Experiments with Interactive Fault Localization Using Simulated and Real Users

Ferenc Horváth*, Árpád Beszédes*, Béla Vancsics*, Gergő Balogh*, 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,beszedes,vancsics,geryxyz,lac,gyimi}@inf.u-szeged.hu

***Abstract*—Fault localization is considered a difficult and time consuming activity. However, tool support for automated fault localization is still limited because state-of-the-art algorithms often fail to provide efficient help to the user. They usually offer a ranked list of suspicious code elements, but the fault is not guaranteed to be found among the highest ranks. In Spectrum-Based Fault Localization (SBFL) – which uses code coverage information of test cases and their execution outcomes to calculate the ranks –, the developer has to investigate several locations before finding the faulty code element. Yet, all the knowledge she a priori has or acquires during this process is not reused by the SBFL tool. We propose an approach in which the developer interacts with the SBFL algorithm by giving feedback on the elements of the prioritized list. We exploit contextual knowledge of the user about the next item in the ranked list (e. g., a statement), with which larger code entities (e. g., a whole function) can be repositioned in their suspiciousness. First, we evaluated the approach using simulated users incorporating two types of imperfections, their knowledge and confidence levels. On SIR and Defects4J, results showed notable improvements in fault localization efficiency, even with strong user imperfections. We then empirically evaluated the effectiveness of the approach with real users, which also showed promising results.**

***Index Terms*—Spectrum-Based Fault Localization, automated debugging, interactive debugging, user feedback, user simulation, user imperfection, user study.**

I. Introduction

Debugging and related activities are among the most difficult and time consuming tasks in software engineering [1]. This activity involves human participation to a large degree, and many subtasks are difficult to automate. In this work, we address *fault localization*, a necessary subactivity in which the root causes of an observed failure are sought. Fault localization is notoriously difficult, and any (semi)automated method, which can help the developers and testers in this task, is welcome. There is a class of approaches to aid fault localization which are popular among researchers, but have not yet been widely adopted by the industry: Spectrum-Based Fault Localization (SBFL) [2]–[4].

Recent studies highlighted some barriers to the wide adoption of the SBFL methods, including a high number of suggested elements to investigate [5], [6], applicability of theoretical results in practice [7], little experimental results with real faults [4], validity issues of empirical research [8], and so on. Kochar *et al.* performed a systematic analysis of practitioner's expectations in the field [9]. With this paper, we aim at bringing closer the applicability of SBFL methods to practice by involving user's knowledge to the process.

The basic intuition behind SBFL is that code elements (statements, blocks, functions, etc.) that are exercised by comparably more failing test cases than passing ones are more suspicious to contain a fault. Additionally, non-suspicious code elements are traversed primarily by passing tests. Suspiciousness is usually expressed by assigning one value to each code element (the *suspiciousness score*), which can then 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. A possible approach to measure the effectiveness of a SBFL method is to investigate the average rank position of the actual faulty element (absolute or relative to the total number of code elements), i. e., the number of elements that have to be investigated before finding the fault (called the *Expense* measure). Later studies revealed that absolute Expense is crucial to the adoption of the method in practice. In particular, research showed that if the faulty element is beyond the 5th element (or 10th according to other studies), the method will not be used by practitioners because they need to investigate too many elements [4]–[6], [9]. A further problem is that there are no guarantees that any scoring mechanism will show sufficiently good correlation between the score and the actual faults [2], [4], [10], [11].

It seems that automatic SBFL methods require external information – not just the program spectra and test case outcomes – to improve on state-of-the-art performance and be more suitable in practical settings. In this work, we propose a form of an *Interactive Fault Localization* approach, called *iFL*. In traditional SBFL, the developer has to investigate several locations before finding the faulty code elements, and all the knowledge she a priori has or acquires during this process is not fed back into the SBFL tool. In our approach, the developer interacts with the fault localization algorithm by giving feedback on the elements of the prioritized list.

We build on the hypothesis that a programmer, when presented with a particular code element, in general has a strong intuition whether any other elements belonging to the same containing higher level code entity should be considered in fault localization. In other words, we exploit the knowledge of the user about the *context* of the next item in the ranked list; e. g., if the item is a statement the whole function is considered, or the whole class if the item is a function. This way, larger

code parts can be repositioned in their suspiciousness in the hope to reach the faulty element earlier. Other interactive approaches have been proposed by researchers as well [12]–[19], but to our knowledge, similar contextual information about higher level entities has not yet been leveraged.

We evaluated the approach in two experiments. First, we used simulated users and measured the Expense metric improvements with respect to traditional Tarantula SBFL [20]. We relied on two benchmarks: artificial defects from the SIR repository [21] and real defects from Defects4J [22]. We also modeled user imperfection, which was rarely studied in related interactive SBFL research. We addressed this aspect from two viewpoints: the user's knowledge and confidence. Experiments simulating these two factors show that *iFL* can outperform a traditional non-interactive SBFL method notably even at low user confidence and knowledge levels. Second, we invited students and professional programmers to solve a set of fault localization tasks using the implementation of the *iFL* approach in a controlled experiment. The goal was to find out whether using the tool shows actual benefits in terms of finding more bugs or finding them more quickly, and this also showed promising results.

In summary, our contributions are:

1) We introduced *iFL*, a novel context-aware interactive fault localization method, embedded in a flexible interactive fault localization framework.
2) We implemented a simulated user and performed experiments on both artificial and real faults. The latter has not yet been studied in interactive fault localization research.
3) We provide an analysis of two dimensions of user imperfection: knowledge and confidence, which was marginally addressed in previous literature.
4) We implemented *iFL* as an Eclipse plug-in that enables interactive fault localization on Java systems at method level granularity.
5) We performed an empirical study involving real users to compare the fault localization efficiency with and without using the *iFL* approach.

II. Motivating Example

For illustration, consider the example in Table I. This is part of program replace from the SIR benchmark repository, which includes manually seeded faults (this benchmark is often used in SBFL research, although being somewhat outdated). 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. The relevant code lines are shown in Table I. 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. The fourth column (*0. iteration*) of Table I corresponds to the suspiciousness scores computed by the Tarantula method along with the ranking position of the elements (the ranking position is arbitrary in the case of ties in the scores). There are several lines in functions getccl, omatch and locate that have higher scores than the faulty one from dodash, which will push it farther in the rank, in particular to the 11th-13th place (in the actual implementation, ties are handled so that the average position among the elements with the same value will be used, in this case 12th).

We can explain failing of SBFL in this case as follows. Recall the Tarantula formula for a code element s [20]:

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

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, and 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, which will be presented shortly). 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 [23] 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 relationship.

III. Interactive Fault Localization

Our approach to improve SBFL is to leverage the background and acquired knowledge of the developer about the system being debugged outside her current focus – the currently investigated code element. We build on the hypothesis that a programmer, when presented with a statement from a particular function, in general has a strong intuition whether any other statements in that function should be considered in fault localization. Or, in a different setting, the programmer is assumed to be able to decide (in certain cases) about the whole class being faulty or not, if presented with one of its methods. Example situations when such decisions could be made include when the element is known to have been reviewed or otherwise tested recently, it was examined in a previous debugging session, class members follow the same pattern such as getters-setters, etc.

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 SBFL 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

In our approach, we call this information the *contextual* knowledge, which can be fed back to the *iFL* engine. More precisely, we define the **context of an investigated code element** as **the other elements of its enclosing higher level syntactic entity**. For example, in the case of a statement, its context are all other statements belonging to its function. A context of a function is its enclosing class, and so on.

Suppose that a developer is performing SBFL and starts with the highest ranked element, statement 366 (see columns 4-7 in Table I). She looks at the function this statement belongs to and concludes that it is not likely to contain the fault. This knowledge is then fed back to the *iFL* 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. Then, the next most suspicious element is given to the user, statement 321 of function locate. Again, the developer decides based on contextual knowledge that this function is not suspicious, so the engine reduces scores of all contained statements to 0. This is repeated for line 145 as well in the next iteration. Consequently, several elements are pushed to the end of the list, moving the faulty one to the next rank position. This terminates the fault localization process with success. 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).

Figure 1 shows a conceptual overview of our approach. The process starts by calculating an initial rank based on some traditional SBFL approach like Tarantula. The elements are then shown to the user starting from the beginning of the list, and the *iFL* 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.

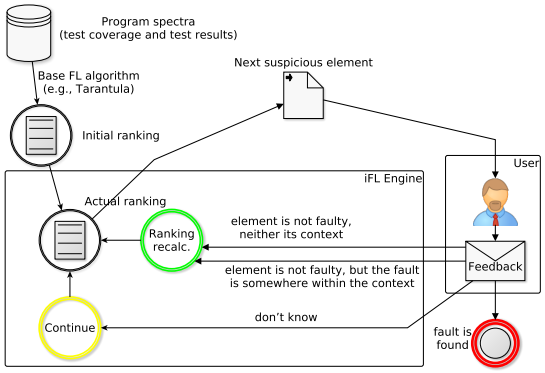

Fig. 1. Basic process of Interactive Fault Localization

Based on the feedback from the user, the *iFL* 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 SBFL approach). In the remaining two cases, the *iFL* 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.

There may be different strategies to make these adjustments,

such as applying proportional reductions or increases to the scores, which are different for the context and other parts of the system, etc. Presently, we follow this approach: in the case of (2), the whole context (i. e., function) gets 0 score, while for (3) everything but the context is reduced to 0.

IV. Evaluation Goals

We verified the effectiveness of the Interactive Fault Localization approach in two separate empirical studies: using *simulated* (Section V) and *real* users (Section VI). The study with simulated users enables large scale and automated experimentation with different faults from existing benchmarks, and predicting the expected effectiveness in real life scenarios. This approach has been followed by most of the related research, e. g. [12], [13], but we also perform measurements by simulating various degrees of user imperfection, which is a novelty compared to previous studies. On the other hand, evaluation with real users provides direct results about the usefulness of the approach, although only for a limited number of fault finding scenarios.

More precisely, the goal of the first part of the evaluation was the following. *With simulated users, how much improvement in localization effectiveness, in terms of elements to be inspected, can we achieve with iFL over a traditional non-interactive SBFL method?* We have the following Research Questions for this part of the evaluation:

RQ1 What improvement can we observe with *iFL* on artificial faults from the SIR repository?

RQ2 What improvement can we observe with *iFL* on real faults from the Defects4J repository?

RQ3 How sensitive *iFL* is to user imperfections?

The goal of the second part of the evaluation was the following. *Given actual fault finding tasks with real users, is it true that users with access to an implementation of iFL in their development environment are able to find more bugs or find them more quickly compared to a control group who did not have access to iFL?* We formulate the following Research Questions for the second part of the evaluation with real users:

RQ4 Is it true that users could find more bugs with *iFL* than users without access to the method?

RQ5 Is it true that users could find bugs more quickly with *iFL* than users without access to the method?

RQ6 How do real users subjectively evaluate the *iFL* method and its implementation in the development environment?

V. Results with Simulated Users

A. Experiment Setup

To answer research questions **RQ1**–**RQ3**, we relied on two sets of benchmarks: the SIR repository which contains mostly artificial faults and Defects4J, a benchmark consisting of real faults. These two benchmarks are different also in terms of their size and complexity so we will perform fault localization at different granularity levels. For SBFL, we selected the Tarantula method [20], which has been reported to be one of the most successful in different settings [4], and are often referred in literature. Regarding the user responses and SBFL engine actions, for **RQ1** and **RQ2** we follow a relatively simple but strict approach (there are no intermediate, partial or uncertain responses and actions, in other words, we simulate a *perfect* user). Of the four possible responses explained in Section III, we will not use the fourth one, "don't know". Furthermore, the mentioned strategy for the actions will be employed, that is, reducing either the whole context or everything but the context to 0. Experiments of user imperfection that answer **RQ3**, including the "don't know" answer, are presented separately in Section V-E.

We implemented the required components of the *iFL* system according to these settings on different granularities for the two benchmarks, and executed it using all available bugs. The simulated user component works so that it takes the elements from the ranked list starting from the first one, compares their context to the context of the known fault and generates the corresponding answers until the faulty element is reached.

B. Evaluation Method

For measuring the effectiveness of fault localization, in this study we follow the strategy to look at "elements that need to be investigated" using the "expected case" in the case of ties (other approaches are available as well [2], [24]) We express this in a set of measures called *Expense*, with two variants: an absolute one expressed in the number of code elements (E) and a relative version compared to the length of the rank list (E'). Parnin and Orso argued that absolute rankings are more helpful in practical situations [6]. The following formulae express precisely how to calculate this value (following [25]):

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

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 a traditional SBFL approach, we will compute Expense metrics for both approaches and compare them in terms of improvement relative to traditional SBFL. Since in each iteration of the approach one block of code is decided upon in one step, we will count each iteration 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.

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 [5], [9]. Hence, any improved rank position which is beyond these thresholds will probably be less useful, no matter how much relative improvement they can achieve. Therefore, we define the notion of *enabling improvement*, an improvement in which the traditional FL algorithm ranks the faulty method at a position larger than 10 (or 5), but the *iFL* method reaches the faulty method in less than 10 (or 5) steps. This way from a practically "hopeless" localization scenario our method enables the user to localize the fault by inspecting

the top elements in the list (the *accuracy* measure by Sohn and Yoo [26] is similar). We will use three concrete cases to express enabling improvement:

- $(10, \infty] \rightarrow (5, 10]$ The base FL score is larger than 10 and *iFL* reaches the fault in 5 to 10 steps.
- $(10, \infty] \rightarrow [1, 5]$ The base FL score is larger than 10 and *iFL* reaches the fault within 5 steps.
- $(5, 10] \rightarrow [1, 5]$ The base FL score is between 5 and 10 and *iFL* reaches the fault fault within 5 steps.

C. Results for Seeded Faults

To answer **RQ1**, seven small C/C++ programs from the Software-artifact Infrastructure Repository (SIR) [21] were included in the experiments, which are the so-called "Siemens" suite. This benchmark contains seeded faults, 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 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 the coverage tool was unable to record coverage in, e. g., headers and macros; 3) we omitted cases where the suspiciousness score of the faulty code element assigned by the actual SBFL technique was zero (these could not be further improved). For preparing the raw data for the *iFL* experiments including the code coverage information and test case results, the tools GCOV [27] and SoDA [28] were used.

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**

Table IV shows the improvements *iFL* was able to achieve on SIR. The performance of the original SBFL algorithm can be seen in column 3, which we used as the reference to evaluate *iFL*. Both absolute and relative versions of the Expense measure are provided. On average, Tarantula prioritized the faulty code elements roughly to the 25th place (24.85), which means that on average 15.43% of the code elements must be examined to find the faulty one.

Column 4 contains the same data for *iFL*. The average Expense measures are notably better than for the original algorithm. This means that in this case a programmer would need only about seven (6.86) steps to find the fault on average. Column 5 shows the actual difference between the absolute and relative Expense measures, which is 17.99 (11.19%). Column 6 of the table contains a summary of improvements in terms of relative changes in the Expense values, expressed in percentage (that is, the difference over the SBFL base value): the improvement is notable, **72.42%**.

The last four columns of Table IV summarize the *enabling improvements iFL* achieved on the SIR benchmark. Here, the number of faults (and their relative ratio) are presented falling in the three categories of enabling improvements. According to the last column, the total ratio of improvements that turned out to be enabling is quite large, **49.41%**. More importantly, most of these improvements are those that bring the faulty code element from outside of top 10 into top 10 (Column 7) or, even better, into top 5 (Column 8). The two programs on which *iFL* produces the highest rate of enabling improvements are tcas and replace. Interestingly, tcas is the smallest and replace is the largest program in our set, which may indicate that there is no connection between improvement rate and program size.

Answer to RQ1: In the case of SIR programs containing seeded faults, *iFL* achieved **72.42% improvement** in Expense, and resulted in **42 enabling improvements**, which corresponds to **49.41% of the faults**.

D. Results for Real Faults

In the field of Interactive Fault Localization, there is an emerging need for studies that go beyond the size and complexity of the SIR repository. Here, we present our measurement results with *iFL* on defects from the Defect4J repository (v1.1.0) [22], to answer **RQ2**.

The dataset contains 6 open source Java programs and 395 bugs in total. For *iFL* experiments, we used Clover [29], Apache Maven [30] and SoDA [28]. Clover is an open source code coverage measurement tool which we used to extract coverage and results data from Java systems in which Maven is used as the software project management tool. Since the JFreeChart, the Closure Compiler and the Mockito projects are not Maven based they were excluded from our experiments. Defects4J provides the fix for each bug as a patch set. We used the SourceMeter source code analyzer tool [31] to get the location of all methods in each program. Then, using the patch sets and the information provided by the static analyzer we were able to create change sets that contain data about which methods were affected by which bug fixes. The minimum requirement of SourceMeter is Java 8, hence older versions are omitted from our experiments. As with the other benchmark, we considered only single method faults, and those faults where the suspiciousness score was not 0. The final set of programs and defects from the Defects4J dataset can be seen in Table V. The last column includes the number of bugs we could use in the experiments.

TABLE IV
iFL IMPROVEMENTS ON SIR

Program	Faults	$E(E')$ Avg rank	$E(E')$ Avg rank w *iFL*	$E(E')$ Diff.	Impr.	Enabling improvements $(10,\infty] \rightarrow (5,10]$	Enabling improvements $(10,\infty] \rightarrow [1,5]$	Enabling improvements $(5,10] \rightarrow [1,5]$	Total
printtokens	1	5.00 (1.81%)	2.00 (0.72%)	-3.00 (-1.08%)	60.00%	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
printtokens2	7	30.71 (11.72%)	7.21 (2.75%)	-23.50 (-8.97%)	76.51%	1 (14.29%)	0 (0.00%)	0 (0.00%)	1 (14.29%)
replace	22	19.18 (4.80%)	4.70 (1.18%)	-14.48 (-3.62%)	75.47%	2 (9.09%)	6 (27.27%)	5 (22.73%)	13 (59.09%)
schedule	2	10.75 (4.78%)	5.50 (2.44%)	-5.25 (-2.33%)	48.84%	1 (50.00%)	0 (0.00%)	0 (0.00%)	1 (50.00%)
schedule2	4	77.38 (39.02%)	14.25 (7.18%)	-63.12 (-31.84%)	81.58%	0 (0.00%)	1 (25.00%)	0 (0.00%)	1 (25.00%)
tcas	31	21.65 (22.78%)	5.58 (5.87%)	-16.06 (-16.91%)	74.22%	9 (29.03%)	11 (35.48%)	1 (3.23%)	21 (67.74%)
totinfo	18	26.00 (13.90%)	10.33 (5.53%)	-15.69 (-8.39%)	60.36%	4 (22.22%)	0 (0.00%)	1 (5.56%)	5 (27.78%)
	85	**24.85 (15.43%)**	**6.86 (4.25%)**	**-17.99 (-11.19%)**	**72.42%**	**17 (20.00%)**	**18 (21.18%)**	**7 (8.24%)**	**42 (49.41%)**

TABLE V
MAIN PROPERTIES OF PROGRAMS USED FROM DEFECTS4J (KLOC, TESTS AND NO. BUGS COLUMNS DATA FROM [22])

Project name	KLOC	Tests	Avg. no. methods	Avg. no. classes	No. bugs	No. suitable bugs
Commons Lang	22	2 245	2 045	101	65	8
Commons Math	85	3 602	4 753	394	106	69
Joda-Time	28	4 130	4 066	179	27	18
Total	**135**	**9 977**	**10 864**	**674**	**198**	**95**

In this experiment, the granularity of fault localization was elevated to the method level because of two reasons. First, we could use statement level granularity as well, but the benchmark contains larger and real programs, and even on method level it includes a large number of code elements. Second, we wanted to check how does the algorithm behave on this level and if there is a significant difference in terms of effectiveness to the other benchmark. Our feedback-based algorithm needed adjustment as well: the basic elements are changed from source code lines to methods, and the context is changed from functions to classes. Otherwise, the main steps of the *iFL* process (from Figure 1) including the responses and actions were the same as with the statement-level granularity. The measurements themselves followed the same steps as we used for SIR in Section V-C, and the results will be presented in the same way in this section as well. Therefore, detailed explanation of the structure of tables will be omitted.

Table VI contains the associated results. The Expense measure for the original SBFL method is quite different than for the SIR programs, it is 42 steps on average, which is much more than the average on small programs, however, the relative measure is smaller, 0.95%. This is due to the significantly larger number of program elements in this benchmark (despite the higher granularity level). *iFL* achieved a notable improvement with this benchmark as well, as can be seen from columns 4 and 5 of the table. The difference is 33, but given the large total number of elements, the change in percentages is modest. However, the relative improvement (column 6) is equally large as for SIR, even a bit higher, **78.45%**. Practically, this means that on average the *iFL* approach could potentially save 78.45% of the human effort.

More importantly, in the case of large programs and real defects there are many cases when *iFL* achieved *enabling improvements*. Detailed data is shown in the second part of Table VI. Overall, *iFL* had **33 (35%)** enabling improvements, which is slightly worse than for the SIR programs. In most cases, *iFL* brings the faulty elements into the top-10 or top-5 range from outside of top-10. These are the cases where the original SBFL produced very bad Expense results initially. Compared to SIR, the lower number of test cases may be one reason for this phenomenon, but finding the actual causes needs more investigation. Note that, this benchmark contains much larger programs and that the original Expense measures were typically much higher as well.

Answer to RQ2: In the case of Defects4J experiments with real faults, *iFL* achieved **78% improvement** in Expense, and produced **33 enabling improvements**, corresponding to **35% of the faults in this benchmark**.

E. Effect of User Imperfections

To answer **RQ3**, in this section we investigate to what extent user imperfection affects the results of our method. This phenomenon is only marginally addressed in interactive fault localization literature. Hao *et al.* [13] tackled the problem of user imperfection by incorporating two factors into their approach and analysis. They introduce a parameter which approximates the confidence of developers by acting as a scaling factor on suspiciousness modifications. Also, they define the concept of accuracy rate to represent the probability that the developer makes correct estimations. Li *et al.* [16], [17] used a similar approach to simulate the reliability of the users by modifying the automated oracle in their experiment such that it gives erroneous answers on a configurable rate. We provide a similar, but more detailed analysis of user imperfection by experimenting with two factors that may influence the validity of simulated users:

Confidence Level: This factor indicates how much confidence we have in the user for providing reliable answers. We model confidence by applying a proportional decrease of the scores instead of nullation as with the base algorithm. The *iFL* engine was modified to scale down the suspiciousness of appropriate code elements proportionally to the confidence level: the new score s' is calculated from the original s using confidence level c as: $s' = (1 - c)s$. Here, $c = 0$ means no confidence in which case the original scores remain, and with perfect confidence, $c = 1$, nullation will be performed.

Knowledge Level: In our model, the knowledge of the user means the rate at which she can make informed decisions

TABLE VI
iFL IMPROVEMENTS ON DEFECTS4J

Program	Faults	$E(E')$ Avg rank	$E(E')$ Avg rank w *iFL*	$E(E')$ Diff.	Impr.	Enabling improvements $(10,\infty] \rightarrow (5,10]$	Enabling improvements $(10,\infty] \rightarrow [1,5]$	Enabling improvements $(5,10] \rightarrow [1,5]$	Total
commons-lang	8	4.38 (0.22%)	2.38 (0.12%)	-2.00 (-0.10%)	45.71%	1 (12.50%)	0 (0.00%)	1 (12.50%)	2 (25.00%)
commons-math	69	29.33 (0.57%)	7.33 (0.15%)	-22.01 (-0.41%)	75.02%	12 (17.39%)	6 (8.70%)	9 (13.04%)	27 (39.13%)
joda-time	18	109.56 (2.77%)	19.11 (0.48%)	-90.44 (-2.29%)	82.56%	4 (22.22%)	0 (0.00%)	0 (0.00%)	4 (22.22%)
	95	**42.43 (0.95%)**	**9.14 (0.21%)**	**-33.29 (-0.74%)**	**78.45%**	**17 (17.89%)**	**6 (6.32%)**	**10 (10.53%)**	**33 (34.74%)**

about the context. Thus, knowledge is modeled by the user's ability to give meaningful answers about the context as a whole. This factor was implemented by letting the user choose the "don't know" response randomly with a frequency that is inversely proportional to the knowledge level. This means that a perfect knowledge, $k = 1$, allows no "don't know" responses, while with no knowledge at all, $k = 0$, every answer will be of this type, falling back to the base FL algorithm.

If either of the two factors is 0, the method will fall back to the traditional FL approach, while if both are 1, we will obtain the base approach we used for research questions **RQ1**-**RQ2**. Any combination of values in between are interesting to observe to what extent they are influencing the effectiveness of the *iFL* method. We re-executed our experiments on both datasets with confidence and knowledge levels set between 20% and 100% in 10% steps. We decided to ignore values below 20% because they simulate an unlikely situation in which the user is very incompetent. Due to the random factor that was introduced by the implementation of the knowledge level, we repeated each measurement 100 times and used the average data that was collected during the iterations.

Improvement levels, in terms of absolute Expense difference, can be seen in Figure 2 for the SIR benchmark (results were similar for Defects4J but due to space constraints we are not presenting them here). Each point on the 3D surface represents a different configuration of knowledge and confidence. The near and far edges of the 3D cube mean the perfect and almost completely incompetent users, respectively (note, that values for configurations of less than 20 are not shown, which tend towards 0).

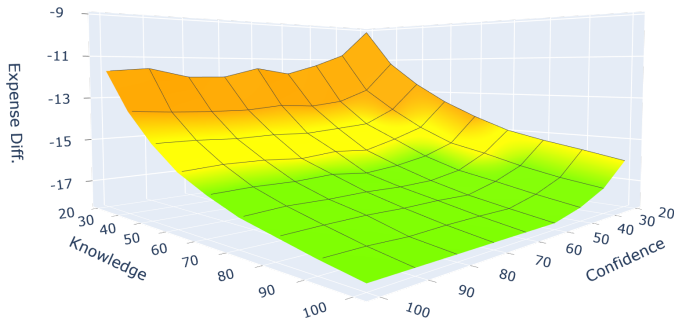

Fig. 2. Improvement with different knowledge and confidence levels on SIR

Results show that both knowledge and confidence affect the performance of *iFL*, but to various extent. *iFL* seems to be relatively stable, because both factors change only a small amount from the perfect user to the very low 20% knowldge and confidence levels. Low knowledge level seems to have a bigger influence on the results than confidence, which can be observed by comparing the left and right hand side edges of the surface. This seems to be aligned with the everyday observation that high confidence combined with low knowledge is a worse situation than a low confidence with high knowledge scenario. These results indicate that relatively low levels of knowledge and confidence are sufficient in order to preserve the accuracy of the approach at an almost perfect level. We can observe from Figure 2, in particular, that when both aspects of user imperfection exceed the roughly 30-40% levels, the overall gain will be above 80% of the gain of the perfect user (14.4 instead of 17.99).

Answer to RQ3: User imperfection, as modeled by our experiments, only marginally affects the results of the context aware *iFL* algorithm. Our experiments show that even very low confidence (**20-30%**) and knowledge levels (**30-40%**) suffice to keep **80%** of the improvements.

VI. RESULTS WITH REAL USERS

A. Experiment Setup

To answer research questions **RQ4**-**RQ6**, we performed a user study involving real programmers and asking them to solve real fault localization tasks within an IDE. For the experiment setup, we reused parts of the methodology followed by Parnin and Orso in their 2011 article [6].

Participants: 36 software engineering students were invited for participation on a voluntary basis, of which 22 were BSc (undergraduate) and 14 were MSc (graduate) students. 23 participants had at least 1 year programming experience working in industry. Only the top-performing students were invited from the class of about 230 based on their previous scholastic performance in the relevant subjects. Only students with sufficient knowledge of Java, Eclipse, and its debugging features were included. The programming experience of the participants was between 0.5–6 years, on average 2.5 years. Three groups ($G1 - G3$) have been formed randomly, each having approximately the same ratio of BSc and MSc students.

To diversify the experiment we also invited professional programmers from the software development teams of our university. We excluded everyone who had some relation to the topic of this paper and from the 4 volunteers we created group $G4$. The average programming experience in this group was 12.75 years (between 5–18 years).

Tool Support: We implemented a prototype tool [32], [33] as an Eclipse plug-in that implements the basic functionality of *iFL*. Currently, it supports Java projects and fault localization on method granularity. It provides in a window a ranked list of methods with the associated suspiciousness scores, the context (enclosing class), and other information. The user can interact with this list, provide the feeedback, make filtering on the scores, navigate to the source, etc.

Task Assignment: We designed altogether 8 different fault localization tasks ($A - H$), keeping in mind that participants should be able to solve each task in about 30 minutes including understanding the problem and documenting the solution (it was treated successful if the participant can briefly explain the required fix but no actual implementation was needed). We also wanted to ensure some diversity, so we selected A and E to be small, the rest large programs, B, E, F be simple bugs and the rest more complex, some to have low Tarantula ranks (A, C, D), and the rest high ranks. This information was not told to the participants.

We selected 6 concrete bugs from the Defects4J database (3 from Commons Math and 3 from Joda-Time) and 2 bugs from a student projectIn particular, the eight bugs with their ID-s, names and Tarantula ranks were the following: [A] ship-3 (rank 11), [B] math-5 (rank 2), [C] time-9 (rank 7), [D] time-8 (rank 7), [E] ship-1 (rank 2), [F] math-53 (rank 1), [G] time-4 (rank 5), [H] math-4 (rank 2).

Groups $G1 - G3$ have each been assigned 4 different tasks from the 8 available, half of which had to be executed using the *iFL* functionality and the rest without it (feedback was disabled). In order to increase the diversity, we assigned the tasks and tool modes in various combinations to the groups. We reserved two more complex tasks for group $G4$, and tool modes were randomly selected when the participants started their tasks. Table VII shows the task assignments (I: *iFL* used, N: *iFL* not used).

TABLE VII
TASK ASSIGNMENT

Group / Task	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
$G1$	N	I	-	N	I	-	-	-
$G2$	-	N	I	-	N	I	-	-
$G3$	I	-	N	I	-	N	-	-
$G4$	-	-	-	-	-	-	I/N	I/N

Experiment Execution: The experiment was executed in two sessions with three 1/2 hour blocks in each session and a break between the sessions. The first block was dedicated to introduce participants to the experiment, explain the goals, the basic functionality of the tool and other instructions. Then, the four tasks have been performed in the next blocks, where 25 minutes were available to actually perform the task and 5 minutes were reserved for documenting the results and switching to the next task. After each task and for each participant, we recorded the following information: number of minutes for completion if it was successful, the solution, how much was the tool used (*none, little, fully*), how much did it help (*none, little, a lot*), if the tool was not used what method was used instead to find the bug. During the final 30 minutes, the participants were asked to fill a questionnaire about their general impressions and comments: how useful was the approach (on a scale 1-5), actual benefits and drawbacks encountered, further information that could be used as the context and other ideas to improve the tool.

B. Results for Bug Finding Efficiency

Results regarding the number of completed tasks are presented in Table VIII.[1] For each task, we include the total number of participants who performed it, using *iFL* support and without it (columns 2, 4 and 6, respectively).[2] Columns *Compl.* show the number of participants who successfully completed the task belonging to the group using the tool and not using it, respectively (also, percentages are given wrt. the number of participants in the corresponding groups).

TABLE VIII
NUMBER OF ALL AND COMPLETED TASKS

Bug	Overall		with *iFL*		no *iFL*	
	All	Compl.	All	Compl.	All	Compl.
A - ship-3	24	3 (12%)	12	2 (17%)	12	1 (8%)
B - math-5	24	19 (79%)	12	9 (75%)	12	10 (83%)
C - joda-9	24	14 (58%)	12	5 (42%)	12	9 (75%)
D - joda-8	24	18 (75%)	12	8 (67%)	12	10 (83%)
E - ship-1	25	15 (60%)	13	8 (62%)	12	7 (58%)
F - math-53	23	23 (100%)	12	12 (100%)	11	11 (100%)
Total	144	92 (64%)	73	44 (60%)	71	48 (68%)

The overall success is 64% but it is highly variable across the different tasks. We could not observe any dependence on the program size (A vs. D), but more simpler bugs could be localized by more participants (B vs. E), overall. Initially, it was not our intent, but task A turned out to be the most difficult to solve by the participants. Besides the relative complexity of the bug, this might also be influenced by the fact that it was the first assignment for the participants, who perhaps did not have enough understanding of the tool at that time.

We could not observe any difference in the success rate of participants who used *iFL* compared to those who did not. In particular, the number of participants who successfully solved the tasks is approximately the same, there is even a slight increase in the overall number of cases without tool support. A very slight improvement within the group using the tool can be observed for tasks A (small but difficult bug), E (complex bug) and F (simple bug and high Tarantula rank).

Answer to RQ4: *iFL* does not seem to help in localizing more bugs. A slight increase in success rate can be observed in the case of complex bugs and when the Tarantula rank is very high.

[1] In group $G4$ success rate was 100% for both tasks, and the completion time varied between 10-29 minutes independently from tool usage, but due to space limitations and the very low number of participants we excluded this group from the evaluation of effectiveness and efficiency.

[2] For E and F, number of group members with and without *iFL* was not equally 12+12 because two workstations had to be exchanged due to technical reasons, which resulted in a slight change to the planned task assignment.

TABLE IX
TASK COMPLETION TIME IN HOURS, MINUTES AND SECONDS

Bug	with *iFL*	no *iFL*	Diff.
A - ship-3	0:16:00	0:25:00	-0:09:00 (-36%)
B - math-5	0:10:33	0:05:48	0:04:45 (82%)
C - joda-9	0:08:00	0:14:53	-0:06:53 (-46%)
D - joda-8	0:15:15	0:21:06	-0:05:51 (-28%)
E - ship-1	0:16:07	0:19:17	-0:03:09 (-16%)
F - math-53	0:12:40	0:09:16	0:03:23 (37%)
Total	**9:30:00**	**11:05:00**	**-1:35:00 (-14%)**

Table IX shows the results we collected about the time required to localize the fault, which was needed for **RQ5**.

For each bug, we present the average times required for completion over all group members who managed to complete the task. The last column shows the difference of the times (absolute and relative) with respect to the cases without tool support. We could observe a noticeable overall improvement, of **14%**. But, results also show that there is a big variance of the difference across the different tasks: in the case of the Commons Math bugs, B and F, the tool even resulted in longer completion times, but in the other cases there was 16-46% improvement on average. Both Commons Math bugs were quite easy to understand and to locate the faulty element in them, so it might be the case that the use of *iFL* resulted in such a big overhead that not using the tool was actually more simpler. The overall improvement excluding these two tasks was about **36%** (5:23:00 vs. 8:25:00 total times). We could not observe any relationship between the program size or Tarantula rank and the completion times.

Answer to RQ5: Using *iFL* reduced the time required to localize the fault, overall by **14%**, except for the cases when the bug was very simple and easy to identify (without these, the improvement was **36%**).

C. Subjective Evaluation by the Participants

For answering **RQ6**, participants were asked to fill out a questionnaire that consisted of two parts: short questions about each bug and questions about the approach and tool in general. Table X includes the data for the first part, the responses per bug (this relates only to tasks where *iFL* was enabled). In more than two-thirds of all cases, participants expressed their opinion regarding the usage and usefulness of *iFL*.

TABLE X
SUBJECTIVE EVALUATION OF *iFL* PER BUG

Bug	Usage				Usefulness			
	no answ.	none	little	fully	no answ.	none	little	a lot
A - ship-3	6	0	3	3	6	2	3	1
B - math-5	2	2	6	2	3	1	6	2
C - joda-9	5	2	5	0	5	2	3	2
D - joda-8	5	2	2	3	6	2	1	3
E - ship-1	3	1	5	4	3	2	5	3
F - math-53	1	0	8	3	1	0	7	4
Total	**22 (30%)**	**7 (10%)**	**29 (40%)**	**15 (20%)**	**24 (33%)**	**9 (12%)**	**25 (34%)**	**15 (21%)**

Users responded that they did not use the tool in 7 cases (10%); they used it a little in 29 cases (40%); and relied fully on *iFL* in 15 cases (20%). Overall, participants used the interactive approach at least a little in **44 cases (60%)**. Considering the usefulness of *iFL*, the results were similar. Participants did not find the approach helpful at all in 9 (12%) cases, but in the remaining 25 (34%) and 15 (21%) cases they found that *iFL* aided fault localization at least a little or a lot, respectively (this is **82%** of the cases when participants responded). Experts also agreed on the usefulness of the tool, but they argued that the complexity of the bugs may have an impact on it. However, we did not identify any pattern in the distribution of opinions wrt. differences in bug types.

In the next part of the survey, participants were asked to rate the usefulness of the interactivity for SBFL in general on a scale 1–5 (*1-not useful*, *5-extremely useful*). The results are presented in Fig. 3. **Two thirds of the participants (24)** said that *iFL* was useful at least moderately and only 8.3% (3) answered that they did not find it helpful at all.

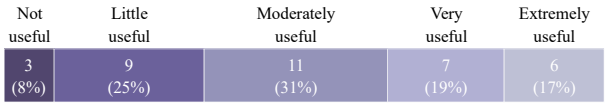

Fig. 3. Overall usefulness of the *iFL* approach

Participants could also write a textual evaluation of the approach and the tool, in which they could list the advantages and disadvantages they experienced. Some typical benefits mentioned: "The tool gives good hints and it can confirm if my idea is good or not", "It is straightforward to navigate between suspicious functions". Some disadvantages of the tool mentioned by participants: "The tool can mislead and so gives an unnecessary overhead", "We can exclude the actually faulty functions with feedback, which cannot be undone".

In addition, participants could articulate suggestions for using or further developing the tool. Several commented that it would be helpful if the selected method could be automatically opened in a separate window or a view. Also, undo-redo and the dynamic score-update were among the most frequently mentioned missing features. Some commented that for a large set of methods, the traditional search function made it easier to find the appropriate methods.

Professional developers said that the tool provides good starting points for debugging and it also helps focusing their efforts on the most suspicious parts of the source code. However, they also added that more information would be beneficial to make full use of the potential of interactivity and to make decisions about contexts easier. They mentioned the visualization of factors that contribute to the suspiciousness scores (e. g., related tests, especially the failing ones) and provide more detailed information about the bug (e. g., stack traces, call-chains, etc.) in an organized, easily understandable way as the most advantageous improvement.

Answer to RQ6: For most tasks where they responded, participants found interactive feedback useful to find the bug (**82%**), and **two thirds** of the participants said that in general it was useful at least at a moderate level. Textual responses about the benefits and drawbacks will help us in further developing the approach and tool.

VII. Related Work

SBFL methods are still finding their way to be employed in practice [7]–[9], [34]. For instance, most studies are carried on using artificial faults [4], and still the faulty element is usually placed far from the top of the rankings [5], [6]. Le *et al.* showed in their study that there is a gap between theoretical and practical results [7]. Since SBFL heavily relies on the coverage and the pass/fail information, test suite properties directly affect fault localization [35], [36]. The Zoltar [37] and GZoltar [38], [39] tools provide the ranked list of diagnosis candidates to help the user in practice.

The closest related works to our approach are the ones that change the ranking of program elements based on the user feedback iteratively. However, the setup of experiments and metrics used for the evaluation are different in every case. Differences include the set of defects, the total number of code elements, different interpretation of the localization effectiveness metrics, etc. This makes it difficult to compare our results directly to the reported ones in these works.

For reference, Lei *et al.* [15] utilize test data generation techniques to automatically produce feedback for interacting with fault localization techniques. They used a very similar metric to ours (E') to measure the relative effectiveness improvement and concluded that the improvement is around 21% on average compared to the 71-72% range achieved by *iFL*. Hao *et al.* [13] propose a trace-based method which is reported to achieve about 8% in a similar measure; they also showed that about 90% accuracy from the user is needed to improve the base SBFL algorithm.

In the work of Gong *et al.* [12], the user simply decides whether the statements are faulty or not. Ranking is updated to find the root cause of the fault using the program spectra. Their approach yields about 12-13% absolute improvement in Expense on average over Tarantula and Ochiai on small programs from the SIR repository [34].

Li *et al.* uses a concept of contextual knowledge that is similar to ours [16], [17]. They build on the assumption that the semantics of a method wrt. inputs and output is well known by developers. They generate queries and use the feedback to guide the SBFL based recommendation process in a debugging scenario. Also, they considered the correlation between the success rate of their approach and the percentage of erroneous answers to these queries. Bandyopadhyay and Ghosh proposed a method to iteratively predict and remove coincidentally correct test cases based on user feedback [14].

Fry and Weimer [40] used software- and defect-related features to study human accuracy at locating faults. They found that certain types of bugs are much harder for humans to locate accurately. Also, they identified source code features that can foretell human FL accuracy and proposed formal models of debugging accuracy based on these features.

We concentrated on statistical analysis of dynamic test case executions, but there have been other approaches proposed for fault localization as well. For details and comparison of these approaches we refer to the surveys of Wong *et al.* [2], [41] and Parmar and Patel [3].

Some debugging approaches are loosely related to the topic of this article, where user feedback may be incorporated, for example the works of Zeller *et al.* on Delta Debugging [42], [43], as well as algorithmic debugging and testing [44].

VIII. Conclusions

In this work, we presented *iFL* – Interactive Fault Localization, an approach to extend traditional Spectrum-Based Fault Localization by providing the ability for the developer to interact with the fault localization algorithm. Interaction means giving feedback on the elements of the prioritized list, based on which the suspiciousness scores are adjusted. We exploit the knowledge of the user about the next item in the ranked list (e. g., a statement or a function) and its context (the containing function or class), with which larger amounts of code elements can be repositioned in their suspiciousness.

In the present phase of the research, we used simulated users and a limited empirical study with real users using an experimental tool implementation, which might not represent real life scenarios fully. However, the empirical results with simulated users showed quite big improvements with respect to non-interactive SBFL even in the case when the users exhibit low levels of reliability. The study with real users also showed promising results. Although, *iFL* does not seem to help in localizing more bugs, the fault localization times were reduced significantly (except for the very simple cases), and subjective feedback was also mostly positive about our experimental implementation. Current results indicate that there is a huge potential in improving the interaction between developers and FL algorithms. The improvement of the score updating mechanism could also result in even stronger results. Therefore, we plan to perform more comprehensive empirical experiments with real programmers, more bugs, and other strategies for the adjustment of suspiciousness scores. We would also like to investigate the assumption about developers' contextual knowledge in our future work.

The interested reader can find more information about the data and software used in this paper on GitHub [45] and in the following archives: [46], [47].

Acknowledgements

We would like to thank Rita Bártfai and Dávid Horváth for their contribution to the development of the *iFL* for Eclipse.

This research was supported by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary. Árpád Beszédes and Ferenc Horváth were supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things. László Vidács was also funded by the János Bolyai Scholarship of the Hungarian Academy of Sciences. Dávid Horváth was supported by project EFOP-3.6.3-VEKOP-16-2017-0002, co-funded by the European Social Fund.

REFERENCES

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

[2] 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.

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

[4] 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.

[5] 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.

[6] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 199–209.

[7] T.-D. B. Le, F. Thung, and D. Lo, "Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 380–383.

[8] 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*. ACM, 2013, pp. 314–324.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *Proceedings of the 39th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 393–403.

[19] D. Lehmann and M. Pradel, "Feedback-directed differential testing of interactive debuggers," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 610–620.

[20] 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.

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

[22] 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*. ACM, 2014, pp. 437–440.

[23] 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.

[24] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*. IEEE Computer Society, 2003, pp. 30–39.

[25] 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.

[26] 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.

[27] "gcov-—a test coverage program," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, last visited: 2020-08-13.

[28] "SoDA library," https://github.com/sed-szeged/soda, last visited: 2020-08-13.

[29] "Atlassian Clover," https://www.atlassian.com/software/clover, last visited: 2020-08-13.

[30] "Apache Maven," https://maven.apache.org, last visited: 2020-08-13.

[31] "SourceMeter," https://www.sourcemeter.com, last visited: 2020-08-13.

[32] G. Balogh, V. Schnepper Lacerda, F. Horváth, and Á. Beszédes, "iFL for Eclipse – a tool to support interactive fault localization in Eclipse IDE," 12th IEEE International Conference on Software Testing, Verification and Validation (ICST'19), Tool Demo Track, Apr. 2019.

[33] G. Balogh, F. Horváth, and Á. Beszédes, "Poster: Aiding Java developers with interactive fault localization in Eclipse IDE," in *Proceedings of the 12th IEEE Conference on Software Testing, Verification and Validation (ICST'19), Posters Track*, Apr. 2019, pp. 371–374.

[34] 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.

[35] 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.

[36] 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.

[37] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: A toolset for automatic fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 662–664.

[38] 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.

[39] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. ACM Press, 2012, p. 378.

[40] Z. P. Fry and W. Weimer, "A human study of fault localization accuracy," in *2010 IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10.

[41] W. E. Wong and V. Debroy, "A survey of software fault localization," *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, vol. 9, 2009.

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

[43] A. Kiss, R. Hodován, and T. Gyimóthy, "Coarse hierarchical delta debugging," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 194–203.

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

[45] "iFL 4 Eclipse," https://github.com/sed-szeged/iFL4Eclipse.

[46] F. Horváth, Á. Beszédes, B. Vancsics, G. Balogh, L. Vidács, and T. Gyimóthy, "Data for Experiments with Interactive Fault Localization Using Simulated and Real Users," 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3991763

[47] ——, "Supplemental Material for Experiments with Interactive Fault Localization Using Simulated and Real Users," 2020. [Online]. Available: https://doi.org/10.6084/m9.figshare.c.5099597.v1