View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Oxford Brookes University: RADAR

Mutation-based Evolutionary Fault Localisation

Diogo M. de-Freitas*, Plinio S. Leitao-Junior*, Celso G. Camilo-Junior* and Rachel Harrison†
*Instituto de Informática (INF), Universidade Federal de Goiás (UFG)
Alameda Palmeiras, Quadra D, Câmpus Samambaia, Goiânia, Goiás, Brazil
†Department of Computing and Communication Technologies
Oxford Brookes University, Wheatley Campus, OX33 1HX, Oxford, United Kingdom
Emails: diogom42@gmail.com, plinio@inf.ufg.br, celso@inf.ufg.br and rachel.harrison@brookes.ac.uk

***Abstract*—Fault localisation is an expensive and time-consuming stage of software maintenance. Research is continuing to develop new techniques to automate the process of reducing the effort needed for fault localisation without losing quality. For instance, spectrum-based techniques use execution information from testing to formulate measures for ranking a list of suspicious code locations at which the program may be defective: the suspiciousness formulae mainly combine variables related to code coverage and test results (pass or fail). Moreover previous research has evaluated mutation analysis data (mutation spectra) instead of coverage traces, to yield promising results. This paper reports on a Genetic Programming (GP) solution for the fault localisation problem together with a set of experiments to evaluate the GP solution with respect to baselines and benchmarks. The innovative aspects are is the joint investigation of: (i) specialisation of suspiciousness formulae for certain contexts; (ii) the application of mutation spectra to GP-evolved formulae, i.e. signals other than program coverage; (iii) a comparison of the effectiveness of coverage spectra and mutation spectra in the context of evolutionary approaches; and (iv) an analysis of the mutation spectra quality. The results show the competitiveness of GP-evolved mutation spectra heuristics over coverage traces as well as over a number of baselines, and suggest that the quality of mutation-related variables increases the effectiveness of fault localisation heuristics.**

I. Introduction

Fault localisation (FL) is the process of identifying the location of software faults that caused failure during testing activities. Hence, FL activities directly impact software cost and quality as they are onerous and time-consuming activities that grow with a project's complexity [1].

Spectra-based (or coverage-based) fault localisation techniques use program spectrum, i.e. the information from a program elements' coverage during execution of test cases. The spectra data commonly used for traditional fault localisation are how many test cases cover (or not) each software element and whether in those tests a failure has occurred.

For each element e in a program, the *Spectrum-based Fault Localisation* (SBFL for short) heuristic calculates a suspiciousness score $S(e)$, that represent the strength of association between e executions and failure occurrences. A rank of all $S(e)$ is calculated so that the developer can analyse all e from top (greater $S(e)$) to bottom until all faults are located.

All heuristics in SBFL are based on the same variables: c_{ep} (number of successful executions of the program that cover a certain element); c_{ef} (number of failed executions of the the program that cover a certain element); c_{np} (number of successful executions of the program that don't cover a certain element) and c_{nf} (number of failed executions of the program that don't cover a certain element).

In addition to the coverage variables, some works have applied mutation analysis data (mutation-related variables) to improve the performance of the FL heuristics, called *Mutation-based Fault Localisation* (MBFL for short) [2], [3], [4].

Mutation Analysis is a is a quality assessment tool for test case sets that applies punctual modifications (mutations) to a program [5], [6]. Each modification renders a modified version of the original program (the mutants). If the output of a mutant is different from the original program, it is said that the mutant was "*killed*". A mutation score is the proportion of killed mutants in relation to the non equivalent mutants. A mutant is said to be *equivalent* if there is no test case which can distinguish the outputs of the mutant and the original program.

MBFL was proposed by Papadakis *et al.* as a combination of mutation analysis and SBFL [2]. Mutation operators are applied to every statement covered by some failed test, then each generated mutant is executed by the test set and the information about which test case was killed by which mutant is stored. Finally a suspiciousness score $S(n)$ is calculated for each mutant in the same statement and the maximum score is the suspiciousness score of the statement. As in SBFL, after $S(n)$ is calculated, a list is organised in descending order to guide the investigation for fault localisation.

Evolved formulae based on Genetic Programming (GP) was introduced and empirically evaluated by Yoo [7] and next theoretically analysed by Xie *et al.* [8] who stated that GP can be an adequate tool for designing the risk evaluation of program elements. Yoo's approach has trained formulae for a set of programs as a whole (not for a particular program) and applied just coverage variables to compose new FL heuristics.

The latest study on theoretical and empirical analysis of GP-based solutions pointed out as further work [9]: (i) *the designing of formulae that are effective in certain contexts*: the generated metrics consider isolated projects instead of sets of projects, i.e. the suspiciousness formula is created to a particular program (program-oriented heuristic) as a result of a training process that learns from the known faults of such program; and (ii) *the use of signals other than program spectra*: the use of other source of fault data distinct from the coverage spectra such as the mutation spectra.

In the light of the competitive results of using humandesigned MBFL (e.g. [2] inter alia) and the promising conclusions of using GP to generate SBFL metrics ([7] inter alia), this paper reports the designing and analysis of an evolutionary approach to compose program-oriented MBFL heuristics automatically. Our work deals with both gaps addressed in [9] and also analyses the effectiveness of heuristics grounded on mutation versus coverage variables on a evolutionary approach, specifically the Genetic Programming algorithm.

Concerning research questions this study seeks evidence of: (RQ1) ***how effective are the GP-evolved solutions based on mutation variables from a relative perspective?*** and (RQ2) ***how effective are the GP-evolved solutions based on mutation variables from an absolute perspective?***. In other words, the overall ability of localising faults and the number of program elements investigated to find the defective ones, respectively.

Recently Pearson *et al.* have found MBFL heuristics perform poorly on real faults [10] but they did not consider the quality of mutation data in their study. Thus we also investigate (RQ3) ***how does mutation spectra quality impact FL ability?***; i.e. whether the number of available mutants is a relevant factor for the performance of GP-evolved MBFL.

This paper is structured as follows. Section II presents related work; Section III describes the proposed approach for evolutionary construction of suspiciousness formulae; Section IV presents the evaluation methods employed; Section V presents the results of the proposed method and classic heuristics; and Section VI concludes and presents future works.

II. Related Work

As the complexity and size of software projects grew, the use of manual techniques for fault localisation became impractical. This motivated the development of techniques to allow automating the fault localisation process so that human intervention was not necessary. Wong *et al.* [11] present a recent survey of fault localisation techniques.

In general automation initiatives proposed formulae to calculate the odds of faulty program elements. Jones *et al.* [12] presented a tool for visualisation of suspicious code called Tarantula. The tool displays each element of code in a colour in the spectrum between the green and red. The colours of the code were calculated by the measure *S(e)* in Formula (1).

$$S(e)_{tarantula} = \frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}} + \frac{c_{ep}}{c_{ep}+c_{np}}} \tag{1}$$

The Ochiai metric was adapted from molecular biology (it is used to calculate genetic similarity) to a fault localisation measure in [13]. The study indicated that Ochiai outperformed the measures used by three different tools, including Tarantula, in the Siemens suite of programs. Later OP^2 was proposed and proved by Naish *et al.* as an optimal SBFL heuristic for the If-Then-Else-2 (ITE2) model program [14].

A. Search-based Fault Localisation

Wang *et al.* proposed a search-based model to build composite linear measures based on metaheuristics [15]. The methodology was structured in two phases: training and deployment.

In the training phase, a Genetic Algorithm (GA) was applied to combine the 22 association measures from Lucia *et al.* [16], including Tarantula and Ochiai. Thus, the generated composite heuristic is a linear combination of previous measures.

The fitness function f was the average proportion of program elements that need to be investigated to locate the first fault in a training set of programs.

In the deployment phase, the best composite measure found by the training phase is used to locate the faults within the deployment set of programs. A rank for each program is assembled by ordering its elements by the suspiciousness score calculated by the composite linear measure.

In addition to the initial proposal for search-based fault localisation, which was introduced in [15], there are other evolutionary initiatives for building and analysing suspiciousness metrics. Yoo introduced a Genetic Programming (GP) approach for evolving risk assessment formulae, which were empirically evaluated by using 92 faults distributed in faulty versions of four programs [7]. The formulae were built generically, i.e. the training phase does not evolve formulae for particular programs but for a whole set of projects, similarly to the approach in [15].

Xie *et al.* performed theoretical evaluations of Yoo's GP-evolved formulae for programs with a single fault, and stated that GP can be an adequate tool for designing the risk evaluation of program elements [8]. Yoo *et al.* pointed out the state of the art of GP for search-based formulae and inferred that no single formula can dominate all, including the GP-evolved ones, i.e the optimal formula does not exist [9].

B. Mutation-Based Fault Localisation

The key idea of MBFL is to assign suspiciousnesses to injected mutants, based on the assumption that test cases that kill mutants carry diagnostic power: the more often a statement s affects failing tests, and the less often it affects passing tests, the more suspicious the statement is considered [10].

In the approach of Papadakis *et al.* [2] a suspiciousness score is calculated with formulae like Tarantula, Ochiai or OP^2 treating killed mutants as covered elements and live mutants as uncovered elements. Hence, for MBFL the following notation is adopted to adapt standard SBFL formulae: m_{kf} is the number of negative test cases that killed the mutant (analogous to c_{ef}), m_{kp} is the number of positive test cases that killed the mutant (analogous to c_{ep}), m_{nf} is the number of negative test cases that did not kill the mutant (analogous to c_{nf}) and m_{np} is the number of positive test cases that did not kill the mutant (analogous to c_{np}). The Formula (2) corresponds to the MBFL version of Tarantula.

$$S(e)_{mbfl-tarantula} = \frac{\frac{m_{kf}}{m_{kf}+m_{nf}}}{\frac{m_{kf}}{m_{kf}+m_{nf}} + \frac{m_{kp}}{m_{kp}+m_{np}}} \tag{2}$$

III. Genetic Programming for MBFL Formulae

Genetic Programming (GP) was proposed by John Koza as a way to search for the fittest individual computer program in the space of all possible computer programs composed of functions and terminals appropriate to the problem domain [17]. In a GP algorithm, a population of computer programs is bred over many generations using the Darwinian principle of natural selection – survival and reproduction of the fittest – in addition to genetic recombination and other natural operations such as: crossover, mutation, gene duplication and gene deletion. The initial population is comprised of randomly generated computer programs formed by the appropriate functions and terminals. The functions may be standard arithmetic operations, programming operations, mathematical functions, logical functions or domain-specific functions [17].

In the FL problem each individual represents a candidate suspiciousness formula to solve the problem. The population is a set of solutions which evolve according to the GP algorithm aiming to achieve better equations to calculate how suspicious each program element is. Thus the search space is the whole set of all valid formulae composed by the functions and the terminals selected to initiate the GP algorithm. Note that non-linear equations are commonly obtained from this process.

In [7] GP was applied to compose SBFL heuristics using four coverage variables $\{c_{ef}$, c_{ep}, c_{nf} and $c_{np}\}$ and the integer constant 1 (one) as terminals and the set of functions: the four basic mathematical operations and square root. The fitness function was similar to the f fitness function from Wang *et al.* [15] i.e.: the minimisation of the mean proportion of code that needs to be evaluated to locate all faults.

This paper presents a development over the aforementioned research efforts to generate program-customised MBFL formulae. We used GP configured with terminals – the four basic mutation variables $\{m_{kf}$, m_{kp}, m_{nf} and $m_{np}\}$ – and the constant 1 (inspired by [7]), and a set of functions which is composed by the four basic mathematical operations (sum; subtraction; multiplication; protected division, i.e. returns one when dividing by zero) and protected square root, i.e. square root of the modulus of a number.

Since a statement usually has more than one mutant, the maximum mutant suspiciousness is considered the element's suspiciousness. As MBFL techniques perform poorly on defects that involve unmutable statements [10], statements with no mutants are assigned the worst score. After the suspiciousness calculation, a ranking of elements is constructed ordering the elements in descending order of suspiciousness.

The proposal is structured in two stages: training and deployment. In the first the engine trains a new heuristic to a set of buggy versions of one program with the aforementioned functions and terminals. In the latter the new arranged heuristic is applied to a different set of versions from the same program, i.e. the sets of versions used in training and deployment are disjoint. As stated by Wang *et al.* a good composition should be able to locate many bugs in the training set with a high accuracy [15]. In a real world scenario the training phase is done over a set of previous versions with known faults while the deployment phase consist of applying the trained heuristic to debug newer versions of the program.

Following the aforementioned studies, the innovative aspects of our approach are the joint investigation of:

- specialisation of suspiciousness for certain contexts;
- the application of mutation spectra to GP-evolved formulae, i.e. signals other than program coverage;
- a comparison of the effectiveness of coverage spectra and mutation spectra in the context of evolutionary approaches. i.e. the analysis of suspiciousness heuristics in terms of absolute (e.g. average score) and relative (e.g. accuracy and wasted effort) performance;
- an analysis of the mutation spectra quality and its impact on fault localisation.

IV. Experiments

To evaluate the proposed method, we have performed three-fold cross validation. Initially the sets of versions of the same program were randomly divided into three subsets each. In every execution two subsets were used as the training set while the remaining one was used to evaluate the trained heuristic as in the deployment phase. Notice that the versions used in deployment were never used at the same time during the training phase. To run one round of experiments for each program, the subsets were alternated. The process is distinct from the one used by Xuan and Monperrus [18] in that for each subject program, 30% of faults are randomly selected to form the training data and the rest (70%) of faults are used for evaluation. Thus, in our approach each subset serves as the deployment set once and one round of experiments is comprised of three executions. Since the experiments use meta-heuristics, we have specified as 30 the number of experiment rounds (total of 90 executions) aiming to reduce the stochastic effects. A fine granularity (line of code) was used so that the method can be evaluated with respect to its ability to locate precisely the faulty lines of code.

A. Benchmark

The programs used in the experiments were obtained from benchmarks: *Siemens Suite* [2], [15], [16], [19], [9]; *Codeflaws* [20], a collection of programs from the *Codeforces* online database, a platform for programming contests; and *Defects4J* [21]. The benchmarks help to build repeatable experiments and enable experiments to later promote industry-based evaluation.

The former was downloaded from the *Software-artifact Infrastructure Repository* (SIR) [22], a project to distribute software-related artefacts for experimentation with program analysis, software testing, and education. The selected programs were *print_tokens2*, *schedule* and *tcas*. The selected Siemens programs were used with a reduced test set provided by SIR. From *Codeflaws*, the programs from problem *475-A* were selected. Regarding the single-bug experiments, some versions were not used because the fault was in a non-instrumented line (e.g. variable declaration bug), the test set did not reveal the fault or the fault was not located in a single line of code. The spectra for the programs were generated with the tools *lcov*, *gcov* (a GNU standard test coverage tool) and a custom program. In order to generate mutants, the *Proteum/IM 2.0* was used [23].

Defects4J [21] is a database and framework of projects with real bugs that aims to offer data for reproducible studies in software testing research, aiming to evaluate how the mutant availability impacts on the quality of mutation-based fault localisation. We collect coverage and mutation data of *Math*, a *multi-bug* real project of Defects4J, by using the repository created by Pearson *et al.* [10].

Another third party tool used was the Python framework DEAP [24], which allows the abstraction of the basic operators necessary for the implementation of evolutionary algorithms and concurrent parallel programming. For the implementation of both the GA and the GP, DEAP was used with basic settings. Hence, only the necessary parameters were defined, using the standard configurations unchanged when possible.

B. Evolutionary Parameters

The parameters applied in both GP algorithms were set as: generations: 1000; population: 100 individuals; crossover probability: 0.85; mutation probability: 0.07; reproduction probability: 0.08; maximum crossover depth: 20; maximum initial depth: 10; minimum initial depth: 2; selection method: roulette rank; crossover operator: one point crossover (swap of randomly selected sub-trees between parents); mutation operator: uniform mutation (replace an sub-tree in the individual with an new one randomly generated); next generation selection: Descendents with elitism (persistence of the best individual of the parents population).

C. Method Evaluation

The set of baseline heuristics was selected from state-of-the-art research on SBFL and MBFL [10], [19] and noted as the most popular ones in the survey [11]. The selected SBFL heuristics were Tarantula [12], [2], Ochiai [13], [19], OP^2 [14], [19], Barinel [25] and DStar [26] together with its respective MBFL counterparts. Also, Yoo's GP-based approach was adapted as an analogue SBFL-oriented version of the proposed method so that it would haver an evolutionary comparison target. Altogether, 11 baseline heuristics were chosen.

V. Analysis

To evaluate the performance of the suspiciousness equations in terms of the number of investigated code elements needed to locate faults, we apply three evaluation measures:

- *Average Score*: refers a metric commonly used in fault localisation studies, e.g. [19], that quantifies the average effort to locate all faults in a set of programs according to the suspiciousness rank provided by the technique. The *score metric* is calculated as a fraction of the suspiciousness rank to the amount of executed statements.
- *Accuracy* (**acc**@n): refers to the number of faults that have been localised within the top n places of the ranking (higher values are better); we use 1, 3 and 5 for n.
- *Wasted Effort* (**wef**@n): refers to the amount of work wasted looking at non-faulty program elements (lower values are better); we also use 1, 3 and 5 for n.

Similarly to *average score*, the *accuracy* and the *wasted effort* measures have been used in fault localisation experiments including the recent ones such as in [27], [28].

A. Effectiveness of Evolved MBFL Heuristics

To answer RQ1 and RQ2 – *how effective are the GP-evolved solutions based on mutation variables from relative and absolute perspectives?*, respectively – we analyse:

1) the evaluation measures for all programs as a whole;
2) the average rank related to all evaluation measures;
3) the comparison of FL techniques in individual programs.

From these three points, the results obtained from the *average score* deal with RQ1 and the results from *accuracy* and *wasted effort* answer RQ2.

TABLE I
Evaluation measures for all programs as a whole: *average score*, *acc* and *wef* values.

Technique	Avg Score	acc@n			wef@n		
		1	3	5	1	3	5
TAR-cov	47.14%	1	8	10	50	137	220
TAR-mut	43.98%	2	8	14	49	136	211
OCH-cov	46.48%	2	9	14	49	135	212
OCH-mut	53.60%	3	6	10	48	138	221
OP2-cov	40.27%	3	11	15	48	129	202
OP2-mut	59.86%	0	2	7	51	149	238
BAR-cov	48.37%	1	8	10	50	137	220
BAR-mut	43.70%	2	8	14	49	136	211
DST-cov	48.02%	2	8	10	49	136	219
DST-mut	46.78%	1	7	12	50	139	219
GP-cov	**30.00**%	0.33	4.63	10.17	50.67	145.63	232.40
GP-mut	36.38%	**6.47**	**15.40**	**20.54**	**44.53**	**118.90**	**181.14**

The results are shown in Tables I, II and III. The proposed evolutionary MBFL heuristics construction technique is referred to as "*GP-mut*" while the evolutionary construction of SBFL heuristics is called "*GP-cov*". Also, the human-based SBFL heuristics Tarantula, Ochiai, OP^2, Barinel and DStar are referred by using the suffixes *-cod* and *-mut* that refer to the use of coverage and mutation variables, respectively, such as "*TAR-cov*" and "*TAR-mut*". According to RQ1 and RQ2, the focus is on the effectiveness of GP-evolved solutions based on mutation variables, i.e. how effective is *GP-mut* in regard with the baselines, specially related to GP-evolved ones based on coverage variables (*GP-cov*).

Evaluating all programs as a whole. An overall comparison is shown in Table I. It presents the *average score* as well as the *acc* and *wef* values for all programs as a whole, the best values are in bold face. For instance in fourth column of Table I (*acc@3* values), *GP-mut* scored the faulty element at the top-three on average 15.40 times whereas *OP2-cov* achieved this 11 times. Whereas *GP-cov* and *GP-mut* are trained for each program and their results are averaged out afterwards, they are expected to outperform the other methods. As a matter of fact, *GP-mut* outperformed the others in all of the *acc* and *wef* values, while it was the second-best for *average score*, but in turn *GP-cov* was superior on the relative perspective as it reached the best result for *average score*. Specifically for

TABLE II
AVERAGE RANK FOR *average score*, *acc@n* AND *wef@n*.

Avg. Score	acc@1	acc@3	acc@5	wef@1	wef@3	wef@5
GP-cov: 2.75	**GP-mut: 4.5**	**GP-mut: 1.5**	**GP-mut: 3**	**GP-mut: 4.5**	**GP-mut: 1.5**	**GP-mut: 1.75**
OP2-cov: 4	OP2-cov: 8.25	OP2-cov: 6.5	OP2-cov: 6	OP2-cov: 8.25	OP2-cov: 6.5	OP2-cov: 5.25
GP-mut: 4.5	**GP-cov: 9.5**	OCH-cov: 7.5	OCH-cov: 6.25	**GP-cov: 9.5**	OCH-cov: 7.5	OCH-cov: 6
OCH-cov: 4.75	OCH-mut: 9.5	TAR-cov: 8.5	**GP-cov: 7.5**	OCH-mut: 9.5	DST-cov: 8	DST-mut: 7.75
DST-cov: 5.75	OCH-cov: 9.75	BAR-cov: 8.5	TAR-cov: 8.75	OCH-cov: 9.75	TAR-cov: 8.5	TAR-mut: 8
TAR-cov: 6.5	DST-cov: 9.75	DST-cov: 8.5	BAR-cov: 8.75	DST-cov: 9.75	BAR-cov: 8.5	BAR-mut: 8
BAR-cov: 7.5	TAR-mut: 10	**GP-cov: 9.25**	DST-cov: 8.75	TAR-mut: 10	**GP-cov: 9.5**	DST-cov: 8.25
TAR-mut: 9	BAR-mut: 10	TAR-mut: 9.75	DST-mut: 8.75	BAR-mut: 10	TAR-mut: 9.75	OCH-mut: 8.5
BAR-mut: 9	TAR-cov: 10.5	BAR-mut: 9.75	TAR-mut: 9.25	TAR-cov: 10.5	BAR-mut: 9.75	TAR-cov: 8.75
DST-mut: 9.25	BAR-cov: 10.5	DST-mut: 10	BAR-mut: 9.25	BAR-cov: 10.5	OCH-mut: 10	BAR-cov: 8.75
OCH-mut: 9.75	DST-mut: 10.5	OCH-mut: 10.5	OCH-mut: 10	DST-mut: 10.5	DST-mut: 10.25	**GP-cov: 8.75**
OP2-mut: 10.5	OP2-mut: 12	OP2-mut: 12	OP2-mut: 10.75	OP2-mut: 12	OP2-mut: 12	OP2-mut: 10.25

acc values, the *GP-mut* scores are much higher than the others (thus the most accurate for FL on the absolute perspective).

Evaluating the average rank. To capture the positional performance of the techniques, we calculate the average ranking position related to the evaluation measures. For each evaluation measure (*average score*, $acc@1$, $acc@3$, and so on), the *average rank* of a method is the mean position in the rank over all programs. If this metric for a method is the same as another, all of them are given the same ranking position: the lowest for them (e.g. if three methods are tied at ranking position two of $wef@5$, so all of them are at ranking position four for $wef@5$). The average ranks are presented in Table II, the *GP-mut* and *GP-cov* values are highlighted in bold face. Each column is sorted so that best average rank values appear first. The average ranks of *GP-mut* are superior across almost all columns as shown in Table II, i.e. the best with respect to all measures of *accuracy* and *wasted effort*, the exception is on *average score* despite the third position. Successively *GP-cov* had the best performance on a relative perspective (*average score*) that is consistent with the aforementioned evaluation of all programs as a whole.

Evaluating individual programs. In a detailed view of the experiments, Table III shows the results for the evaluation measures (*average score*, *acc@n* and *wef@n*) for each individual program. The best technique associated with each measure for every program is in boldface. It's noticeable that the performance of *GP-mut* is superior in each measure related to Program *tcas*, which has more faulty versions (i.e. better training capability). In the other programs *GP-mut* loses in *average score*, but it is the best in most *acc* and *wef* values. This indicates that *GP-mut* puts more faulty program elements at top positions in the suspiciousness rankings than the others. We conjecture the low performance of *GP-mut* on Programs *475-A*, *print_tokens2* and *schedule* is due to the fact that they have few defective versions. This behaviour occurs similarly with *GP-cov*, but according to the content of Table III *GP-mut* is seemingly more sensitive to training than the latter.

In summary, the *GP-mut* results are consistent with each other when evaluating all programs as a whole, individual programs, as well as *average ranks*: well-adapted (2nd and 3rd) on a relative perspective and the best on all the measures from an absolute perspective. On the relative perspective *GP-cov* presented good results showing agreement with previous work (e.g. [7], [9]). Thus GP-evolved heuristics grounded on mutation spectra revealed competitiveness with respect to their coverage-based counterpart baseline heuristics.

B. Quality of Mutation Spectra

MBFL techniques deal with whether a change to a statement alters the test outcome. The more often a statement affects failing tests, and the less often it affects passing tests, the more suspicious the statement is considered [10].

The number of mutants depends commonly on aspects such as the type of statement, the mutation operators, the mutant prioritisation process, among others. As the mutation spectra includes the mutants generated by the program's statements, we investigate *how does mutation spectra quality impact FL ability? (RQ3)*; i.e. how the number of available mutants is a relevant factor for the performance of GP-evolved MBFL.

Pearson *et al.* have found recently MBFL heuristics perform poorly on real faults [10], but they do not consider the availability of mutation data in their study. In our experiments, we use the program *Math* from Defects4J, which is one of the programs that have prompted the Pearson *et al.*'s conclusions.

To answer RQ3 we deal with two issues:

- **Fault Of Omission (FOO).** As the key idea of MBFL is to assign suspiciousnesses to injected mutants, based on the assumption that test cases that kill mutants carry diagnostic power for FL [10], omission faults do not provide diagnosis data for MBFL techniques. Furthermore, the Defects4J repository (see Subsection IV-A) treats a single omission fault (*e.g.* a missing statement or method) as a multi-bug scenario: it considers each location potentially related to the omission as a new fault (*e.g.* the statement nearest to the possible fix locations). In other words, there may be more than one possible place to insert fixing commands, so the cardinality between FOO and fixing location is one-to-many.
- **Minimum average number of mutants (MinAM).** We select faulty versions from the subject based on *minimum average number of mutants of defective statements* that represents a threshold k on which to base the measure of mutation spectra quality in our analysis.

TABLE III
COMPARISON OF FAULT LOCALISATION TECHNIQUES IN INDIVIDUAL PROGRAMS.

Prog	Technique	Avg Score	acc@n			wef@n		
			1	**3**	**5**	**1**	**3**	**5**
475-A	TAR-cov	61.21	0	0	1	8	24	39
	TAR-mut	86.07	0	0	1	8	24	38
	OCH-cov	69.00	0	0	1	8	24	39
	OCH-mut	82.12	0	0	1	8	24	38
	OP2-cov	69.00	0	0	1	8	24	39
	OP2-mut	82.12	0	0	1	8	24	38
	BAR-cov	69.10	0	0	1	8	24	39
	BAR-mut	84.09	0	0	1	8	24	38
	DST-cov	69.00	0	0	1	8	24	39
	DST-mut	82.12	0	0	1	8	24	38
	GP-cov	**55.51**	0	0	0	8	24	40
	GP-mut	75.73	0	**1.37**	**2.37**	8	**21.3**	**32.57**
print_tokens2	TAR-cov	36.21	1	3	3	9	23	37
	TAR-mut	45.35	0	0	0	10	30	50
	OCH-cov	31.68	**2**	**4**	4	**8**	**21**	33
	OCH-mut	69.40	0	0	0	10	30	50
	OP2-cov	**18.19**	**2**	**4**	4	**8**	**21**	33
	OP2-mut	73.35	0	0	0	10	30	50
	BAR-cov	36.21	1	3	3	9	23	37
	BAR-mut	45.35	0	0	0	10	30	50
	DST-cov	35.65	**2**	3	3	**8**	22	36
	DST-mut	59.25	0	0	2	10	30	47
	GP-cov	38.15	0	0.87	0.90	10	28.27	46.50
	GP-mut	33.94	1.67	3.83	**5.03**	8.33	21.27	**31.43**
schedule	TAR-cov	46.81	0	1	2	7	19	29
	TAR-mut	73.31	0	0	0	7	21	35
	OCH-cov	44.83	0	1	3	7	19	27
	OCH-mut	73.31	0	0	0	7	21	35
	OP2-cov	24.34	0	1	**4**	7	19	**25**
	OP2-mut	73.31	0	0	0	7	21	35
	BAR-cov	46.81	0	1	2	7	19	29
	BAR-mut	73.31	0	0	0	7	21	35
	DST-cov	46.43	0	1	2	7	19	29
	DST-mut	73.31	0	0	0	7	21	35
	GP-cov	**20.70**	0.33	0.77	2.43	6.67	19.37	28.90
	GP-mut	63.96	**1.53**	**1.73**	1.90	**5.47**	**16.10**	26.37
tcas	TAR-cov	47.10	0	4	4	26	71	115
	TAR-mut	22.60	2	8	**13**	24	61	**88**
	OCH-cov	45.68	0	4	6	26	71	113
	OCH-mut	33.43	3	6	9	23	63	98
	OP2-cov	44.20	1	6	6	25	65	105
	OP2-mut	44.20	0	2	6	26	74	115
	BAR-cov	47.10	0	4	4	26	71	115
	BAR-mut	22.66	2	8	**13**	24	61	**88**
	DST-cov	46.75	0	4	4	26	71	115
	DST-mut	23.96	1	7	9	25	64	99
	GP-cov	21.51	0	3	5	26	74	117
	GP-mut	**17.78**	**3.27**	**8.47**	11.17	**22.73**	**60.23**	90.77

In Table IV Line 1 shows the minimum average number of mutants k and the following lines present the number of versions of Program $Math$ based on the threshold k from MinAM: Lines 2 and 3 refer to versions with and without faults of omission, respectively. For instance, from the 77 original versions (Line 2), 41 of them have no FOOs (Line 3). There are an number of versions (19 and 16 with and without FOOs, respectively) whose faulty commands have less than two mutants on average. Potentially this indicates poor MBFL data source for learning about defective program elements.

Our goal is not to find the ideal value of K, but to use this answer of the research question. We use $K = 3$ after some pilot experiments, mainly due to the commitment on the number of versions used on training and to perform cross validation. In this way, we use the minimum average number of mutants in two distinct scenarios, so that the empirical analysis compares three sets of versions of Program *Math*: α is composed by all original versions in the repository (77 versions); β is composed by the versions whose minimum average number of mutants is limited to 3 (37 versions); and γ excludes omission faults from β (22 versions). So we consider Sets β and γ as enriched mutation spectra with respect to α as they have higher MinAM.

TABLE IV
VERSIONS OF PROGRAM *Math*, BASED ON THE MINIMUM AVERAGE NUMBER OF MUTANTS K OF DEFECTIVE STATEMENTS.

K	1	2	3	4	5	6	7	8	9	10
with FOOs	77	48	37	30	26	20	19	14	13	10
without FOOs	41	25	22	20	18	15	14	10	9	6

Table V shows the results of the three sets. As these sets have different numbers of versions, we use the average of the *accuracy* and *wasted effort* measures in order to compare them (*Avg. acc* and *Avg. wef* for short, respectively).

The results of *Set* α reveal the superiority of *GP-cov* on all evaluation measures. However regarding the enriched mutation spectra (Sets β and γ), *GP-mut* shows best scores on all *accuracy* measures; the superiority is also observed on all *wasted effort* measures. On *average score* measure, *GP-mut* presents higher sensibility to k threshold (MinAM) with respect to *GP-cov*: 14.41, 8.76 and 13.17% on Sets α, β and γ, respectively, while *GP-cov* reveals relative stability (4.82, 4.50 and 4.85%). In the omissions faults context, *GP-cov* exposes similar scores between Sets α and β (e.g. 4.82 and 4.50% on *average score* measure and 0.26 and 0.25 on *acc@1* measure), different from the *GP-mut* on all evaluation measures. Thus the performance of MBFL heuristics are more sensitive to the presence of faults of omission than coverage based heuristics.

The *average number of mutants* of defective statements improves the performance of the evolved MBFL heuristics. We use this measure as a quality factor of mutation spectra. It is worth noting that the experiments of RQ1 and RQ2 used robust benchmark for mutation spectra related to the RQ3 one, so their results are more positive for *GP-mut*. Moreover the presence of omission faults impacts on the effectiveness. These findings are good additions to the research area, but certainty they require further work before they can be generalised.

C. Statistical Analysis

We carried out statistical analysis following Arcuri *et al.*'s guidelines [29] using two complementary tests: the *Wilcoxon* Rank Sum Test to assess statistically significant differences with $\alpha = 0.05$, and *Vargha and Delaney's* $\hat{A}_{12}$ statistic for effect size comparison with *GP-mut* as A_1 and *GP-cov* as A_2. Table VI shows results of both statistical tests for *average score* and *accuracy* with respect to *GP-mut* and *GP-cov*, where overall expressive statistical differences have been observed across all programs.

TABLE V
EVALUATION MEASURES: PROGRAM *Math* (DEFECTS4J).

Set	Technique	Avg. Score	Avg. acc@1	Avg. acc@3	Avg. acc@5	Avg. wef@1	Avg. web@3	Avg. wef@5
α	GP-cov	4.82%	0.26	0.53	0.65	0.77	2.08	3.25
	GP-mut	14.41%	0.23	0.43	0.55	0.81	2.27	3.59
β	GP-cov	4.50%	0.25	0.48	0.61	0.82	2.27	3.69
	GP-mut	8.76%	0.37	0.67	0.81	0.72	1.91	2.91
γ	GP-cov	4.86%	0.10	0.27	0.40	0.91	2.61	4.03
	GP-mut	13.17%	0.27	0.48	0.57	0.75	2.03	3.08

In Table VI all *p-values* highlighted with boldface indicate statistical significance at the 5% level and all *V-D* values highlighted indicate a advantage in magnitude. If the *V-D* value is marked my a *, then there is an advantage to the GP-cov method, otherwise *GP-mut* has superior magnitude. On the *average score*, statistical significance (*p-value* $\leq$ *0.05*) is noted on almost all samples except Set 3 of *print_tokens2*. In turn *Vargha and Delaney*'s test shows the advantage of *GP-cov* despite the *GP-mut*'s superiority on program *tcas*.

Concerning accuracy (*acc* values in Table VI), "–" indicates that as both samples are identical (all zeros). Except for the Math's Set α (as expected), the *Wilcoxon*'s and *Vargha and Delaney*'s tests show the better performance of *GP-mut* (statisticall significance and effect size) regarding almost all the samples. The tests with Math's Set α indicate statistical advantage to *GP-cov*, but not as much as the other sets.

D. Threats to Validity

We mitigated threats to *internal validity*, *i.e.* reducing results by chance using: baseline methods and evaluation measures used in prior studies; multiple executions to reduce the algorithm's stochastic-nature; and existing open source frameworks that have been used in a variety of applications. With respect to the *external validity*, *i.e.* whether the results can be generalised, assessment on a large scale for faults and programs, inclusion of more fault localisation baseline methods and other programming languages are needed. However, we have selected real programs as well as artificial and real faults from different benchmarks that are used in many contexts related to software engineering experiments, which have reduced the bias to certain types of projects and have promoted the results to be aligned with the efforts over the research field. Threats to *construct validity* concerns whether we measured everything properly. We use absolute and relative evaluation metrics, and take their measures as in previous studies, so that they can communicate realistically the fault localisation ability.

VI. CONCLUSION

This paper reports on a Genetic Programming (GP) solution for the fault localisation problem together with a set of experiments to evaluate the found formulae with respect to literature studied baselines and benchmarks.

The innovative aspects are the joint investigation of: (i) specialisation of suspiciousness formulae for certain contexts; (ii) the application of mutation spectra to GP-evolved formulae, i.e. signals other than program coverage; (iii) a comparison of the effectiveness of coverage spectra and mutation spectra in the context of evolutionary approaches; and (iv) the impact of mutation spectra quality on SBFL effectiveness.

Well-known heuristics were used as baselines along with one metaheuristic method. The proposed mutation-based evolutionary approach showed overall better results, albeit a possible greater sensibility to training data (e.g. number of faulty programs, omission faults, quality of mutation data).

We measured the performance of GP-evolved heuristics based on mutation spectra from relative and absolute perspectives (RQ1 and RQ2): the percentage and the number of investigated elements to locate faults, respectively (the latter is closer to the measure perceived by software engineers). We used *the minimum average number of mutants* (MinAM) to measure mutation spectra quality and its impact on FL (RQ3).

The empirical analysis along with statistical analysis show that the proposal is competitive in both perspectives (relative and absolute) and our results are consistent with each other on evaluating all programs as a whole, individual programs, as well as *average ranks*. Specifically on absolute terms the method scored the faulty element top much more frequently than the baselines did. Also, the use of MinAM improves the performance of the evolved MBFL heuristics.

We conclude that GP-evolved heuristics grounded on mutation spectra represent a valid addition to evolutionary FL heuristics, and the quality of mutation-related variables increases the effectiveness of FL. As further work, we plan a study with hybridisation of coverage and mutation data, and larger scale investigation on mutation spectra quality.

REFERENCES

[1] I. Vessey, "Expertise in debugging computer programs: An analysis of the content of verbal protocols," *IEEE Trans. Syst. Man Cybern.*, vol. 16, no. 5, pp. 621–637, Sep. 1986.

[2] M. Papadakis and Y. L. Traon, "Using mutants to locate "unknown" faults," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 691–700.

[3] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST '14, Washington, DC, USA, 2014, pp. 153–162.

[4] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test., Verif. Reliab.*, vol. 25, pp. 605–628, 2015.

[5] R. Demillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

TABLE VI
WILCOXON AND VARGHA AND DELANEY'S $\hat{A}_{12}$ TESTS WITH GP-MUT (A_1) AND GP-COV (A_2).

		Avg. Score		**acc@1**		**acc@3**		**acc@5**	
Program	**T. Set**	**p-vaule**	**V-D**	**p-vaule**	**V-D**	**p-vaule**	**V-D**	**p-vaule**	**V-D**
475-A	1	**8.90E-13**	**1.00***	–	0.50	–	0.50	–	0.50
	2	**3.71E-3**	0.30	–	0.50	**2.10E-08**	**0.85**	**2.10e-08**	**0.85**
	3	**1.67E-07**	**0.86***	–	0.50	–	0.50	**1.67e-14**	**1.00**
print_tokens2	1	**3.43E-10**	**0.97***	**3.78E-10**	**0.90**	**1.67E-4**	0.74	**3.61e-09**	**0.90**
	2	**1.34E-12**	**0.00**	**5.64E-10**	**0.90**	**3.34E-11**	**0.93**	**3.34e-11**	**0.93**
	3	0.52	0.55	–	0.50	**2.37E-08**	**0.85**	**4.17e-12**	**0.98**
schedule	1	**1.40E-11**	**1.00***	**7.90E-4**	0.72	0.13	0.59	**7.62E-3**	0.36
	2	**8.35E-12**	**1.00***	**1.47E-09**	**0.88**	**1.97E-11**	**0.93**	**2.30E-4**	0.72
	3	**1.61E-06**	**0.86***	–	0.5	0.16	0.47	**6.11e-07**	**0.20***
tcas	1	**2.91E-3**	0.28	**1.55E-09**	**0.9**	**8.96E-13**	**1.00**	**8.97e-13**	**1.00**
	2	**2.31E-2**	0.66	**1.26E-12**	**0.98**	**1.90E-06**	**0.80**	**3.75e-05**	**0.77**
	3	**1.36E-11**	**0.01**	**6.46E-4**	0.67	**8.58E-10**	**0.92**	**2.59e-13**	**1.00**
Math (α)	1	**3.08E-08**	**0.92***	0.13	0.61	0.25	0.41	0.14	0.61
	2	**2.98E-11**	**1.00***	0.93	0.49	**3.64E-06**	**0.16***	**1.11E-3**	**0.26***
	3	**3.00E-11**	**1.00***	**4.76E-08**	**0.09***	**4.70E-10**	**0.04***	**8.68E-11**	**0.01***
Math (β)	1	**5.97E-07**	**0.88***	**3.09E-11**	**0.98**	**3.08E-11**	**0.99**	**2.44E-11**	**1.00**
	2	**2.98E-11**	**1.00***	**6.89E-05**	**0.79**	**2.06E-05**	**0.82**	**1.12E-2**	0.69
	3	**2.76E-11**	**1.00***	0.80	0.52	3.64E-3	0.71	**2.29E-06**	**0.85**
Math (γ)	1	**4.17E-10**	**0.97***	**1.62E-4**	**0.78**	**5.28E-4**	**0.75**	**7.40E-3**	0.69
	2	**2.02E-09**	**0.95***	**8.86E-05**	**0.77**	**1.51E-06**	**0.84**	**3.08E-08**	**0.74**
	3	**2.52E-08**	**0.92***	**1.16E-10**	**0.94**	**1.50E-10**	**0.97**	**9.20E-11**	**0.98**

[6] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments? [software testing]," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, May 2005, pp. 402–411.

[7] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Search Based Software Engineering*, ser. Lecture Notes in Computer Science, G. Fraser and J. Teixeira de Souza, Eds., 2012, vol. 7515, pp. 244–258.

[8] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *Proc. of the 5th International Symposium on Search Based Software Engineering - Volume 8084*, ser. SSBSE 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 224–238.

[9] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 1, pp. 4:1–4:30, Jun. 2017.

[10] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, Piscataway, NJ, USA, 2017, pp. 609–620.

[11] 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, Aug 2016.

[12] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *in Proceedings of ICSE 2001 Workshop on Software Visualization*, Toronto, ON, Canada, 2001, pp. 71–75.

[13] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '06, Washington, DC, USA, 2006, pp. 39–46.

[14] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[15] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11, Washington, DC, USA, 2011, pp. 556–559.

[16] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *2010 IEEE Intl. Conference on Software Maintenance*, Timișoara, Romania, Sept 2010, pp. 1–10.

[17] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA, USA, 1992.

[18] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 191–200.

[19] P. Gong, R. Zhao, and Z. Li, "Faster mutation-based fault localization with a novel mutation execution strategy," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2015, pp. 1–10.

[20] S. H. Tan, J. Y. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, *Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools*.

[21] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proc. of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.

[22] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Engg.*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[23] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. d. S. Simão, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero, "Proteum: A family of tools to support specification and program testing based on mutation," in *Mutation Testing for the New Century*, W. E. Wong, Ed., Norwell, MA, USA, 2001, pp. 113–116.

[24] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[25] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–99.

[26] 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, March 2014.

[27] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 273–283.

[28] T.-D. B. Le, D. Lo, C. Le Goues, and 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*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 177–188.

[29] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, May 2014.