A New Interactive Fault Localization Method with Context Aware User Feedback

Ferenc Horváth*, Victor Schnepper Lacerda*, Árpád Beszédes*, László Vidács*†, and Tibor Gyimóthy*†
*Department of Software Engineering, University of Szeged, Hungary
†MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, Hungary
{hferenc,lacerda,beszedes,lac,gyimothy}@inf.u-szeged.hu

***Abstract*—State-of-the-art fault localization tools provide a ranked list of suspicious code elements to aid the user in this debugging activity. Statistical (or Spectrum-Based) Fault Localization (SFL/SBFL) uses code coverage information of test cases and their execution outcomes to calculate the ranks. We propose an approach (called *iFL*) in which the developer interacts with the fault localization algorithm by giving feedback on the elements of the prioritized list. Contextual knowledge of the user about the current item (e. g., a statement) is exploited in the ranked list, and with this feedback larger code entities (e. g., a whole function) can be repositioned in the list. In our initial set of experiments, we evaluated the approach on the SIR benchmark using simulated users. Results showed significant improvements in fault localization accuracy: the ranking position of the buggy element was reduced by 72% on average, and *iFL* was able to double the number of faults that were positioned between 1-5.**

***Index Terms*—Statistical fault localization, spectrum based fault localization, testing, interactive debugging, user feedback.**

I. Introduction

Debugging is one of the most difficult and time consuming tasks in software development and evolution, since it involves human participation to a large degree and its subtasks are difficult to automate. In this work, *fault (bug) localization* is addressed, a necessary subactivity in which the root causes of an observed failure are sought.

There is a class of approaches to aid fault localization which are quite popular among researchers, called Statistical Fault Localization (SFL), or Spectrum-Based Fault Localization (SBFL) [1]–[3]. Recent studies highlighted some barriers to the wide adoption of SFL, including a high number of suggested elements to investigate [4], [5], and validity issues of empirical research [6], among others. Kochar *et al.* performed a systematic analysis of practitioner's expectations in the field [7], and also identified a number of challenges.

The basic intuition behind SFL is that those code elements (statements, functions, etc.) are more suspicious to contain a fault that are exercised by comparably more failing test cases than passing ones, while non-suspicious elements are traversed mostly by passing tests. One way to express the suspiciousness is to assign a value to each code element, *a suspiciousness score*, which can be used to *rank* the code elements.

When this ranked list is given to the developer for investigation, it is hoped that the fault will be found near the beginning of the list, hence providing a useful advice. A possible approach to measure the effectiveness of a SFL method is to investigate the average rank position of the actual faulty element relative to the total number of code elements [1] (henceforth the *Expense* metric). In particular, research showed that if the faulty element is beyond the 5th (or 10th according to some other studies) element, the method will not be used by practitioners because they need to investigate too many elements in vain [3]–[5], [7].

There are many different scoring mechanisms, but these are essentially all based on four fundamental statistics: counts of passing/failing and traversing/non-traversing test cases in different combinations [1], [3], [8]. Xie *et al.* examined the equivalence and hierarchy between a number of formulae [8], while Yoo *et al.* showed that there does not exist a perfect scoring formula which outperforms known techniques found by humans or even by automatic search-based methods [9].

One additional reason an SFL formula may fail is what researchers call *coincidental correctness* [10]–[12] (i. e., the situation when a test case traverses a faulty element without failing). This can happen quite often since not all exercised elements may have impact on the computation performed by a test case [12], and if there are relatively more such cases than traversing and failing test cases, the suspiciousness score will be negatively affected [10].

In this work, we propose a form of an *Interactive Fault Localization* approach, called *iFL*, which tries to address these inherent challenges by involving the user's knowledge about the system. In traditional SFL, the developer has to investigate several locations before finding the faulty code elements, and all the knowledge he or she a priori has or acquires is not used. In our approach, the developer interacts with the fault localization algorithm by giving feedback on the elements of the prioritized list. Contextual knowledge of the user about the next item is exploited in the ranked list (e. g., a statement), with which larger code entities (e. g., a whole function) can be repositioned in their suspiciousness. This way, the next proposed suspicious elements can be influenced in the hope to reach the faulty element earlier.

We evaluated the approach on the SIR benchmark [13] using simulated users and measureming the Expense metric improvements with respect to the traditional SFL method Tarantula [14]. In this initial set of experiments, we observed significant improvements in fault localization accuracy: the ranking position of the buggy element was reduced by 72% on average, and *iFL* was able to double the number of faults that were positioned between 1-5.

II. Motivating Example

For illustration, consider the example in Table I. This is a part of program replace from the SIR. Line 116 is a predicate inside function dodash, where an artificial fault is seeded: the relation is changed and the `+1` part is deleted (the original version of the code line is shown in a comment). There are three other functions in this program that closely participate in exposing this particular fault, getccl, omatch and locate. Function getpat is first called from the main program which indirectly calls getccl and eventually dodash to calculate and return a value. This value is subsequently passed to change and eventually to omatch and locate where the fault will be manifested in form of failing test cases.

Table I also shows the coverage relationship between some typical test cases and the code elements in question, which expose different behavior with respect to the suspicious elements. We can see that there are passing and failing test cases, and that they are exercising different parts of the program. The faulty statement is traversed both by passing and failing test cases. Column '0. iteration' corresponds to the suspiciousness scores computed by the Tarantula method along with the ranking position of the elements (ties are handled by averaging the positions of elements with the same score). It can be seen that there are several lines in functions getccl, omatch and locate that have higher scores than the faulty one from dodash, which will push it to the 11th-13th place.

We can explain the failure of the SFL in this case as follows. Recall the Tarantula formula for a code element s [14]:

$$T(s) = \frac{\frac{ef(s)}{ef(s)+nf(s)}}{\frac{ef(s)}{ef(s)+nf(s)} + \frac{ep(s)}{ep(s)+np(s)}},$$

where the functions $ef(s)$, $nf(s)$, $ep(s)$ and $np(s)$ count the number of test cases that execute s and fail, do not execute s and fail, execute s and pass, do not execute s and pass, respectively. Table II shows the four basic statistics for lines 116 (the actual fault), 366 (one of the most suspicious statements in the initial ranking) as well as 145 and 321 (the two most suspicious statements in intermediate iterations of our algorithm). We can observe that all failing test cases are exercising statement 116 $(30/30)$, while only $(25/30)$ statement 366. This, in itself, would make the first statement more suspicious, however, the counts for the passing test cases will change the result. In particular, a lot more passing test cases exercise statement 116 $(2280/5511)$ than statement 366 $(1066/5511)$. In other words, there are comparably more coincidentally correct tests for the actual faulty statement than for the other, and despite the correct ordering in terms of failing test cases, the final score will flip their relatioship.

III. Interactive Fault Localization

Our approach to improve SFL is to leverage the background and acquired knowledge of the developer about the system being debugged. Assuming that the SFL is performed on statement level, the developer could potentially make decisions on function level as well, thus guiding the SFL process in bigger steps. In our approach, we call this the *contextual knowledge* because, if given a code element (a statement), the developer could leverage the knowledge about its context (the function) and feed this information back to the SFL engine.

Suppose that the developer of our example is performing SFL and starts with the first highest ranked element, statement 366 (see columns 4-7 in Table I). Now, the developer looks at the function this statement belongs to and concludes that it is not likely to contain the fault (because it was not changed recently, or the developer examined it in a previous debugging session, etc.). This knowledge is then fed back to the SFL engine, which in turn reduces the suspiciousness scores for all contained elements to 0, sending other highly ranked elements to the end of the list. In the next iteration, the next highest suspicious element is given to the user, statement 321 of function locate. Now, based on contextual knowledge, the developer decides that this function is not suspicious as a whole, so the scores of all contained statements are reduced to 0. This is repeated for line 145 in the next iteration. These steps result in pushing several other elements to the end of the list, and moving the faulty element, statement 116, to the next rank position. This terminates the fault localization process with success. In this example, the effort required to locate the fault was reduced from 12 steps to only 5 (3 steps for removing the three functions and two steps in the final iteration to select the middle one from the three elements with the same suspiciousness score[1]).

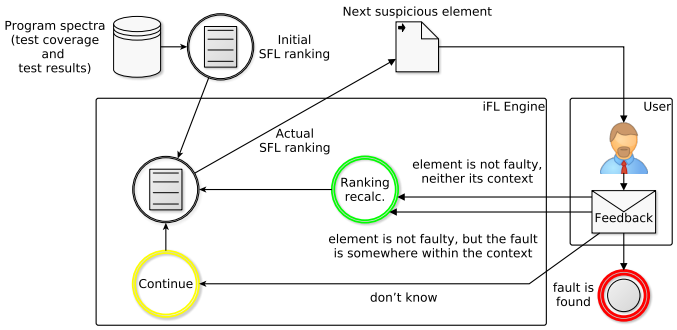

Fig. 1. Basic process of Interactive Fault Localization

This example illustrates the basic approach we use to improve SFL. Introducing user feedback to the loop has been proposed by other researchers as well, though using different approaches. The developer typically has additional information about the system of which the SFL engine is not aware of. For example, Li *et al.* [15], [16] reuses the knowledge about passing parameter values in a debugging session, Hao *et al.* [17] asks for feedback about the execution trace, Gong *et al.* [18] asks only for a simple yes/no feedback for a given statement. To our knowledge, contextual information about higher level entities (for instance, statement vs. enclosing function) has not yet been leveraged for interactive SFL.

[1]Selecting the middle element in the case of score ties is the approach we have chosen because it reflects the expected localization effort in terms of the rank positions.

TABLE I
EXAMPLE CODE AND FAULT LOCALIZATION PROCESS WITH SEEDED FAULT

Line	Source code: Code	Test cases: 557	560	855	857	864	Scores and ranks: 0. iteration		1. iteration		2. iteration		3. iteration	
93	`void dodash(delim, src, i, dest, j, maxset)`	•	•	•	•	•	0.658	(23.)	0.658	(20.)	0.658	(7.)	0.658	(5.)
115	`else if ((isalnum(src[*i - 1])) && (isalnum(src[*i + 1]))`	•	•	•	•		0.677	(14.)	0.677	(12.)	0.677	(5.)	0.677	(4.)
116	`&&(src[*i - 1] > src[*i])) { //faulty version`	•	•	•	•		0.707	(11.)	0.707	(9.)	0.707	(2.)	0.707	(1.)
116	`//&&(src[*i - 1] <= src[*i + 1])) { //original version`													
118	`for (k = src[*i-1]+1; k<=src[*i+1]; k++)`	•	•	•	•		0.707	(12.)	0.707	(10.)	0.707	(3.)	0.707	(2.)
122	`*i = *i + 1;`	•	•	•	•		0.707	(13.)	0.707	(11.)	0.707	(4.)	0.707	(3.)
123	`}`													
131	`bool getccl(arg, i, pat, j)`	•	•	•	•	•	0.658	(24.)	0.658	(21.)	0.658	(8.)	0	
144	`} else`													
145	`junk = addstr(CCL, pat, j, MAXPAT);`			•	•	•	0.709	(10.)	0.709	(8.)	0.709	(1.)	0	
305	`bool locate(c, pat, offset)`	•	•	•	•	•	0.762	(5).	0.762	(3.)	0		0	
313	`flag = false;`	•	•	•	•	•	0.762	(6.)	0.762	(4.)	0		0	
314	`i = offset + pat[offset];`	•	•	•	•	•	0.762	(7.)	0.762	(5.)	0		0	
315	`while ((i > offset)) {`	•	•	•	•	•	0.762	(8.)	0.762	(6.)	0		0	
317	`if (c == pat[i]) {`	•	•	•	•	•	0.765	(4.)	0.765	(2.)	0		0	
318	`flag = true;`		•		•	•	0.677	(15.)	0.677	(13.)	0		0	
319	`i = offset;`		•		•	•	0.677	(16.)	0.677	(14.)	0		0	
320	`} else`													
321	`i = i - 1;`	•	•	•	•	•	0.768	(3.)	0.768	(1.)	0		0	
322	`}`													
323	`return flag;`	•	•	•	•	•	0.762	(9.)	0.762	(7.)	0		0	
327	`bool omatch(lin, i, pat, j)`	•	•	•	•	•								
366	`if (locate(lin[*i], pat, j + 1))`			•	•	•	0.811	(1.)	0		0		0	
367	`advance = 1;`					•	0.665	(18.)	0		0		0	
368	`break;`			•	•	•	0.811	(2.)	0		0		0	
	Pass/Fail Status	**P**	**F**	**F**	**F**	**P**								

TABLE II
BASIC SFL STATISTICS FOR THE EXAMPLE PROGRAM

Line	ef	ep	nf	np	Tarantula score
116	30	2 280	0	3 231	0.707
145	25	1 882	5	3 629	0.709
321	30	1 662	0	3 849	0.768
366	25	1 066	5	4 445	0.811

Figure 1 shows a conceptual overview of our approach. The process starts by calculating an initial rank based on some traditional SFL approach (we used Tarantula, but any other method could be used as well). The elements are then shown to the user starting from the beginning of the list, and the SFL engine is waiting for user feedback. The user investigates the recommended element and gives one of the following answers: 1) fault is found, 2) element is not faulty, neither its context, 3) element is not faulty, but the fault is somewhere within the context, or 4) don't know. In our approach, the user makes the feedback, but in the present state of our research, we perform the experiments using simulated users.

Based on the feedback, the SFL engine performs the following actions. In the case of (1), the process terminates, while at (4) it is continued as usual with the next suspicious element (this means that in the worst case when the developer has no background knowledge, the method falls back to the pure SFL approach). In the remaining two cases, the SFL engine makes adjustments to the suspiciousness scores, recalculates the ranking and shows the next element from the new list to the user in the next iteration.

In our experiments, we used the following approach: in case of (2), the whole context (i. e., function) gets 0 score, while for (3) everything but the context is reduced to 0. This means that we are assuming the following about the user: A) We assume that the developer is competent and is able to give reliable answers of types (2) or (3), so that we can safely perform the nullation of the scores B) We assume that the developer can make a decision about the context as a whole, i. e., she can decide about every statement of a function in one step. If the developer would need to investigate each statement individually, we would fall back to the traditional process.

If these assumptions hold, the ranking position of the faulty element can only be reduced. Performing a sensitivity study to assess the effects of user imperfections on the improvements provided by *iFL* is a topic for future work.

IV. DESCRIPTION OF THE EXPERIMENT

The iterative approach requires the interaction between the user and the fault localization engine. At this phase of research, we are using simulated users, since an empirical study with actual human participation would require a large effort with uncertain outcomes. The same approach has been followed by most of the related research, e. g. [17], [18].

The main goal of our study was to find out *how much improvement in localization effectiveness and accuracy can iFL achieve over a traditional non-interactive SFL method?*

A. Experiment Setup

For SFL, we used the Tarantula algorithm [14] for suspicioness score calculation, since it is reported to be one of the most successful ones in different settings [3], and is often referred in literature. Regarding the user responses and SFL engine actions, we adopted a relatively simple but strict approach (meaning that there are no intermediate or uncertain responses or actions). Of the four possible responses explained in Section III, we will not use the fourth one, "don't

know". Furthermore, the given strategy for the actions will be employed, that is, reducing either the whole context or everything but the context to 0.

For user simulation, we used the actual position of the known fault, looking at its context and comparing it to the context of the actually recommended element in the rank list. We then generated the corresponding answers.

B. Evaluation Method

For computing the effectiveness of an SFL method, we follow the strategy to look at "elements that need to be investigated" by the programmer before finding the fault [1], using the "expected case" in the case of ties of the suspicioness scores [19]. We express this in a set of measures called *Expense* with two variants: an absolute one counted with the number of code elements (E) and a relative version compared to the length of the rank list (E'), which is the program size. Relative expense is frequently used in literature, but Parnin and Orso argued that absolute rankings are more helpful in practical situations [5]. The following formulae express precisely how to calculate these values (following [20]):

$$E = \frac{|\{i|s_i > s_f\}| + |\{i|s_i \geq s_f\}| + 1}{2},\ E' = \frac{E}{N} \cdot 100\%,$$

where N is the number of code elements, for $i \in \{1, \ldots, N\}$ s_i is the suspiciousness score of the ith code element and f is the index of the faulty code element.

To compare the *iFL* method to the traditional SFL, we will compute the Expense metrics (both absolute and relative) for both approaches, and compare them in terms of improvement relative to traditional SFL. In each iteration of the approach one block of code is decided upon (either the function of the context or everything outside it), hence each iteration will be counted as an equivalent of one rank position for calculating Expense. The amount of improvement will then be calculated for each defect and suitable averages will be produced.

Apart from the general average improvement, there is a set of improvements which are particularly important. As mentioned earlier, the practical usability of SFL depends on the position of the faulty element (in absolute terms, not in relative). Recent user studies report that developers tend to investigate only the top 5 or at most the top 10 elements in the recommendation list provided by localization methods before giving up [4], [7]. To measure and express the improvement between *iFL* and the traditional SFL approach, we used the concept of *accuracy* following Sohn and Yoo [21]. It counts the number of faults that have been localized within the top *n* places of the ranking (@1, @3 and @5).

C. Subject Programs

Seven small C/C++ programs from the Software-artifact Infrastructure Repository (SIR) [13] were included in the experiments, which are the so-called "Siemens" suite. This benchmark contains seeded faults, which were produced by mutation, and both the original and faulty versions are available. The subject programs are listed in Table III. Column 2 shows the size of the programs in lines of code (LOC) including the comment and empty lines, along with the number of executable code elements (CE) for which coverage information could be obtained. In column 3, the number of functions in the program is given (this corresponds to the context in *iFL*). The number of test cases in the test suite is presented in column 4, while the 5th one contains the number of available faulty versions (each version has exactly one fault in it).

The last column (Number of suitable faults) of Table III shows the number of defects we were able to use in the experiments: 1) We filtered out versions where there were multiple faulty code elements; 2) We omitted faults where GCOV was unable to record coverage e. g., headers and macros; 3) We omitted cases where the suspiciousness score of the faulty code element assigned by the actual SFL technique was zero (these cannot contain the fault due to no failing test cases traversing them).

TABLE III
DETAILS OF SUBJECT PROGRAMS FROM SIR

Program	LOC (CE)	No. func.	Tests	No. faults	No. suitable faults
printtokens	726 (277)	18	4 130	7	1
printtokens2	570 (262)	19	4 115	10	7
replace	564 (400)	21	5 542	32	22
schedule	412 (225)	18	2 650	9	2
schedule2	374 (198)	16	2 710	10	4
tcas	173 (95)	9	1 608	41	31
totinfo	565 (187)	7	1 052	23	18
Total	**3 384 (1 644)**	**108**	**21 807**	**132**	**85**

V. RESULTS

Table IV shows the improvements *iFL* achieved on SIR. The performance of the original SFL algorithm can be seen in column 2, which we used as the reference to evaluate *iFL*. Both absolute and relative versions of the Expense measure defined in Section IV are provided. A summarization row is provided as well with the corresponding average values. SFL prioritized the faulty code elements to the 25th place on average, which means that 15% of the executable code elements must be examined on average to find the faulty one.

Column 3 contains the same data for *iFL*, which produces an Expense of 6.86 (4.25%) on average. This means that in this case a programmer would need only about 7 steps to find the fault on average, which is notably better than for the original algorithm. Column 4 shows the difference between the absolute and relative Expense measures. Column 5 of the table contains a summary of improvements in terms of relative changes in the Expense values, (that is, the difference over the SFL base value in percentage), which is 72.42% on average.

Relative improvements are presented in more detail in Figure 2 in form of distributions per subject program. The median of the relative differences is similar for all programs, it is around 50-80%. But, for some subjects, the values take a wide range and for others they are more concentrated.

Table V shows the results of the accuracy metric for both traditional SFL and *iFL*. We can observe that *iFL* has much

TABLE IV
E (E') OF TARANTULA AND *iFL* ON SIR

Program	Tarantula	*iFL*	Diff.	Impr.
printtokens	5.00 (1.81%)	2.00 (0.72%)	-3.00 (-1.08%)	60.00%
printtokens2	30.71 (11.72%)	7.21 (2.75%)	-23.50 (-8.97%)	76.51%
replace	19.18 (4.80%)	4.70 (1.18%)	-14.48 (-3.62%)	75.47%
schedule	10.75 (4.78%)	5.50 (2.44%)	-5.25 (-2.33%)	48.84%
schedule2	77.38 (39.02%)	14.25 (7.18%)	-63.12 (-31.84%)	81.58%
tcas	21.65 (22.78%)	5.58 (5.87%)	-16.06 (-16.91%)	74.22%
totinfo	26.00 (13.90%)	10.33 (5.53%)	-15.69 (-8.39%)	60.36%
Average	**24.85 (15.43%)**	**6.86 (4.25%)**	**-17.99 (-11.19%)**	**72.42%**

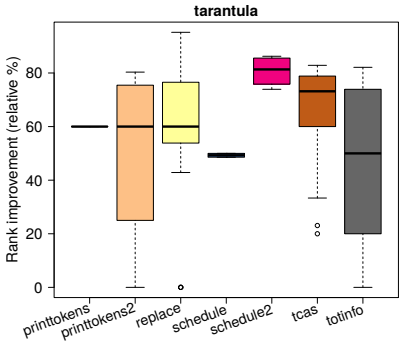

Fig. 2. SIR improvement distribution by programs

TABLE V
ACCURACY OF TARANTULA AND *iFL* ON SIR

Program	@1			@3			@5		
	SFL	*iFL*	Diff.	SFL	*iFL*	Diff.	SFL	*iFL*	Diff.
printtokens	0	0	-	0	1	+1	1	1	-
printtokens2	1	2	+1	3	4	+1	4	4	-
replace	2	5	+3	6	14	+8	7	18	+11
schedule	0	0	-	0	1	+1	1	1	-
schedule2	0	0	-	0	0	-	0	1	+1
tcas	0	5	+5	6	9	+3	6	18	+12
totinfo	0	0	-	2	3	+1	5	6	+1
Total	**3**	**12**	**+9**	**17**	**32**	**+15**	**24**	**49**	**+25**

higher accuracy than Tarantula in all three settings. For faults that were found in the 1st ranking position, *iFL* delivers 4 times more items compared to Tarantula. In the other two cases (@3 and @5), the advantage is practically twice as with the original algorithm. These are significant improvements, since accuracy counts the number of faults that have been localized within the most useful top n places of the ranking.

Summary: Compared to Tarantula, *iFL* achieved **72% improvement** in Expense, and resulted in 12, 32 and 49 faults in the top 1, 3 and 5 ranking positions, respectively, which means a **2-4 times better accuracy** than Tarantula.

VI. RELATED WORK

SFL/SBFL is one of the main approaches to fault localization [1]. However, these methods are still finding their way to be employed in practice [6], [7], [22]. For instance, most studies are carried out using artificial faults [3], and still the faulty element is usually placed far from the top of the rankings [4], [5]. Abreu *et al.* [20] investigated the accuracy of fault localization methods in practice. Since SFL heavily relies on the coverage and the pass/fail information, test suite properties directly affect fault localization [11], [23]. GZoltar [24] provides the ranked list of diagnosis candidates to help the user in practice. The FLINT method [25] improves the effectiveness of fault localization by trying to reduce the Shannon entropy regarding the locality of the fault.

Our approach is to use a context aware feedback method, hence the closest related works to our approach are the ones that change the ranking of program elements based on the user feedback iteratively [15]–[18], [26], [27]. Among these there are three closely related works [17], [18], [27]. These papers also incorporate the Siemens suite into their set of subject programs. However, the setup of the experiments and the metrics used for the evaluation are different in every case, which makes it difficult to compare our results directly to the reported ones in these works. The differences include the set of defects, the total number of code elements, different interpretation of the localization effectiveness metrics, etc.

For reference, Gong *et al.* report [18] that their approach yields about 12% absolute improvement in Expense over Tarantula. The best performing approach of Hao *et al.* [17] is reported to achieve about 8% in a similar measure. Lei *et al.* [27] used a similar metric to ours to measure the relative effectiveness improvement. They conclude that the improvement is around 21% compared to the 72% achieved by *iFL*.

We concentrated on the analysis of test case executions, but there are other approaches for fault localization as well. These include, slicing [28], statistical [29], model [30], mutation [31], metamorphic testing [32], and IR-based techniques [33].

Approaches, that may incorporate user feedback, are loosely related to the topic of this article e. g., the works of Zeller *et al.* on delta debugging [34], crowd debugging [35], Lin *et al.* [36], as well as algorithmic debugging and testing [37].

VII. CONCLUSIONS

In this work, we presented *iFL*, an approach to extend traditional Statistical Fault Localization by providing the ability for the developer to interact with the SFL algorithm. The user gives feedback on the elements of the prioritized list, based on which the suspiciousness scores are adjusted. We exploit contextual knowledge of the user about the next item in the ranked list, with which larger code entities can be repositioned in their suspiciousness.

In the present phase of the research, we used simulated users, which might not accurately represent real life scenarios. However, the empirical results show quite big improvements with respect to a traditional method: the localization efficiency in terms of Expense improved on average by 49-82% compared to Tarantula. Furthermore, considering the most important top n items of the ranked lists, our approach has 2-4 times better accuracy than the original SFL algorithm. This lets us believe that usable improvements would be obtained also with real users, which suffer from various imperfections such as limited knowledge and error proneness. It is therefore our next goal to evaluate the method by simulating user imperfection

and eventually with real users and various expense measures, and on different benchmarks. We also plan to investigate how the different user actions affect our method, and other strategies for the adjustment of suspiciousness scores.

Measurement data are available at: http://tinyurl.com/ifldata

ACKNOWLEDGMENT

Árpád Beszédes was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. This work was partially supported by grant 2018-1.2.1-NKP-2018-00004 "Security Enhancing Technologies for the IoT" funded by the Hungarian National Research, Development and Innovation Office. Ministry of Human Capacities, Hungary grant 20391-3/2018/FEKUSTRAT is acknowledged.

REFERENCES

[1] 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.

[2] P. Parmar and M. Patel, "Software fault localization: A survey," *International Journal of Computer Applications*, vol. 154, no. 9, pp. 6–13, 2016.

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

[4] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, oct 2016, pp. 267–278.

[5] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 199–209.

[6] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 314–324.

[7] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. New York, New York, USA: ACM Press, 2016, pp. 165–176.

[8] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 31:1–31:40, Oct. 2013.

[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] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems*, ser. DEFECTS '09. New York, NY, USA: ACM, 2009, pp. 1–5.

[11] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *28th international conference on Software engineering*, ser. ICSE '06. ACM, 2006, pp. 82–91.

[12] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 8:1–8:28, Feb. 2014.

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

[14] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. of International Conference on Automated Software Engineering*. ACM, 2005, pp. 273–282.

[15] X. Li, M. d'Amorim, and A. Orso, *Iterative User-Driven Fault Localization*. Cham: Springer International Publishing, 2016, pp. 82–98.

[16] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proceedings of the 40th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2018)*. ACM, 2018.

[17] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, "Interactive Fault Localization Using Test Information," *Journal of Computer Science and Technology*, vol. 24, no. 5, pp. 962–974, sep 2009.

[18] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *IEEE International Conference on Software Maintenance, ICSM*. IEEE, 2012, pp. 67–76.

[19] X. Xu, V. Debroy, W. E. Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, pp. 803–827, 2011.

[20] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 89–98.

[21] 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. ACM, 2017, pp. 273–283.

[22] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009.

[23] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 201–210.

[24] A. Riboira and R. Abreu, "The GZoltar Project: A Graphical Debugger Interface," in *Testing: Academia-Industry Collaboration, Practice and Research Techniques*. Springer, Berlin, Heidelberg, 2010, pp. 215–218.

[25] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 3, p. 1, jul 2013.

[26] A. Bandyopadhyay and S. Ghosh, "Tester feedback driven fault localization," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 41–50.

[27] Y. Lei, X. Mao, Z. Dai, and D. Wei, "Effective fault localization approach using feedback," *IEICE Transactions on Information and Systems*, vol. E95.D, no. 9, pp. 2247–2257, 2012.

[28] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 33–42.

[29] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on software engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[30] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 128–137.

[31] M. Papadakis and Y. Le Traon, "Metallaxis-FL: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, aug 2015.

[32] H. Jin, Y. Jiang, N. Liu, C. Xu, X. Ma, and J. Lu, "Concolic metamorphic debugging," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, July 2015, pp. 232–241.

[33] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. ACM, 2016, pp. 262–273.

[34] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[35] F. Chen and S. Kim, "Crowd debugging," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 320–332.

[36] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, pp. 393–403.

[37] J. Silva, "A survey on algorithmic debugging strategies," *Adv. Eng. Softw.*, vol. 42, no. 11, pp. 976–991, Nov. 2011.