit et al. [17], to isolate bugs by profiling several runs of the buggy program and then using statistical analysis to pinpoint the likely cause of failure. It uses predicates to provide extra information, as each predicate can hold information about a state of the program or information about the control flow of the program. Predicates can be used to generate bug signatures [23], which are sets of said predicates, providing information about the cause or effect of a bug or other information helpful for debugging. This provides the *context* for where and how the bug occurs and, thus, can be used to understand and locate the bug itself.

As an example, when confronted with a bug which leads to a program crash, the usual entry point for the programmer is the stack trace (a report of the active stack frames) and, thus, the location the program failed in. Often, the programmer is now presented with a scenario, in which it is obvious *why* the code failed at this location (e.g., a null-reference) but not *how* the program managed to arrive at this state. The programmer now has to work backwards through the stack trace, trying to find the location of the bug that caused the failure. However, often the bug is not directly part of the stack-trace. It may be in a function that manipulated some variables but already returned or – in a multi-threaded scenario – had happened in another thread.

A bug signature tries to explain a bug by showing the programmer a set of relevant predicates that have been observed during the execution of the program prior to the crash. These sets are extracted by comparing failed program executions to normal executions (i.e., without a crash) and have a high chance of being correlated to the cause of the program failure [23]. This gives the programmer additional entry points into the code, at locations with some relevance to the bug, and may thus expedite the process of identifying the bug. Additionally, the bug signatures proposed by Hsu et al. [23] consist of predicates which provide information about either the state of the program, its control flow or its data flow prior to the crash. This is additional information that the programmer can use to reconstruct the program execution more easily and which may help them locate the bug.

In this paper, we want to explore the usage of predicates in a java environment. We implemented a tool for instrumenting the java bytecode to collect the predicate data during execution, a mining algorithm that can find the top-k bug signatures from this data and, finally, compare the bug localization performance of our approaches with a state-of-the-art bug localization approach. Additionally, we explore the idea of *joined predicates*, i.e., predicate chains consisting of one or more predicates, as an improvement for predicated bug signatures.

**RQ1:** Can a bug signature based approach be used for bug localization?

**RQ2:** Can a bug signature based approach be enhanced by using *joined predicates*?

## 2. Background

In this section, we explain important concepts of predicates, itemsets and generators used in this work.

### 2.1. Predicates

Predicates are part of an approach developed by Liblit et al. [17]. Here, a program is instrumented at prior defined *instrumentation sites*. At each of these sites, one or multiple *predicates* are evaluated (as true or false) and the evaluation results are tracked. Liblit et al. [17] used the following instrumentations:

> **branches:** At each conditional statement, one *predicate* tracks whether the true branch is taken at runtime, and one *predicate* tracks the false branch.
>
> **returns:** Sometimes, function return values are used to track success or failure (either directly as boolean or with a numeric value). At each scalar returning method call, six *predicates* are used: $< 0, \leq 0, = 0 \neq 0, \geq 0, > 0$.
>
> **scalar pairs**: At each assignment to a scalar variable $x$, for each other in scope scalar variable $y_i$ the following six *predicates* are possible: $x < y_i$, $x \leq y_i$, $x = y_i$, $x \neq y_i$, $x \geq y_i$ and $x > y_i$.

Additionally, we define and track the following predicates:

> **nullness:** At each assignment of an object to a variable, we track whether the object equals `null`.

### 2.2. Itemsets, Generators and Gr-Tree

In this work, we will use the following terminology for itemsets and generators:

- An *itemset* $I$ is an unordered set of distinct items $I = \{i_1, i_2, ..., i_m\}$.
- A *transaction* $t$ is a set of items, and $t \subseteq I$.
- A *transaction database* $T$ is a set of transactions $T = \{t_1, t_2, ..., t_n\}$.
- A *frequent itemset* is an itemset that appears in at least $x$ transactions from a transaction database.
- The *support* of an itemset is the number of transactions in a transaction database that contain the itemset.
- A *generator* is an itemset $X$ such that there does not exist an itemset $Y$ strictly included in $X$ that has the same support.

Frequent itemset mining algorithms are a data mining technique used to find all frequent itemsets for a given transaction database. In a naive approach, the search space is exponential to the number of items in the transaction database. Starting with the Apriori algorithm [24], better algorithms for finding frequent itemsets have been developed. One important difference between most frequent itemset mining algorithms and the one used by our approach based on [25] is the specification of *support* as *positive support* and *negative support*. A transaction has positive support if it is the result of a successful run, and it has negative support if it is the result of a failed run.

Gr-trees (generator trees) and a depth-first Gr-growth algorithm for mining frequent generators were introduced in [26]. A Gr-tree is a typical trie structure, providing a compact representation of a transaction database. Each Gr-tree has a prefix that is the distinct itemset prefixing all items in the Gr-tree. In [25] and [27], the authors provide improved algorithms, albeit still using a Gr-tree as basis. We base our implementation on the work done in [25].

# 3. Fault Localization with Mined Bug Signatures via Predicates and Joined Predicates

## 3.1. Generating Predicates

In this first step, the goal is to modify the existing java code in such a way that we can gather predicate information during later execution.

### 3.1.1. Instrumentation with ASM

The ASM library[1] [28] is one of the more popular frameworks for instrumenting java code. We used its ability to manipulate the bytecode of already compiled java classes to *instrument* java programs and to generate predicate data if the program is run afterward. The core ASM library provides an *event-based* representation of a compiled java class, with each element of the class being represented by an event. The core ASM library uses the visitor design pattern in which each class is passed to one (or multiple) `ClassVisitors` which can modify and transform it. The same happens for fields and methods. We implement our own visitors, which react to the events emitted by ASM while parsing the bytecode. Now, when ASM is reading a class and visits a relevant instruction, we can generate predicates and insert instructions for triggering the evaluation of those predicates: if the predicate evaluates as true during the execution of the code, the program saves this information.

### 3.1.2. Joined Predicates

In addition to the predicates introduced in [17], our approach aims to evaluate the usefulness of *joined predicates*. In our context, joined predicates are composed of multiple ”traditional” predicates and carry information about their order of appearance in the program flow.

As a motivating example, in the buggy code shown in Listing 1, a classical approach would place predicates in lines 3 and 5, among others. The included bug only manifests if *both* branches

[1] https://asm.ow2.io

Listing 1: Example code with tests

```
1  private boolean anyTrue(boolean first, boolean second) {
2     int x = 0;
3     if (first)
4        x++;
5     if (second)
6        x++;
7     return x == 1; // bug, should be: x >= 1
8  }
9  public Test1() { assert(!anyTrue(false,false)); }
10 public Test2() { assert(anyTrue(true,false)); assert(anyTrue(false,true)); }
11 public Test3() { assert(anyTrue(true,true)); } // this fails
```

at line 3 and 5 are true, but not in any other combination. If we run all example tests from Listing 1, the predicate database would contain the same predicates L:3[true] and L:5[true] after running either test 2 or test 3, making them impossible to distinguish based on only this information. Our approach would now generate 4 additional joined predicates: 1) L:3[true] $\succ$ L:5[true], 2) L:3[false] $\succ$ L:5[true], 3) L:3[true] $\succ$ L:5[false], and 4) L:3[false] $\succ$ L:5[false].

These joined predicates are different from the two separate predicates, as they additionally encode the order of execution. For example, L:3[true] $\succ$ L:5[true] means: the branch at line 3 was evaluated to true, and then, the *next* branch statement was at line 5 and was true, as well. This joined predicate is only evaluated to true in the failed test case, allowing us to distinguish it from the other test cases.

### 3.2. Gathering execution data

In the next step, we execute the instrumented code to generate the trace data. During execution, we collect predicate execution data for, e.g, each test in a test suite, while also classifying each trace as successful or failed, depending on the test outcome. The approach expects multiple execution profiles and at least one failed execution to work correctly.

The current prototype of our approach uses a simple rule for generating joined predicates: each pair of two simple predicates is a possible joined predicate. During execution of the instrumented code, each time a predicate gets evaluated, we dynamically create a new joined predicate from the last evaluated simple predicate and the currently triggered one and add it to our list of joined predicates, if it has not been encountered, previously. It is possible to use more complex rules for the creation of joined predicates, e.g., using the previous two predicates to create a joined predicate consisting of three simple ones or even only considering specific types of predicates.

During the execution of tests, all predicates for an instrumented location are directly evaluated by the executing code, and the results are stored in a list. This list contains all predicates and joined predicates that got triggered during a run and is exported at the end of the run.

### 3.3. Mining bug signatures

In this step, we try to find the top-k discriminative *bug signatures*. For this, we use an adaptation of the mining algorithm presented in [25].

The input is a database of transactions with each transaction containing a single run of the program and the information if that run was successful. A *Gr-tree* (see subsection 2.2) is constructed from the database. The Gr-tree stores all its items in its head table and links them to their corresponding nodes in the tree structure. The output is a list containing the top-k discriminative signatures.

Our mining algorithm implements the pseudo code mining algorithm described in [25]. Most differences stem from the java implementation and do not alter the basic idea. The algorithm also has some parameters controlling the size of the mined bug signatures, the cutoff point for significance, and k, the size of the returned list. We used the same default parameters as defined in [25].

## 4. Experimental Setup

### 4.1. Choosing a base for a comparison

The result of bug signature identification is different from other fault localization techniques. Other fault localization techniques output a list of program elements ordered by suspiciousness, while a bug signature approach instead returns a ranked list of bug signatures. Each bug signature consists of one or multiple program elements, describing a supposed bug context. The advantage of this additional information is difficult to quantify, as each programmer may process the information from such a bug context in a unique way and most likely different than a program would. A bug signature can therefore consist of program elements that do not have an immediately obvious relation to the bug under examination.

We decided to quantitatively compare our approaches to *spectrum-based fault localization (SBFL)*, a popular fault localization technique. Thus, we compare the following approaches:

| | |
|---|---|
| *SBFL* | state-of-the-art *SBFL* as implemented in *BugLoRD*[2], using JaCoCo[3]to collect execution profiles (test coverage data) and using the DStar metric [29] to calculate suspiciousness scores |
| Predicates | our approach using predicates |
| Joined Predicates | our approach using predicates and joined predicates (see subsubsection 3.1.2) |

### 4.2. The scoring algorithm

The result of our approach, described in section 3, is a ranked list of *bug signatures*. The result of SBFL is a ranked list of source code lines. As a bug signature can contain information about one or more code locations (note that a bug signature may also contain additional information

[2]https://github.com/hub-se/BugLoRD
[3]https://www.eclemma.org/jacoco/index.html

that has no equivalent in SBFL), we developed an algorithm to calculate suspiciousness scores for a bug signature. A scoring approach for fault localization experiments is a metric, first proposed by Renieres and Reiss in [30] and used by others [31, 32], based on static program dependencies. In the first step, a *program dependence graph (PDG)*, a graph that contains a node for each expression in the program, is constructed from the source code. Next, nodes in the PDG get marked ”faulty” for being related to the bug under inspection. Using the results of the fault localization, for each prediction, one or more nodes can be marked as ”reported” if they contain the predicted program element. Now, a score can be calculated as the fraction of the *PDG* that would need to be examined to get from a ”reported” node to a ”faulty” one by shortest path.

We used Soot[4] [33, 34] to generate intra-procedural PDGs. Soot stores the code for each method in a `Body`. Each `Body` contains a chain of `Units` which represent the actual code inside of a method. Each `Unit` represents a statement in the original source code. We now define a `codelocation` as the line of code in the source code and all following lines not belonging to another `Unit`.

---

**Definition 1 - codelocation**

---

Given a reference to a line of java source code (e.g., `MyClass:7`) and a corresponding java method in a Soot representation, a `codelocation` is the source code line of a `Unit` corresponding to the referenced line and all lines after that, until the next `Unit` or the end of its method.

The result of our algorithm is a ranked list of bug signatures. Each bug signature contains one or multiple predicates. Each predicate has a reference to the line of source code, where it was evaluated. Now, we can generate `codelocations` from a bug signature by using all references to lines of source code inside the bug signature. We generate the `codelocations` for the bug under examination (using the source code lines associated with the bug) and the `codelocations` from the bug signatures reported by our algorithm (or the *SBFL* results).

---

**Definition 2 - path cost**

---

The `path cost` is the sum of all edge costs on the shortest path between two `codelocations`.

In our setting, each edge between two classes weighs 25, each edge between two methods weighs 10 and each edge between two `Units` inside a method weighs 1. The higher cost of class and method transitions represents the additional mental effort while debugging. The numbers are chosen based on personal experience and can be adjusted.

Both SBFL and our approach rank their output by an internal suspiciousness score (see [25] for a definition of *discriminative significance (DS)* and [35] for a definition of the used SBFL metrics). Often, multiple elements get assigned the exact same score, due to the nature of the used formulas, the limited availability of diverse enough execution data (i.e., test cases with diverse execution profile) or simply due to identical execution behavior that is dictated by the implementation itself. Even worse (from an evaluation approach), a bug signature, by

[4] https://soot-oss.github.io/soot/

definition, usually contains multiple predicates and/or joined predicates and thus contains multiple `codelocations`. This means that the actual position of a `codelocation` in a ranking is nondeterministic. For each of such cases, we therefore have a *best* and *worst* case. In the best case, the `codelocation` that will lead to the smallest score is evaluated first, and in the worst case, it is evaluated last among all `codelocations` with the same suspiciousness score.

$$minCodeLocations_{before}(c_i) = |\{c_j \in C \mid susp_{DS}(c_j) > susp_{DS}(c_i)\}|$$
$$maxCodeLocations_{before}(c_i) = |\{c_j \in C \mid susp_{DS}(c_j) \geq susp_{DS}(c_i)\}|$$

An alternative, more correct, definition for $maxCodeLocations_{before}(c_i)$ would subtract 1 to not count $c_i$ twice. Now, by combining the `path cost` and `codelocation`, we can define our `EvaluationScore`:

$$Score_{best} = minCodeLocations_{before} + \left(pathCost * \sqrt{minCodeLocations_{before} + 1}\right)$$
$$Score_{worst} = maxCodeLocations_{before} + \left(pathCost * \sqrt{maxCodeLocations_{before} + 1}\right)$$
$$Score_{avg} = \frac{1}{2} * (Score_{best} + Score_{worst})$$

Note that $Score_{avg}$ is just the average of $Score_{best}$ and $Score_{worst}$. To calculate a real average `EvaluationScore`, one would use $averageMinCodeLocations_{before}$ and `path cost`. In our definition, the `path cost` is also included in the average.

The `EvaluationScore` is based on the usefulness of a bug prediction to a programmer while debugging. The `EvaluationScore` is 0, a perfect score, if the first reported `codelocation` is exactly the line of source code associated with the bug. The `EvaluationScore` grows by one for every additional `codelocation` to check. Additionally, the `EvaluationScore` is scaled with the number of `codelocations` already checked, as a programmer checking the first few locations will be motivated to look deeper, but when having already looked at multiple locations before, might stop his investigation sooner. So, while there might be an incredibly long path in the `Unit`-graph linking the suspected `codelocation` and the bug location, it is highly unlikely that this connection would be useful to the programmer. I.e., the more the `path cost` between code locations and bug increases, the less likely it is to track down the bug.

### 4.3. Evaluation Subjects

The Defects4J Benchmark[5] [36] is a collection of open source projects with each being available in multiple buggy versions. Each buggy version consists of the respective source code and a change set with changes *exclusively* related to the bug. This omission of other changes (e.g., refactorings) in the change set makes it more reliable as a source for the lines of code related to the bug. Sobreira et al. [37] did an analysis of many Defects4J bugs and showed a method for linking a bug to lines of code. The big number of real (not artificially created) bugs from many different projects make this a sensible benchmark choice for our purposes.

We used the bugs in the version 2.0.0 from the projects in Table 1. From the original set of bugs, we had to exclude 21 bugs from our final results. The most common reasons were problems

[5]https://github.com/rjust/defects4j

**Table 1**
Used projects from the Defects4J Benchmark

| project | size[loc] | #bugs |
|---|---|---|
| jfreechart (Chart) | 96k | 26 |
| commons-cli (Cli) | 2k | 39 |
| commons-codec (Codec) | 3k | 18 |
| commons-csv (Csv) | 1k | 16 |
| gson (Gson) | 6k | 18 |
| commons-lang (Lang) | 22k | 64 |
| commons-math (Math) | 84k | 106 |
| joda-time (Time) | 90k | 26 |
| Total | | 313 |
| Applicable after Instrumentation | | 292 |
| Applicable after Runtime Eval. | | 162 |

with compilation of the source code, crashes of the instrumenter and *JVM* crashes during test execution. During the runtime evaluation, we encountered some additional problems, because we could not relate the `Unit` containing the line of source code to the given bug/prediction, or because the evaluation produced a time out. This leaves us with 162 bugs which were used for the following graphs.

## 5. Results

In Figure 1, we see that $Score_{best}$ (`EvaluationScore` if the reported location is examined first among all locations ranked equally; see section 4 for more details) for both joined predicates and predicates is lower (better) than for SBFL. $Score_{worst}$ is more similar for all three approaches, with the joined predicates being significantly worse than simple predicates. If we look at the included p-values, we can see that the `EvaluationScores` for both predicate approaches are significantly different in our data set. The only non significant difference is between the worst `EvaluationScore` for joined predicates and SBFL.

In Table 2, we can see the mean values for $Score_{best}$ and $Score_{worst}$. For a better visualization, we can look at Figure 3(left), where the total `EvaluationScores` for each approach are summed up. Both predicate approaches have significantly lower total `EvaluationScores` than the SBFL approach when comparing $Score_{best}$. When comparing $Score_{worst}$, the results are more even, with only simple predicates being significantly different than the other two. If we look at the median values in Table 2, we can see the very small difference between all three approaches for $Score_{best}$. This shows us that a lot of bugs have similar, very small scores and most of the

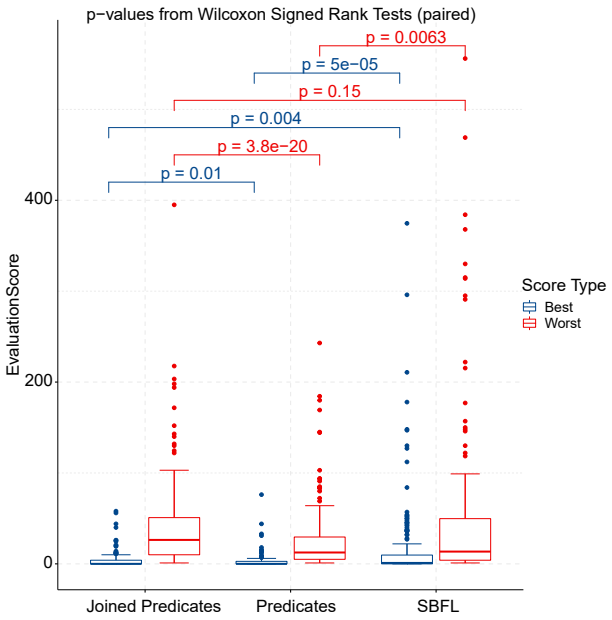


**Figure 1:** Summary of results

differences stem from a few bugs with high (i.e. bad) scores. When comparing the medians for $Score_{worst}$, the median for joined predicates is nearly double as big as the medians of the other approaches.

In Figure 3(right), we further split up our results regarding the different Defects4J projects used. It shows the different mean values for $Score_{avg}$ separated by project. Here, we can see that both predicate approaches have lower (i.e., better) results for nearly all projects, with only 'cli' in favor of SBFL.

Because a lot of the scores are close to or equal to 0, i.e., the best result, we additionally looked at the density of scores. Figure 2 shows the density distribution of $Score_{avg}$ from 0 to 75. The dashed lines show the mean values for each approach. Here, we see that the simple predicate approach has the highest density in the 0-10-range. This represents a lot of very low $Score_{avg}$ results – results where the correct `codelocation` was highly ranked, while not having too many similarly ranked `codelocations`. SBFL seems to have both very good scores, but also a lot of very bad ones, leading to a significantly worse mean value.

**Table 2**
Mean and median values for $Score_{best}$ and $Score_{worst}$

| Approach | $\overline{Score_{best}}$ | $\widetilde{Score_{best}}$ | $\overline{Score_{worst}}$ | $\widetilde{Score_{worst}}$ |
|---|---|---|---|---|
| Joined Predicates | 3.97 | 0 | 41.25 | 26.33 |
| Predicates | 2.94 | 0 | 26.14 | 12.50 |
| *SBFL* | 18.94 | 1 | 48.34 | 13.50 |

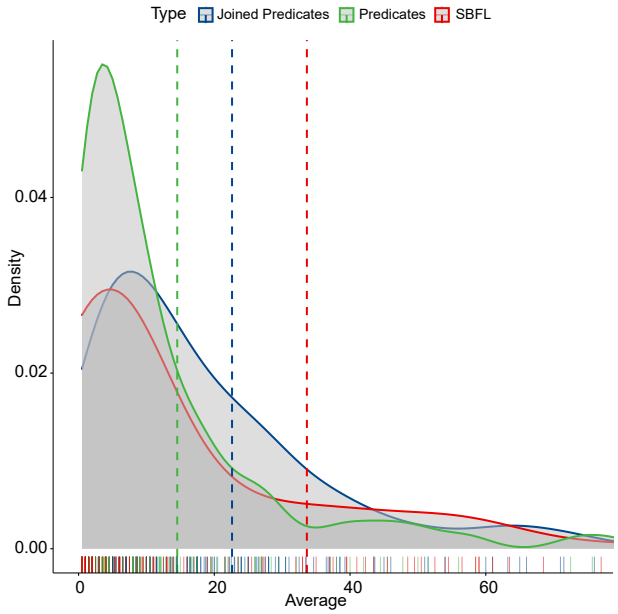


**Figure 2:** Density plot for $Score_{avg}$ with x-axis limited to 75

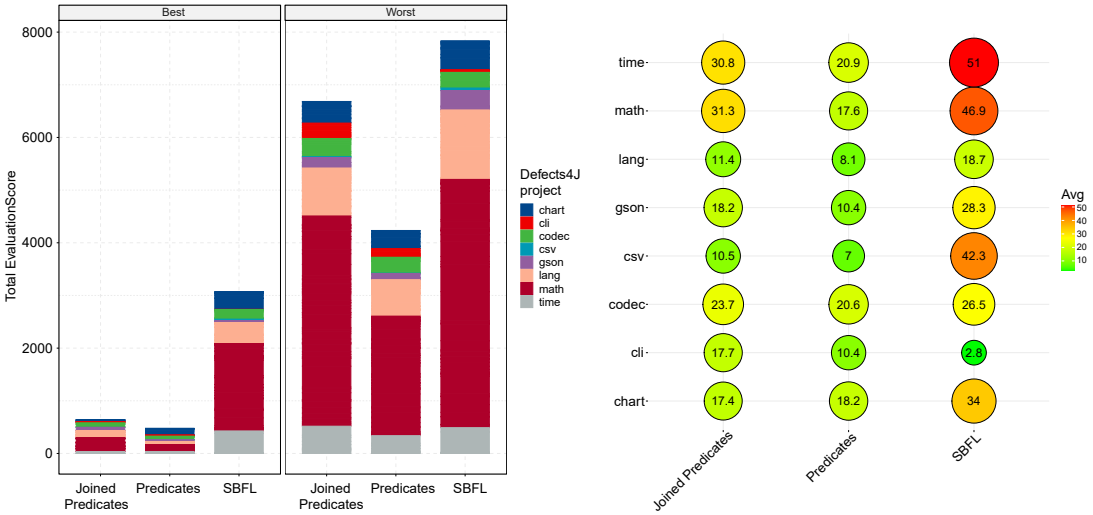


**Figure 3:** Left: Best and Worst `EvaluationScore`, Right: Mean values for $Score_{avg}$, separated by Defects4J project

## 5.1. Spread and Path Cost

Next, we analyze the `EvaluationScore` in more detail to better understand the differences between the predicate based and SBFL approaches. For this, we split up the `EvaluationScore` into its two main components:

- *path cost*: number of `Units` in the graph between the reported `codelocation` and the buggy `codelocation`
- *penalty*: number of `codelocations` to be examined before finding the reported `codelocation` in the ranking

We see in Figure 4(left), that SBFL has a *significantly* lower `path cost` than the two predicate based approaches. This is expected, as SBFL directly outputs a line of code as result, while a bug signature from a predicate based approach also contains additional information about the state of the program in a faulty run. The code lines we extract and use are actually the locations of the instrumentation sites for the predicates that the bug signature consists of. Since not every executable line is a valid instrumentation site for a predicate, but some of the evaluated bugs are located on such lines of code, the predicate based approaches can not reach a `path cost` of 0 in those cases.

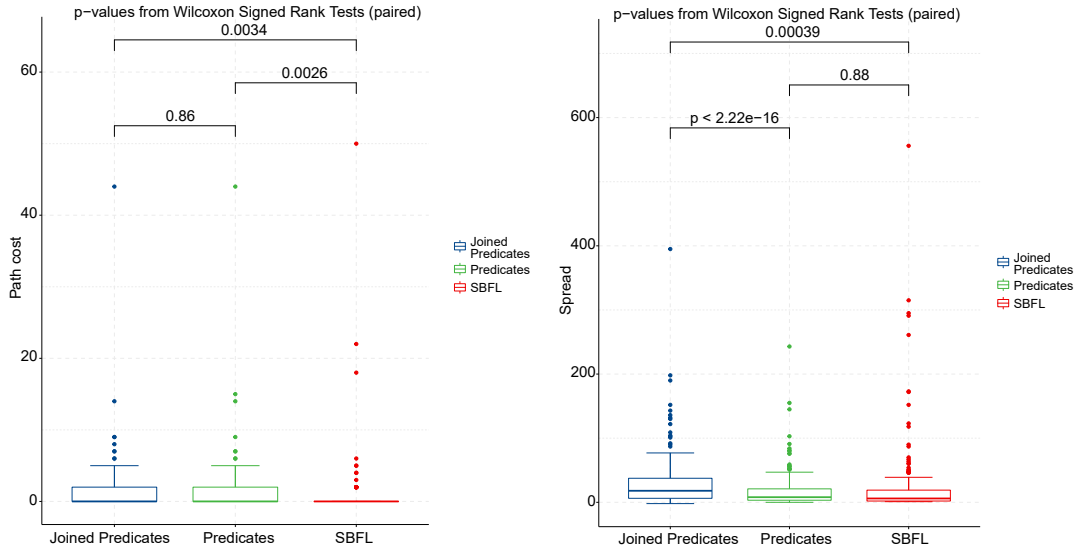


**Figure 4:** Left: Comparison of `path cost` (number of `Units` in the graph between the two `codelocations`) Right: Comparison of spread $(maxCodeLocations_{before} - minCodeLocations_{before})$

Comparing the spread in Figure 4(right), one can see that joined predicates differ *significantly* from the other approaches. This leads to the conclusion that joined predicates have more ties in the ranking of its results which makes sense, as in addition to bug signatures having the same rank, all predicates in the bug signature will have the same rank as the bug signature, itself. Additionally, a bug signature consisting of three predicates can have up to three `codelocations` , while a bug signature with joined predicates increases this number by one for each joined predicate that is part of the bug signature. For example, a bug signature consisting of 2 joined and one simple predicate can have up to five `codelocations`.

An additional observation we made during our experiments is the partly unfeasibly high run time of our experiments. Some bug signature mining runs took multiple days of computation

time, making the approach in its current form largely unusable in a real world scenario – at least without further optimization.

## 6. Conclusions

After having analyzed the data from section 5, we summarize the following conclusions:

| **RQ1:** Can a bug signature based approach be used for bug localization? |
|---|

**Yes**, as we have seen in, e.g., Figure 1 and Figure 3(left), a bug signature (predicate) based approach consistently gets a lower `EvaluationScore` than *SBFL*.

These are welcome results, because while we expected it, as bug signature approaches have been shown as effective in other languages, e.g., [23, 25] showed predicate based approaches in C, there were multiple additional, java-specific difficulties with regard to the instrumentation.

| **RQ2:** Can a bug signature based approach be enhanced by using *joined predicates*? |
|---|

**Maybe**, but we did not manage to find any *significant* improvement by using our (rather simple) joined predicate approach over the single predicate approach. In our studies, we only evaluated joined predicate with a simple ”two-following” generation rule, and we did not exhaust other possible rules for generating joined predicates, yet. While a three-following rule might deem too time-intensive, we have many ideas for more elaborate rules concerning two predicates. Another factor could be our definition of the `EvaluationScore`. As we discussed in section 4, we did not find a way to use an established scoring algorithm to evaluate bug signatures. While the problems we faced were beyond the scope of this project, most of them could be solved in the future, allowing for a new evaluation of our results or future results.

We still strongly believe that a bug signature based approach can be enhanced by using joined predicates, but with other joined predicates beyond a simple combination of two following predicates into a joined one.

Another point is the differences between SBFL and predicate approaches visible in Figure 4(left). SBFL has a *significantly* lower average `path cost` than the two predicate approaches. We can explain this with the different output formats used by the two techniques. The fact that the predicate based approaches still get better overall scores thus must mean they perform better in the non `path cost` part of our `EvaluationScore`.

Furthermore, the bug signature mining algorithm described in [25] which our approaches are based on, too, actually has a parameter for controlling how many predicates can be part of a single bug signature. A bigger size limit significantly increases the computation time, which is why we used a size limit of 3, as recommended in [25]. This could have been chosen too small, as in the joined predicate approach, the joined predicates now compete with the others for a spot inside a bug signature. On the one hand, while a bigger bug signature may potentially be more helpful to a real programmer, it would negatively impact our score, as it does not differentiate between processing multiple predicates inside one bug signature or multiple small bug signatures containing the same predicates. Even worse for our evaluation (and likely any

other based on lines of source code), just one "correct" predicate inside a bug signature is all that is needed to score it positively, while ignoring all the others inside the bug signature. On the other hand, in [25], the authors noted that increasing the size of the mined bug signatures increases the predictive power but increases the computation time.

Especially noteworthy is that *all* the extra information a programmer is supposed to get out of a bug signature, e.g., it containing a predicate with $var = null$ for a $var$ where $null$ is not expected, can lead to another investigation path than just the information to investigate that line. An evaluation method considering those aspects could have completely different results than ours.

**Acknowledgements** Lars Grunske would like to thank the intensive care unit of the Achenbach hospital in Königs Wusterhausen for the tremendous help during his Covid-19 case.

## References

[1] M. Zhivich, R. K. Cunningham, The real cost of software errors, IEEE Security & Privacy Magazine 7 (2009) 87–90. doi:10.1109/msp.2009.56.

[2] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, J. Penix, Using static analysis to find bugs, IEEE Software 25 (2008) 22–29. doi:10.1109/ms.2008.130.

[3] M. Perscheid, B. Siegmund, M. Taeumel, R. Hirschfeld, Studying the advancement in debugging practice of professional software developers, Software Quality Journal 25 (2016) 83–110. doi:10.1007/s11219-015-9294-2.

[4] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (2016) 707–740. doi:10.1109/tse.2016.2521368.

[5] J. A. Jones, M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05, ACM Press, 2005, pp. 273–282. doi:10.1145/1101908.1101949.

[6] T. Reps, T. Ball, M. Das, J. Larus, The use of program profiling for software maintenance with applications to the year 2000 problem, ACM SIGSOFT Software Engineering Notes 22 (1997) 432–449. doi:10.1145/267896.267925.

[7] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, K. Vaswani, Holmes: Effective statistical debugging via efficient path profiling, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, IEEE, 2009, pp. 34–44. doi:10.1109/icse.2009.5070506.

[8] R. Santelices, J. A. Jones, Y. Yu, M. J. Harrold, Lightweight fault-localization using multiple coverage types, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, IEEE, 2009, pp. 56–66. doi:10.1109/icse.2009.5070508.

[9] W. Masri, Fault localization based on information flow coverage, Software Testing, Verification and Reliability 20 (2009) 121–147. doi:10.1002/stvr.409.

[10] R. A. Assi, W. Masri, Identifying Failure-Correlated Dependence Chains, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 607–616. doi:10.1109/ICSTW.2011.42.

[11] T. B. Le, D. Lo, C. L. Goues, L. Grunske, A learning-to-rank based fault localization

approach using likely invariants, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, ACM, 2016, pp. 177–188. doi:10.1145/2931037.2931049.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE Transactions on Software Engineering 27 (2001) 99–123. doi:10.1109/32.908957.

[13] S. Hangal, M. S. Lam, Tracking down software bugs using automatic anomaly detection, in: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, Association for Computing Machinery, New York, NY, USA, 2002, pp. 291–301. doi:10.1145/581339.581377.

[14] B. Pytlik, M. Renieris, S. Krishnamurthi, S. P. Reiss, Automated fault localization using potential invariants, CoRR cs.SE/0310040 (2003). arXiv:preprintcs/0310040.

[15] S. K. Sahoo, J. Criswell, C. Geigle, V. Adve, Using likely invariants for automated software fault localization, in: Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems, 2013, pp. 139–152. doi:10.1145/2451116.2451131.

[16] M. A. Alipour, A. Groce, Extended program invariants: applications in testing and fault localization, in: Proceedings of the Ninth International Workshop on Dynamic Analysis, 2012, pp. 7–11. doi:10.1145/2338966.2336799.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. I. Jordan, Scalable statistical bug isolation, ACM SIGPLAN Notices 40 (2005) 15–26. doi:10.1145/1064978.1065014.

[18] C. Liu, X. Yan, L. Fei, J. Han, S. P. Midkiff, Sober: Statistical model-based bug localization, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13, Proceedings of 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, 2005, pp. 286–295. doi:10.1145/1081706.1081753.

[19] C. Liu, L. Fei, X. Yan, J. Han, S. P. Midkiff, Statistical debugging: A hypothesis testing-based approach, IEEE Transactions on Software Engineering 32 (2006) 831–848. doi:10.1109/TSE.2006.105.

[20] P. A. Nainar, T. Chen, J. Rosin, B. Liblit, Statistical debugging using compound boolean predicates, in: D. S. Rosenblum, S. G. Elbaum (Eds.), Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07, ACM Press, 2007, pp. 5–15. doi:10.1145/1273463.1273467.

[21] Z. You, Z. Qin, Z. Zheng, Statistical fault localization using execution sequence, in: 2012 International Conference on Machine Learning and Cybernetics, volume 3, IEEE, IEEE, 2012, pp. 899–905. doi:10.1109/icmlc.2012.6359473.

[22] V. Dallmeier, C. Lindig, A. Zeller, Lightweight defect localization for java, in: ECOOP 2005 - Object-Oriented Programming, Springer Berlin Heidelberg, 2005, pp. 528–550. doi:10.1007/11531142_23.

[23] H.-Y. Hsu, J. A. Jones, A. Orso, Rapid: Identifying bug signatures to support debugging activities, in: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE, IEEE, 2008, pp. 439–442. doi:10.1109/ase.2008.68.

[24] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases,

in: VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile, Morgan Kaufmann, 1994, pp. 487–499. URL: http://www.vldb.org/conf/1994/P487.PDF. doi:10.5555/645920.672836.

[25] C. Sun, S.-C. Khoo, Mining succinct predicated bug signatures, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013, ACM Press, 2013, pp. 576–586. doi:10.1145/2491411.2491449.

[26] J. Li, H. Li, L. Wong, J. Pei, G. Dong, Minimum description length principle: Generators are preferable to closed patterns., in: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1, volume 1 of *AAAI'06*, 2006, pp. 409–414.

[27] Z. Zuo, S.-C. Khoo, C. Sun, Efficient predicated bug signature mining via hierarchical instrumentation, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014, ACM Press, 2014, pp. 215–224. doi:10.1145/2610384.2610400.

[28] E. Bruneton, ASM 4.0 A Java bytecode engineering library, 2007.

[29] W. E. Wong, V. Debroy, R. Gao, Y. Li, The DStar method for effective software fault localization, IEEE Transactions on Reliability 63 (2014) 290–308. doi:10.1109/tr.2013.2285319.

[30] M. Renieres, S. Reiss, Fault localization with nearest neighbor queries, in: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings., ASE'03, IEEE Comput. Soc, 2003, pp. 30–39. doi:10.1109/ase.2003.1240292.

[31] H. Cleve, A. Zeller, Locating causes of program failures, in: Proceedings of the 27th international conference on Software engineering - ICSE '05, ICSE '05, ACM Press, 2005, pp. 342–351. doi:10.1145/1062455.1062522.

[32] P. A. Nainar, T. Chen, J. Rosin, B. Liblit, Statistical debugging using compound boolean predicates, in: D. S. Rosenblum, S. G. Elbaum (Eds.), Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07, ACM Press, 2007, pp. 5–15. doi:10.1145/1273463.1273467.

[33] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan, Soot, in: CASCON First Decade High Impact Papers on - CASCON '10, ACM Press, Mississauga, Ontario, Canada, 2010, p. 13. doi:10.1145/1925805.1925818.

[34] P. Lam, E. Bodden, O. Lhoták, L. Hendren, The soot framework for java program analysis: a retrospective, in: Cetus Users and Compiler Infastructure Workshop (CETUS 2011), volume 15, 2011, p. 35.

[35] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. Hoorn, A. Filieri, D. Lo, An evaluation of pure spectrum-based fault localization techniques for large-scale software systems, Software: Practice and Experience 49 (2019) 1197–1224. doi:10.1002/spe.2703.

[36] R. Just, D. Jalali, M. D. Ernst, Defects4j: a database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014, ACM Press, 2014, pp. 437–440. doi:10.1145/2610384.2628055.

[37] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, M. de Almeida Maia, Dissection of a bug dataset: Anatomy of 395 patches from defects4j, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 130–140. doi:10.1109/saner.2018.8330203.