Suspicious Types and Bad Neighborhoods: Filtering Spectra with Compiler Information

Leonhard Applis[1,2]
National University of Singapore
Singapore, Singapore
l.applis@nus.edu.sg

Matthías Páll Gissurarson[1]
Chalmers University of Technology
Gothenburg, Sweden
pallm@chalmers.se

Annibale Panichella
TU Delft
Delft, The Netherlands
a.panichella@tudelft.nl

***Abstract*—Spectrum-based fault localization and its formulas often struggle with large spectra containing many expressions irrelevant to the fault, which impacts its overall effectiveness. Spectra can inflate for large programs or on finer granularity, such as expression-level coverage from other languages like Haskell. To address this, we introduce 25 rules to filter the spectra based on type information, AST attributes, and test results. These aim to reduce the suspiciousness of innocent locations (bug-free expressions) and improve the performance of SBFL formulas w.r.t. `TOP50` and `TOP100` metrics. Our experiment, conducted on 11 Haskell programs, shows that individual filters significantly reduce spectra size, although some data points (faulty expressions) become unsolvable. By applying established SBFL formulas like Ochiai and Tarantula to these reduced spectra, we observe average improvements of up to 40% w.r.t. `TOP50` for individual *soft rules*, such as proximity to failure. Combining the best-performing filters yields improvements of 45.5% for Ochiai, 67.4% for DStar2, and 45.5% for Tarantula. The most effective filtering rules over all formulas captured proximity to failing expressions, usage of a non-unique type, and whether a failing test covered the expression. Our results suggest that simple, straightforward filters can produce substantial performance gains. We further identify 4 *uncovered* bugs originating from code generation (common in functional programming) and system tests, which can not be addressed purely by spectrum-based fault localization.**

***Index Terms*—Fault Localization, Functional Programming, Haskell**

I. Introduction

Spectrum-based fault-localization (SBFL) [1] is a widely-used technique to assist developers in locating bugs. It generates a spectrum of the program under test by tracking coverage for both failing and passing tests. Except for some machine-learning applications [2], these spectra are processed by a ranking formula, which estimates the likelihood of each location being faulty (called suspiciousness). There has been a rich body of research to improve this general framework: Test-Generation [3], [4] to get a richer spectrum, more specialized formulas [5], [6], or additional program information (e.g. the usage of entity relations [7] or proximity [8]).

Despite this progress, SBFL continues to face challenges [9], [10], usually related to the programs' size and its complexity. Some of the other techniques, such as test generation [3], have a hard time adjusting for this — the more complex a program gets, the harder test generation becomes [11].

With this work, we aim to improve SBFL without introducing additional techniques like test generation. We enrich the spectra at generation by adding information on the locations' type, as well as their AST position, allowing us to filter the spectra before applying ranking formulas. The basic idea is simple: If we are able to filter out *innocent* (non-buggy) expressions, then the final ranking can be significantly improved. Yet determining which program attributes are strongly related to faults is not straightforward. For instance, a complex type can both be the origin of issues (i.e., because they are hard to understand) or limit the possibilities for misuse (i.e., they are explicit and understandable). To explore this, our work designs a set of initial filtering rules and evaluates their efficiency for both spectrum reduction and the influence on the suspiciousness scores. To avoid bias and thoroughly investigate the effectiveness, we also invert individual rules to test the antithesis of our assumptions.

The implementation (and examples) from this work are provided in Haskell. Functional programming languages like Haskell have not been widely investigated in fault localization research. However, they present unique challenges for SBFL. For instance, Haskell's lazy evaluation paradigm can complicate mapping between tests and the expressions under test since the latter are evaluated only when necessary. Higher-order functions and sophisticated type systems introduce indirection, making locating faulty expressions harder. Despite these additional challenges, Haskell offers rich information through the Glasgow Haskell Compiler (GHC), which allows the extraction of the expression properties required for our filtering rules. Our research shows how SBFL may be modified for functional programming but the rules presented are transferable to imperative languages and can be re-implemented or extended for Java. At the same time, we shed some light on the applicability of initial SBFL approaches for functional programming, a field that is currently understudied.

A. Contributions

We enrich the normal approach to SBFL as seen in Figure 1. Our contributions can be summarized as follows:

1) Expanding spectra by introducing type information for each expression.

[1]Both authors contributed equally to this research.
[2]This work was initiated while the author was affiliated with TU Delft.

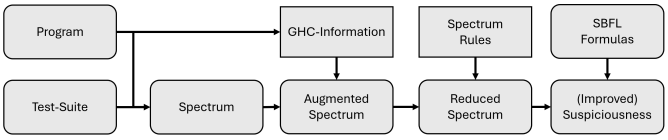

Fig. 1: Overview: Spectrum-Reduction Pipeline

2) A configurable set of predicates that filter spectra for type-, execution- and AST-attributes.
3) Evaluating SBFL with and without spectra-filters for 11 real-world Haskell programs.
4) Implementation of a expression-level spectrum generation add-on to Haskells' tasty framework.

We find that while initially SBFL is promising for functional programs, it is significantly improved by applying (single) filters by up to 42% w.r.t. `TOP50` scores. Combining predicates (filtering rules) *can* improve this further, yet the results are data point dependent and overfitting occurs.

B. Research Questions

To evaluate the spectra-reduction, we first establish the baseline performance of SBFL for the programs under test by applying common formulas from literature[1].

Based on the original (known) fault locations we are first interested in the *impact* of filters, in particular if they improve the ratio of $\frac{|faults|}{|expressions|}$:

RQ1: Effectiveness of Filters
Are AST- and Type-based rules an effective approach to reducing spectra while preserving faults?

While the filters might produce a preferable ratio of elements, this does not guarantee an overall improved performance; some formulas exploit specific attributes, which might be removed by filters loosing the most promising suggestions. As such, **RQ2** investigates the changes of formula-rankings when combined with a *single filter*:

RQ2: Impact of Filters for SBFL-Formulas
How is the individual performance of formulas impacted by AST- and Type-based spectrum filters?

Lastly, as we hope to see partial improvements in RQ1 and RQ2, we aim to combine rules in a way that noticeably improves the performance of formulas. By searching for a combination, we aim to also combine the partial results into an actionable outcome formulated in **RQ3**:

RQ3: Searching for (optimal) Rule Combinations
Can we identify a combination of rules that leads to a significant performance improvement for existing formulas?

[1]Based on Naish et al. [12] we implement a total of 8 formulas, covering all equivalence classes identified by their work. From `Optimal`,**`OptimalP`**,**`Tarantula`**, **`Ochiai`**, **`Dstar2`**, **`Dstar3`**, `Rogot1`, `Jaccard` and `Hamming`, we focus on the **bold** ones for their success on the data points and prominence within existing work.

```haskell
module Ex where

divs :: Int -> [Int]
divs n = go 2
  where
    go i | i == n = []
    go i = if d i
           then i : go (i + 1)
           else go (i + 1)
    d i = n `mod` i == 0

smallestDiv n = head (divs n)


prop_10 =
  divs 10 `has` [2, 5]
prop_15 =
  divs 15 `has` [3, 5]
prop_128 =
  divs 128 `has` [2, 4, 8, 16, 32, 64]
prop_8  =
  smallestDiv 8 == 2
prop_13 =
  smallestDiv 13 == 13
prop_evens n = n > 2 && even n ==>
  smallestDiv n == 2
prop_odds n = n > 2 && odd n ==>
  n `mod` (smallestDiv n) == 0
```

Fig. 2: A buggy program with a test suite. **`has`** checks that the list contains all the elements listed.

In summary, this research explores directions to design filters that improve performance of existing SBFL formulas.

C. Motivating Example

Consider the function and properties in Figure 2. The **`divs`** function returns the list of divisors of the given number, **`smallestDiv`** should return the smallest divisor. The properties in this figure specify both unit tests using concrete examples and two metamorphic tests, which check that transformations to the input lead to expected output.

The implementation in Figure 2 contains a bug: When called with a prime number it returns the empty list, where the **`smallestDiv`** expects a non-empty list. For prime numbers such as in **`prop_13`** and generated examples for **`prop_odds`**, the program crashes with an error: **`Prelude.head: empty list`**. This is due to an off-by-one error — **`divs`** does not include the number itself.

Running the test suite for the program in Figure 2 produces the spectrum in Table I. Standard spectra consist of the tests, their status, and the locations covered by each test. For this paper, we augment the spectra: We include the types of expressions, the framework of the test (properties, unit tests, golden tests), identifier names (whether the location possesses an identifier), as well as the number of calls for this location in each test. Our coverage is not *binary*, i.e. whether they were called by a test, but are fine-grained and track how often a

TABLE I: (Partial) spectrum for the code in Figure 2

name	type	result	`i`	`n`	`i == n`	`[]`	`d i`	`i: go (i + 1)`	`go (i + 1)`	`d i = ...`	`smallestDiv`	`divs n ...`
type	-	-	Int	Int	Bool	[Int]	Bool	[Int]	[Int]			
ident	-	-	i	n								
10	Unit	True	4	4	0	0	2	2	2	4	0	1
15	Unit	True	4	4	0	0	5	2	2	4	0	1
128	Unit	True	63	63	0	0	57	6	57	63	0	1
8	Unit	True	1	1	0	0	0	1	0	1	1	1
13	Unit	False	12	12	1	1	11	0	11	11	1	1
evens	QC	True	100	100	0	0	0	100	0	100	100	100
odds	QC	False	2	2	1	0	1	0	1	1	1	1

$$\textbf{Tarantula: } \frac{\frac{n_{i_f}}{n_{i_f}+n_{u_f}}}{\frac{n_{i_f}}{n_{i_f}+n_{u_f}}+\frac{n_{i_p}}{n_{i_p}+n_{u_p}}}$$

$$\textbf{Ochiai: } \frac{n_{i_f}}{\sqrt{(n_{i_f}+n_{u_f})(n_{i_f}+n_{i_p})}}$$

Fig. 3: Tarantula & Ochiai, drawn from [12]. n_{i_f} and n_{i_p} are the number of times the expression e is involved in a failing or passing test respectively, while n_{u_f} and n_{u_p} is how many times the expression was uninvolved.

Location	Which is	Ochiai	Tarantula
6:12-17	**`i == n`**	1.0	0.625
6:21-22	**`[]`**	0.707	1.0
12:1-29	**`smallestDiv`**	0.707	0.714
7:15-17	**`d i`**	0.632	0.625
9:18-27	**`go (i + 1)`**	0.632	0.625
10:5-24	**`d i = ...`**	0.534	0.5

Fig. 4: `TOP5` suspiciousness scores from classic SBFL formulas, with the bug location in bold

location was called. The notation `6:12-17` in Table I and Figure 4 stands for line 6 columns 12 to 17.

Using the formulas detailed in Figure 3 on the spectrum, we can score the locations as shown in Figure 4.

The results highlight three possibilities of fixing the bug: ① Fix the off–by–one error in line 6 by correcting the comparison, ② provide a default empty list, or ③ replace the definition of **`smallestDiv`** to handle cases when **`divs`** returns an empty list. This short example program spans only 35 locations, of which 5 are relevant to find the bug. Real programs have a lot more locations. Two of the considered programs Pandoc and Duckling, have 91k and 277k locations respectively, accounting only for those involved in the test suite (see Table III). In comparison, their faults are *the needle in the haystack*, ranging from 6 to (at most) 72 expressions, touched by only a handful failing tests. This forms a known limitation for traditional SBFL formulas [10], which unfortunately affects application to real-world bugs. By *filtering* the spectrum, we can focus our attention on only the locations we are interested in. In Section III-B we introduce a rule-base system that allows users to filter a spectrum before analysis (i.e. the ranking by formula application), users can encode their intuitions on the "usual suspects", and increase both localization and runtime performance of SBFL techniques. The rules in this work express our intuitions, and even simple filters can proof useful: By applying the rule `rIsIdentifier == 0.0`, we reduce the spectrum to 26 locations that do not specify an identifier. This still results in the bug locations above, with the same Ochiai scores. If we want to focus only on AST-leaves, we apply the rule `rASTLeaf == 0`, and reduce the spectrum to only 14 locations, while still keeping the `6:21-22` location. A further reduction is possible by combining rules: As an example, `(not (rIsIdentifier == 0)) && (rASTLeaf == 0)` reduces the spectrum to only 5 locations, keeping the fault at `6:21-22`. But which attributes should we focus on? In the rest of this work, we take large open-source Haskell programs to provide detailed analysis and shed light on which attributes are important.

II. Background and Related Work

a) Spectrum-based Fault Localization: Spectrum-based fault localization (SBFL) was developed as a technique to cover well-testable issues related to the year 2000 problem [1], and is considered one of the most prominent due to its efficiency and effectiveness [10]. After defining a failing test that triggers the Y2K problem of an application, the program tests were executed in order, and their coverage was recorded. Under the assumption that there are (passing) tests covering expected behavior, the issue must originate in statements covered by failing tests without being in passing tests. This approach formed the core of modern SBFL: from the initial concept of intersection, techniques emerged that use formulas [13], [14], [5], [6] to assign suspiciousness scores to different parts of the program.

Many refinements have been proposed: promising work revolves around the introduction of new AST elements and program states [8], the application of mutation [15], [16], [17], [18], meta- or machine learning approaches [19], [20], [21], or the filtering of tests and statements [22], [23], [24].

An important piece of work we draw from is Naish et al. [12], which discusses mathematical attributes of spectrum-based formulas. In addition to introducing two new formulas, they prove that some formulas must result in the same ranking (equivalence classes). Within this work, we implement at least one formula from each identified equivalence class.

b) Li et al.: Comparable work on spectrum-based fault localization for Haskell originates from Li et al. [18]. They collect open-source bugs and apply existing SBFL formulas

to expression-level spectra. To improve generalization and introduce an ML approach, the programs were mutated to extend their data set. Although they publish the dataset which we incorporate, the original code is unavailable. Li et al. have similar goals in introducing SBFL for Haskell, but the details differ: On a more fundamental level, our spectra extend previous work with unique attributes of types, tests, and identifiers. We introduce rules that extend the existing literature to capture more concepts than SBFL formulas currently can. Their approach includes data augmentation, which is an venue to synthesize the efforts of both works in future research.

c) SBFL Tie breaking: Research that is close to our work is the field of *Tie-Breaking* SBFL Formulas [25], [10]. Similarly to our motivation, SBFL in Java also faces rankings with identical execution patterns or suspiciousnesses.

Tie-breaking mechanisms often emphasize approaches similar to our rules: Sarhan et al. [26] utilize *method-call* frequency to break ties, treating more frequently called locations more suspicious, Beszedes et al. [27] exploit chains in function calls to identify prominent candidates and Wen et al. [28] utilize the commit history to rank recently changed locations higher.

Our work differs in abstraction from individual rules by providing multiple configurable options and their combinations. The provided richer spectra can also be picked up by other frameworks, e.g. as input for machine learning models.

d) Other Fault Localization efforts for functional programming: Fault localization in a functional setting has been explored in Liquid Haskell [29], using `refinement types`, a type system augmented with logical predicates. Tondwalkar et al. collect constraints and localize faults by mapping a minimal set of atomic unsatisfiable type constraints to likely bug locations. The work relies on a more powerful type system than Haskell has, namely liquid types, which localizes (and repairs) errors on the type level. The Liquid Haskell approach requires precise modeling of the expected system-behavior at the type level, which often means giving up type-inference In this work, we target programs with existing test suites, and enable developers to get more out of previous testing efforts. Using liquid types, a form of test generation can form supplementary work similar to test generation efforts in program repair.

III. Implementation & Experiment Setup

A. Spectrum Generation

We introduce a TASTY-SPECTRUM package[2] which adds an *ingredient* to the Tasty test framework that captures coverage when tests are run and generates a spectrum. Tasty-Ingredients are a modular way to implement plugins for Tasty adding additional behavior around tests such as re-running, timeouts, or, in this case, data extraction.

To generate spectra, we use the instrumentation provided by Haskell Program Coverage (HPC) and programs compiled with the `-fhpc` flag. This generates `.mix` files that allow HPC to connect the indices it produces to the source locations in the modules. Our implementation also includes a *GHC source plugin*, which integrates with the compiler and extracts type and identifier information from modules during compilation and generates `.types` files.

GHC Source Plugins: GHC allows users to define *source plugins*, which are run at the end of various stages of compilation, including parsing, type-checking, and renaming. These plugins allow users to modify and interact with the source code after each stage. In the TASTY-SPECTRUM package, we define a plugin that operates at the end of the type-checking stage, which traverses the type-checked expressions, and notes their types in a `.types` file. The `.types` files are saved alongside the `.mix` files and later combined with the `.mix` information during spectrum generation.

Haskell Program Coverage (HPC): HPC instrumentation is integrated into GHC, and is based on maintaining an array that counts executions for each source location (which corresponds to expressions) in the module during runtime. Whenever an expression is evaluated, this triggers a "bump" in the array, allowing HPC to track the number of times each expression was evaluated in the module. This array allows access, manipulation and re-initialization at runtime.

Spectra are generated by running the test suite. As the code has been compiled with the `-fhpc` flag, the RTS will keep the TIX array in memory. Before running each test, we reset the HPC state. After each test, we read the current state of HPC, and track which expressions were evaluated.

After running all tests, the TIX array for each test is combined with the module structure from the MIX files and the type/identifier information from the `.types` files to produce a *type-augmented spectrum* as a `.csv` file. To reduce the size of the spectrum and focus only on relevant data, we exclude locations that were not involved in any tests, i.e., those that have zero evaluations across the entire test suite.

B. Rules

We present the summary of rules in Table II. As outlined in the introduction, none of these rules *are proven to be good*; instead, they form a starting point for filtering and might prove less applicable or only useful for individual data points. The thresholds presented form the result of an exploration via trial-error tuning. The thresholds have been adjusted if a rule either did not filter any entries or filtered out all locations or faults across all programs under analysis. A special focus was on the rules' complementary aspect — we aim to exploit multiple dimensions of a single program. As such, we did not want to cover all possible AST positions in a broad range but rather hoped to find meaningful intersections of test, type, and execution patterns that benefit the overall approach.

We admit the rules constitute only a starting point, and a detailed analysis (e.g., interpreting thresholds as an optimization task) is desirable. We consider this to be promising future work, but expect it to be most fruitful on a per-project basis: As seen in the following section III-C, the projects utilize different features in varying capacities, making a generalization

[2]https://github.com/Tritlo/TastySpectrum

TABLE II: Code & Rule Overview

Code	Filter	Explanation	Rationale
LEAF	rASTLeaf == 0	Filters for locations that are AST-leaves	AST-leaves contain a lot of potentially interesting elements, such as primitive values or inputs for logical conditions
NLEAF	rASTLeaf > 0	Filters for non-AST-leaves	
ID	rIsIdentifier == 1	Filters for expressions that are an identifier	Catches cases of using the wrong identifier with the correct type.
HO	rTypeOrder >= 1	Covers types of functions, i.e. the expression is not a variable	Higher-order functions are a known source for complexity, e.g. `fold`, `map` or `traverse`
UN	rTypeArity <= 1	Arity expresses the number of arguments to a function. This rules covers non-functions and functions with 1 argument.	
HA	rTypeArity >= 3	Filters for functions that have 3 or more arguments.	Complex functions with many arguments might appear more often in faults. Famous Haskell troublemakers, like `foldr`, take 3 arguments.
ST	rTypeLength <= 5	Filters for types whose string-representation is shorter than 5 characters.	We consider type-length as a proxy for type complexity. Types can be complex for a variety of reasons (Order, Arity, etc.) but all commonly result in longer type signatures.
LT	rTypeLength >= 10	Only locations whose type when converted to text is longer than 10 characters	
VLT	rTypeLength >= 25	Only locations whose type when converted to text is longer than 25 characters	
VNF	rDistToFailure <= 2	rDistToFailure covers the length of the *shortest path* in the AST to a node touched by a failing test.	Capture proximity to test-failures. Not dependent on test-type.
NF	rDistToFailure <= 4	Closer than 4 AST edges towards a node touched by a test-failure.	Softer version of VNF.
IDFQ	rNumIDFails >= 3	Expressions whose identifiers have been executed in 3 or more failing tests.	Poor variable assignment can chain to a more common re-usage of poor variables.
UID	rNumIDFails == 1	Expressions whose identifier only appears once in a test-failure.	Reverse of IDFQ - we might consider often-used identifiers more innocent, and instead aim for unique ones.
UT	rNumTypeFails == 1	Expressions whose type-signature is unique within test-failures	Commonly used types might be less error prone - and unique ones more suspicious.
SID	rNumTypeFails >= 2	Expressions whose type-signature was at least twice within test-failures	Negation of UT
CID	rNumTypeFails >= 5	Expressions whose type-signature appears in 5 or more failing tests	Due to diversity in types, this should lead to only primitive, more common data-types.
HFF	rTFailFreq >= 10	Expressions who have been executed more than 10 times over test failures.	Captures execution patterns beyond the sum of binary coverage.
HPF	rTPassFreq >= 25	Expressions who have been executed more than 25 times in passing tests.	
DFP	rTFailFreqDiffParent >= 2	Filter for AST-nodes whose execution-frequency is different than its parents in failing tests.	Will be triggered by conditionals that lead to one failing and one or more passing branches.
CSTF	rNumSubTypeFails >2	Only expressions that handle a non-unique subtype, e.g. functions that take strings as arguments.	
USTF	rNumSubTypeFails <= 1	Uniquely failing sub-types and expressions that don't take arguments	
TF	rTFail >= 1	Filter for expressions that are touched by at least one failing test.	
UNF, PRF, GF	rUnitFail rPropFail rGoldenFail >= 1	Locations that have been touched by at least one failing test of a specific test-framework	

challenging. We hope that the results presented in this work provide an educated perspective on transferring spectra filters to other languages and frameworks.

C. Data Collection

We draw data from two Haskell fault datasets, `HasBugs` [30] and `HaFla` [18]. Both datasets provide a similar granularity of faults originating from projects with known bugs (based on issues and pull requests) whose fault-fixing commits include a test. The tests were extracted to produce a *faulty but tested* version with a failing test suite. We determine faulty expressions as all expressions that are completely within faulty lines, extracted from the git-difference.

A subset of the data was chosen to produce the spectra that met the required versions of Cabal, tasty (>v1.0), and GHC (>= 8.6). Other limitations excluded projects like PureScript (many of the tests run against compiled JavaScript) or Cabal (all bug-asserting tests are package-level tests outside the tasty

test suite). This results in a total of 11 programs[3] from 3 projects - **Pandoc**, **Duckling** and an **HLS**-plugin. An overview of the data points used is presented in table III.

Comparison with Defects4J - Comparing spectra between paradigms is challenging, but to approximate, we consult data from Defects4J [31]. We derive data from a public repository shared by René Just[4] that provides statistics from applying GZoltar [32] to a subset of 395 bugs from Defects4J.

The Defects4J bugs inspected have a mean *Source lines of code*(SLOC)[33][5] of 57.7k and a median of 62.5k. The mean number of tests in Defects4J is 1439 (median of 202), and with an average of 2 failing tests. Under the assumption that most of the SLOCs represent line-level statements, the resulting spectra will have a comparable number of elements. We approximate the faulty SLOC for Defects4J as an average of 2.56 based on the lines removed by the patch. We conclude that the programs and bugs used in this work are comparable in size to the averages of Defects4J, but admittedly Defects4J has a significantly higher number of datapoints.

D. Experimental Setup

Based on the fault-fixing commits of a data point, we revert the source code patch while keeping the changes to the test code, observing a test failure during `cabal test`. At this stage, we also distinguish *noisy* test failures as mentioned in Table III, marking tests that fail before and after the changes as *noisy*(other works refer to such tests as Fail-2-Fail (F2F)). Next, the cabal file is altered to include spectrum generation and coverage, following the description in Section III-A

a) Considered Metrics: The primary metric considered for ranking the expressions is `TOP-X` [34]. Within `TopX`, the recommended elements are sorted by their *suspiciousness*, and the *correct* classifications (truly faulty expressions) within the first `X` are counted. For this work, after seeing the results of initial rankings, we opted for `TOP50`. The `TOP10` was *too difficult* for most data points, and the `TOP100` did follow the `TOP50` with few exceptions.

Another common metric is `EXAM` [35], assuming that the user follows every recommendation in order until the real fault(s) are fixed. The index of the first correct fault is used to calculate the ratio of the inspected (total) program, with the exam score expressing *how many locations can be skipped when following the recommendations?* The EXAM score is proportional to the *mean reciprocal rank*, another metric commonly reported for FL. For this work, we discarded MRR and EXAM, as we work with different granularity due to our expression level spectrum: when introduced in 2003, EXAM was targeting block-level spectra, but the sheer difference in the quantity of (mostly benign) expressions would draw a highly beneficial picture of our approach. Therefore, for ranking evaluations, we focus on the `TOP-X` metrics [9].

[3]9 from HasBugs, two from HaFla

[4]https://bitbucket.org/rjust/fault-localization-data/src/master/

[5]SLOC are lines of code, after removing white space, comments and other *non-functional* elements.

b) RQ1: We answer RQ1 by applying each rule to each spectrum and report the mean reduction in statements as well as the mean delta in $\frac{|faults|}{|statements|}$. Each rule configuration is given a simple, short code name. *Very Long Type* becomes `VLT`, etc. The rules, their codes, and corresponding expressions are shown in Table II.

c) RQ2: is explored by applying all formulas to the reduced spectra and comparing their `TOP-X` scores with the respective baseline (i.e., the un-changed spectra of the data points/projects under analysis).

d) RQ3: To find possible configurations, we draw the most promising rules from `RQ1` and `RQ2`, respectively, and form the subsets of combinations. Due to computational considerations and an expected lower payoff with growing rules, we consider only combinations of 2 and 3 rules. The rules are chosen based on their success—we selected 9 of the initial 25 rules for pairing because they improved formulas over the baseline. The triples are made of the seven formulas of *successful* rule-pairs. This results in a total of $\binom{25}{1}+\binom{9}{2}+\binom{7}{3} = 96$ rule-configurations considered.

We re-apply the procedures from `RQ1` (comparison of reduction) and `RQ2` (performance of formulas) and report the average differences to the individual baselines.

IV. Results

a) Attributes of Spectra: The created spectra range in size from 25Kb (`HLS`), 200 MB (`duckling`) to up to 500 MB (`Pandoc`). Spectrum generation is not a costly addition to the runtime of tests, but compilation time of projects is longer because the `-fhpc` flag (instrumentation for program coverage) is required.

When implementing and controlling the rules, we noticed that a filter for `rTestFailure >= 1` lead to some data points becoming unsolvable, i.e. there were no faults touched by failing tests. This is the case for `duckling-4cfe88ea`, `duckling-ea8a4f6d`, `duckling-1dac46a8` and `pandoc-4`.

The authors double-checked the test suite and, for duckling, the correct (and expected) corpus tests were failing. The tests do not run against the *original source*, but instead generated code. Arguably, the generated code is faulty, but not the origin of the issue as it was fixed in the commit. These unique cases are between other data points, e.g. `duckling-328e59eb` which has faults covered by failing tests. For `pandoc-4` there are faulty locations on a *reader* that need changes in the data format. The relevant `golden test` run with a compiled binary of `pandoc` (unlike the other `pandoc` data points) which is invoked by tasty, and not collected by project coverage. Thus, we have a failing test suite, but the *touched* expressions originate only from noisy test failures.

The existence of faults that are *not directly covered* poses a challenge for this work and a under-represented aspect of fault localization. Stemming from real projects, the tests are realistic and express community efforts. Although tests cover bugs *semantically*, it does not cover the faulty code and

TABLE III: Overview of the used data points

Data Point	Issue	Faulty LOC	Faulty Locations	Total Locations	Failing Tests	Noisy Test Failures	Total Tests
pandoc-3be256efb	Wrong application of 'Big Note' highlighting when converting to LaTeX. Reordering necessary.	1	6	88k	6	0	3254
pandoc-4	Failure converting combined code and bold text to LaTeX.	3	12	91k	1	1	3056
pandoc-5	Misinterpretation of code blocks when converting to ROFF MS. Requires escaping.	1	8	61k	2	6	2400
pandoc-6	Wrongly converting code blocks starting with (1) into enumerations.	5	39	59k	10	13	2365
pandoc-7	Empty multi-cells not picked up when reading LaTeX.	27	72	61k	3	7	2415
hls-2	Issue accounting for relative location "./" instead of "."	2	15	269	1	0	6
hls-afac9b18	HLS-Plugins can reformat code, Stylish Haskell removed the last line regardless of whether it had content.	1	17	122	2	0	13
duckling-ea8a4f6d	Wrong pronomina for German million. Regex adjustment.	1	5	288k	1	0	364
duckling-4cfe88ea	Missing combined durations cases (e.g "2 hours and 20 minutes")	18	4	260k	1	1	342
duckling-28ddc3bf	Wrong parsing of `1.000,00` for Dutch.	1	5	299k	1	0	346
duckling-328e59eb	Missing cases for weights (and combinators) in Portuguese language.	19	26	277k	1	1	360

require new spectrum techniques. To some extent, these tests are juxtaposed to *automatically generated* tests, which cover code behavior without necessarily capturing the semantics of faults [36], [37], [38]. Due to the common usage of Haskell for domain-specific languages, parsers, and code generation tooling, we expect these types of faults to be more common in functional paradigms than in other languages. This constitutes an instrumentation shortcoming when tests work against any kind of artifact outside of the initial code base — Java will face similar issues in system tests, e.g. within a micro-service architecture and remote actors.

b) Single Filters and Shrinkage: An overview of shrinkage is presented in fig. 5, with rules grouped by their domain. We observe that rules around *Test Execution* (i.e. unit test failures, golden test failures, etc.) give the average best reduction in filter size - keeping most faults while removing most unrelated locations. The codes reduce spectra, but do (on average) not favorably impact the ratio of faults to entries, as indicated by the linear interpolation.

Over all *code-types* we see that there are *softer* rules and *harder* rules, respective to how much of the spectrum they filter. Many of the stronger filters rendered one or more data-points unsolvable by removing all true faults from the spectrum. This also the major set-back for the *Test Execution* filters: Depending on the test suite, a filter for failing golden tests (which has a very good ratio of reduction) will make issues tested by a property unsolvable. As such, the *Test Execution* filters perform well, but must be used either with some form of oracle, or by an educated user.

We'd like to highlight some filters in table IV: First, we see with `TF` and `UNF` two codes that strongly reduce the spectra, but remove all faults from 4 and 5 data points. The table shows by how much these actually shrink spectra - the overall ratio of faults to entries reaches 1%, making even *guessing* have an average `TOP50` of 2. This would make utilizing such filters a *dream scenario* and as such, these filters should be employed.

`LEAF` is an interesting case, as it tells us by itself a lot about our program: 63.6% of our nodes are not AST-leaves, but 75.4% of our faults are. As such, reducing by leaves should benefit most formulas while keeping all programs solvable. Next, we see with `NF` (proximity to failure ¡= 4) *proxies* `TF` by being a generally softer filter, resulting in very little reduction.

`CID`, `UT`, `LT` and `VLT` mean to be examples of the code type *Type Frequency*. They remove about as many entries as faults, and stricter rules produce unsolvable data points.

TABLE IV: Excerpt: Single Filter Spectra Shrinkage

Code	Entry Reduction	Fault Reduction	$\frac{\lvert faults \rvert}{\lvert entries \rvert}$	Points with 0 Faults
TF	99.2%	32.5%	0.01	4
UNF	99.1%	25.6%	0.01	5
LEAF	63.6%	25.6%	¡0.01	0
NF	1.3%	2.3%	¡0.01	0
CID	42.0%	41.5%	¡0.01	2
UT	98.4%	95.3%	¡0.01	6
LT	65.7%	49.0%	¡0.01	1
VLT	93.3%	71.6%	¡0.01	6

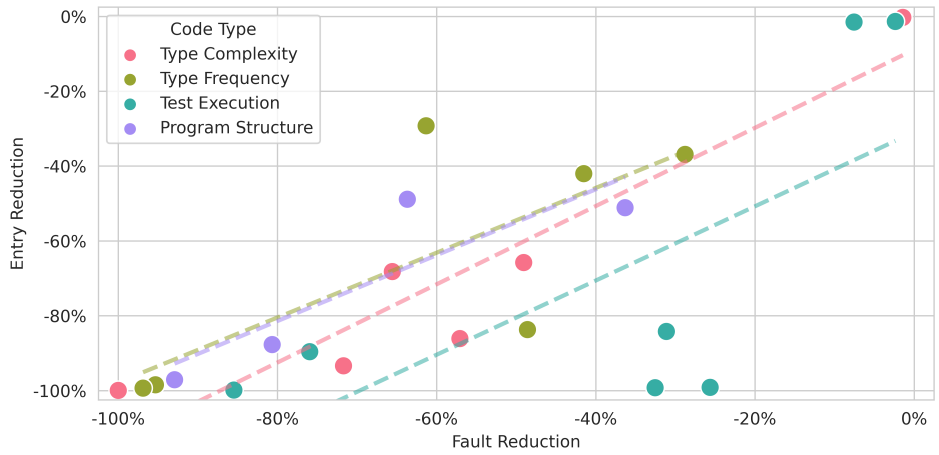

Fig. 5: Overview of Shrinkage by Code - Grouping

TABLE V: Average TOP50 improvement for single rule filters

OptimalP	Tarantula	Ochiai	DStar2	DStar3
BASE 4.0	BASE 2.8	BASE 4.7	BASE 4.2	BASE 4.3
NF 4.1 (+2.5%)	CID 2.9 (+3.5%)	SID 5.0 (+5.8%)	SID 5.1 (+21.4%)	-
SID 4.1 (+2.5%)	ST 3.4 (+21.4%)	UNF 5.6 (+17.5%)	NF 5.5 (+30.9%)	CID 4.8 (+11.6%)
UNF 5.7 (+42.5%)	NF 4.1 (+46.4%)	NF 6.4 (+34.6%)	UNF 6.0 (+42.8%)	UNF 5.7 (+32.5%)

RQ1: Shrinkage of Filters

The most preferable reductions are done by filters for *Test Types*, e.g. unit or golden tests. These require an educated user to not render data-points unsolvable. Only *soft* filters like AST-leaves or proximity to failure preserve faults in all data points, at the cost of lower reduction.

c) Filters and Formulas: Table V shows the codes that lead to the highest improvement for the most promising formulas. Single codes also improved other formulas like Rogot and Jaccard, but for brevity we focus on those that had the highest initial (and final) TOP50 score: OptimalP, Tarantula, Ochiai, DStar2 and Dstar3. The greatest improvement of 42.8% was achieved for DStar2 when filtering for expressions executed by at least one failing unit test. The best overall performance for single rules was achieved by Ochiai and a filter for NF, i.e. nodes that were at most 4 AST-edges away from a node touched by a test-failure. With this configuration, a final TOP50 of 6.4 was achieved (+46.1%).

In total, there were 9 codes that lead to improvement: 'SID', 'VLT', 'ST', 'UN', 'TF', 'NF', 'CID', 'CSTF', and 'UNF'. These fall into two categories: type- and test-based. We see rules on sharing types (SID, CID, CSFT), type complexity (VLT,ST,UN) and test execution correlation (NF, TF, UNF). These 9 codes were chosen for further combination.

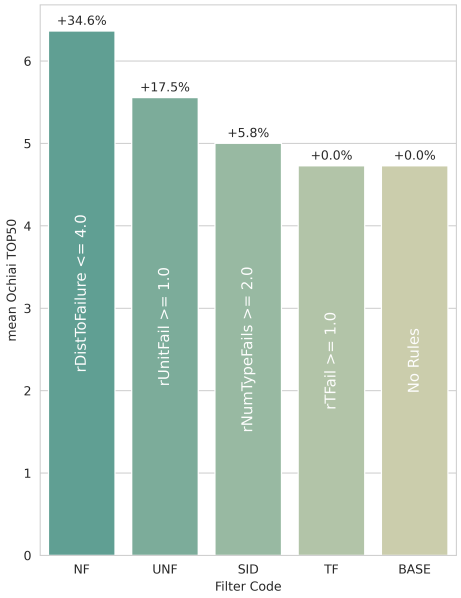

Fig. 6: Best performing codes for Ochiai

One notable fact is that the filters that lead to the highest improvements in TOP50 did not necessarily align with the highest shrinkage. A prominent code is NF which reduces spectra only by a few percent. We expect this to be a sweet spot between improving some, while not interfering with other, data points of the set. Filters for test-type improve the applicable programs significantly, but they also render some unsolvable. On average, for the single rule filters, this *hard* approach seems not beneficial. *Soft* filters like NF, SID and CID reduce the spectra evenly while keeping nearly all data points solvable.

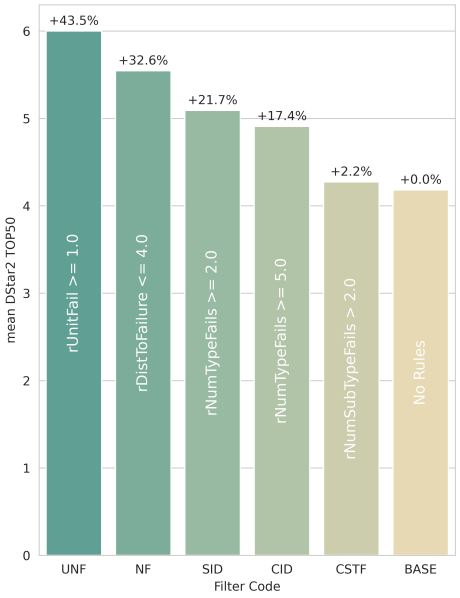

Fig. 7: Best performing codes for DStar2

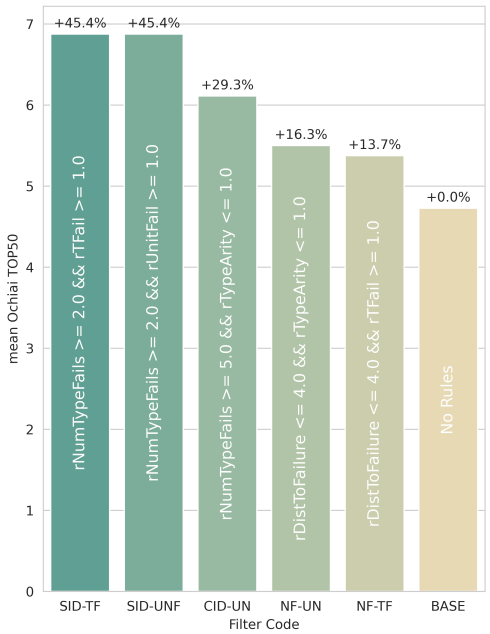

Fig. 8: Best performing multi-codes for Ochiai

RQ2: Effects of Filters for `TOP50` Scores

Some filters improve formula-`TOP50`-performance drastically: Filtering for proximity to failure improves Tarantula by 46%, Ochiai by 42% and DStar2 by 31%. The other prominent code filters for unit test failure, improving DStar by 43% and Ochiai by 17.5%. The effectiveness of filters does not align with the general reduction of the spectra and the ratio of faults to entries it leaves.

d) Combining Filters: When applying pairs of filters, we see another increment in performance. An *successful* example is seen in fig. 8: The strongest improvement is achieved by a combination of (unit) test failure (UNF / TF) and the type of a location being used non-uniquely (SID). Compared to *just* applying UNF, this combination brings another 19% improvement on overall `TOP50`, resulting in the best configurations being 45.4% better than baseline performance (which is 10.8% better than the best single-code filter for Ochiai, `NF` in fig. 6). As discussed for shrinkage, these *top performers* are assume that the test-framework is known and might not be usable for other programs. Still, we also see *soft* rules (CID-UN, NF-UN) that give improvements while being test-framework-agnostic.

We see further improvement on a *three code* filters, such as `NF-TF-UNF` reaches a `TOP50` of 8.5 for Tarantula (+200% to baseline) and DStar3 (+99% to baseline), and bring less common formulas up to the same performance (Jaccard, a primitive formula, also reaches `TOP50` of 8.5). This can be considered over-fitting: Within the data points, they have uneven numbers of faults. These filters that specialize on the test-types remove all noise for the fault-rich data points (pandoc-6, pandoc-7) and results in overall high average `TOP50`, despite not performing better for most data points. The occurrence of `UNF` (unit test failure) which should be a subset of `TF` (test failure) in combination indicates a form of over-fitting.

RQ3: Combining Filters

Pairs of filters can further improve performance, e.g. by 10.8% for Ochiai (to 45.4% above baseline). The highest performance increases are achieved by filters that contain test-type restrictions. We see less strong, but overall improving combinations of *soft* filters filtering by type complexity. Once combining three codes, the filters over-fit on a few fault rich data points.

V. Discussion

a) Should we use test-based Filters: A set of filters that *can* lead to great results are focused on test type: (Unit) Test failures can both heavily shrink the spectra and assist the formulas by reducing noise. These average results draw from its population: For some (fault-rich) data points, UNF is a silver bullet, yet for others, it removes all valid locations.

These test-type rules are also conceptually different from the other rules; every program will have AST leaves and more or less complex types, but not every program utilizes properties. As such, for researchers or uncertain errors, we recommend proxying these *hard* rules with the *soft* rule NF. Its proximity to failure left all data points solvable and still improved the average performance of Ochiai by 34.6%.

An actual maintainer who is debugging an issue is in a much better position. Once the test suite is fully known and

the tester knows the tests they provided, the over-fitting seen in RQ3 can be avoided. The resulting filters not only reduce noise and double formula performance but also speed up the whole ranking process.

b) Dealing with uncovered Faults: We have seen a total of 4 *uncovered* bugs, where we observe both a semantically correct test failure and there is a connected piece of faulty code, yet the faulty location is not covered by its test. This happens due to system-level tests in the unit test suite or due to code generation. While code generation is more prominent in FP, Java has similar problems for system-level tests, e.g., in micro-service architectures [39]. Due to (all) formulas requiring a test failure to rank locations higher than zero, unassisted SBFL cannot locate these bugs. We suggest to utilizing `Test Generation`[40] and in particular `Test Carving` [41], [42]. Such approaches can generate tests exploiting the existing *miss-locating* test as an oracle, and might benefit from the enriched spectra as well: In the case of code generation, there is direct relation between structure (e.g., code layout), types (which type results in which generated type), and naming (generated code often follows framework-specific conventions). This might be the best pursuit in Java, as a rich body of existing literature and tools exist.

c) Chances for imperative Programming: Haskell is not exactly a wonderland of fault localization — but we believe that some of the rules implemented in this work are (easily) transferable to Java and C. Both languages utilize types, and we have seen that type length is a decent proxy for type complexity. While more nuanced, looking for sub-types, functions, polymorphism, or generics might not be necessary given the results obtained in our work.

Unit tests form the focus of fault-localization for e.g., Java, but detecting other frameworks can be achieved programmatically (by looking for framework keywords and annotations) or heuristically by the naming conventions. Java unit tests could also provide other information to consider, for example, if pre- and post-conditions are specified (JUnits `@BeforeEach` and `@BeforeAll`). In a similar fashion, Java has an AST too, yet access to it requires meta-programming libraries or other forms of additional instrumentation. From this work, we see that the most relevant AST attribute is `NF`: proximity to failure. Our intuitive guess is that Java would benefit from similar simple heuristics, e.g. *is in failing class* or *is in failing module*. Work on fault localization that tried to identify faults on a file level [43] might be sources of such filters.

VI. Threats to Validity

External Validity: Our dataset/benchmark comprises 11 real-world Haskell projects maintained by different developer teams from different application domains. Nonetheless, the results might not generalize to other Haskell projects or programming languages. However, the rule set we have developed is based on general programming concepts and could be applied to other languages, including imperative languages like Java. We have elaborated on this aspect already in Section V. Besides, as discussed in Section III-C, the programs and bugs used in this work are comparable in size to Defects4J, which is a widely used Java benchmark for fault localization research.

Construct Validity: The implementation of the rules and generation of the spectra are based on the HPC tool, which is a standard tool for Haskell code coverage. The rules could be further refined or extended to improve the fault localization performance and capture different aspects of the code. Another potential threat is related to the possible bias of our conclusions with regard to the choice of SBFL formulas. To address this, we selected multiple formulas (e.g., DStar, Tarantula, Ochai) that are widely used in the literature. Our results show that the rules can improve the performance of these formulas, although with differences in the extent of improvement. To analyze our results, we use well-established performance metrics, such as `TOP-X` and `EXAM`, which are widely used in the literature. However, these metrics might not capture all aspects of the fault localization performance.

VII. Conclusion

This paper aims to improve spectrum-based fault localization by filtering the spectra using AST- and type-based rules. To achieve this, we implemented a Tasty ingredient that allows the generation of spectra with expression-level granularity, including additional information on types and identifiers. Making use of the richer information, we implemented *rules* that capture the complexity of types, AST structure, or identifiers and applied them to filter a total of 11 real-world programs. The results indicate that even single rules can improve formulas like Ochiai by 42% by reducing irrelevant locations. Combining rules leads to diminishing returns and faces the possibility of over-fitting and should only be applied by educated users. Our exploration further uncovered unique kinds of failures: faults that were not covered by failing tests. Further research should focus on transferring these filters toward imperative languages, like Java, and automatically fine-tuning rules on a per-project basis.

Acknowledgements

We thank David Sands and Matthew Sottile for their input on spectra, and Alejandro Russo for input on FP-specific adaptions. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knuth and Alice Wallenberg Foundation. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001.

References

[1] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, 1997, pp. 432–449.

[2] M. Golagha, A. Pretschner, and L. C. Briand, "Can we predict the quality of spectrum-based fault localization?" in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 4–15.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 49–60.

[4] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 257–267.

[5] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[6] D. Lo, L. Jiang, A. Budi *et al.*, "Comprehensive evaluation of association measures for fault localization," in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–10.

[7] H. He, J. Ren, G. Zhao, and H. He, "Enhancing spectrum-based fault localization using fault influence propagation," *IEEE Access*, vol. 8, pp. 18 497–18 513, 2020.

[8] Z. Zhang, W. K. Chan, T. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 43–52.

[9] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[10] Q. I. Sarhan and A. Beszédes, "A survey of challenges in spectrum-based software fault localization," *IEEE Access*, vol. 10, pp. 10 618–10 639, 2022.

[11] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017.

[12] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[13] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 467–477. [Online]. Available: https://doi.org/10.1145/581339.581397

[14] J. A. Jones and 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*, 2005, pp. 273–282.

[15] M. Papadakis and Y. Le Traon, "Effective fault localization via mutation analysis: A selective mutation approach," in *Proceedings of the 29th annual ACM symposium on applied computing*, 2014, pp. 1293–1300.

[16] ——, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[17] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 153–162.

[18] F. Li, M. Wang, and D. Hao, "Bridging the gap between different programming paradigms in coverage-based fault localization," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, 2022, pp. 75–84.

[19] L. C. Ascari, L. Y. Araki, A. R. Pozo, and S. R. Vergilio, "Exploring machine learning techniques for fault localization," in *2009 10th Latin American Test Workshop*. IEEE, 2009, pp. 1–6.

[20] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.

[21] M. Gao, P. Li, C. Chen, and Y. Jiang, "Research on software multiple fault localization method based on machine learning," in *MATEC web of conferences*, vol. 232. EDP Sciences, 2018, p. 01060.

[22] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. J. van Gemund, "Prioritizing tests for software fault localization," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 42–51.

[23] G. Dandan, W. Tiantian, S. Xiaohong, and M. Peijun, "A test-suite reduction approach to improving fault-localization effectiveness," *Computer Languages, Systems & Structures*, vol. 39, no. 3, pp. 95–108, 2013.

[24] L. Vidács, Á. Beszédes, D. Tengeri, I. Siket, and T. Gyimóthy, "Test suite reduction for fault detection and localization: A combined approach," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 204–213.

[25] X. Xu, V. Debroy, W. Eric Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 06, pp. 803–827, 2011.

[26] Q. I. Sarhan, B. Vancsics, and Á. Beszédes, "Method calls frequency-based tie-breaking strategy for software fault localization," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021, pp. 103–113.

[27] A. Beszédes, F. Horváth, M. Di Penta, and T. Gyimóthy, "Leveraging contextual information from function call chains to improve fault localization," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 468–479.

[28] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2348–2368, 2019.

[29] A. Tondwalkar, R. Recto, W. Weimer, and R. Jhala, "Finding bugs in liquid haskell,-," 2016.

[30] L. Applis and A. Panichella, "Hasbugs-handpicked haskell bugs," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 223–227.

[31] R. Just, D. Jalali, and 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*, 2014, pp. 437–440.

[32] A. Riboira and R. Abreu, "The gzoltar project: A graphical debugger interface," in *International Academic and Industrial Conference on Practice and Research Techniques*. Springer, 2010, pp. 215–218.

[33] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A sloc counting standard," in *Cocomo ii forum*, vol. 2007. Citeseer, 2007, pp. 1–16.

[34] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," *SIAM Journal on discrete mathematics*, vol. 17, no. 1, pp. 134–160, 2003.

[35] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[36] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–42, 2014.

[37] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–49, 2015.

[38] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[39] M. Waseem, P. Liang, G. Márquez, and A. Di Salle, "Testing microservices architecture-based applications: A systematic mapping study," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 119–128.

[40] G. Fraser and A. Zeller", ""mutation-driven generation of unit tests and oracles"," in *Proceedings of the ACM International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831728

[41] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 253–264.

[42] A. Kampmann and A. Zeller, "Carving parameterized unit tests," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 248–249.

[43] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 76–87.