



# Using contextual knowledge in interactive fault localization

Ferenc Horváth<sup>1</sup> · Árpád Beszédes<sup>1</sup> · Béla Vancsics<sup>1</sup> · Gergő Balogh<sup>1</sup> ·  
László Vidács<sup>1,2</sup> · Tibor Gyimóthy<sup>1,2</sup>

Accepted: 26 May 2022 / Published online: 6 August 2022  
© The Author(s) 2022

## Abstract

Tool support for automated fault localization in program debugging is 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. There are existing approaches in which the developer interacts with the SBFL algorithm by giving feedback on the elements of the prioritized list. We propose a new approach called *iFL* which extends interactive approaches by exploiting 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. We implemented a closely related algorithm proposed by Gong *et al.*, called TALK. First, we evaluated *iFL* using simulated users, and compared the results to SBFL and TALK. Next, we introduced two types of imperfections in the simulation: user's 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 in two sets of experiments: a quantitative evaluation of the successfulness of using *iFL*, and a qualitative evaluation of practical uses of the approach with experienced developers in think-aloud sessions.

**Keywords** Spectrum-based fault localization · Automated debugging · Interactive debugging · User simulation · User study · Think-aloud sessions

---

This article belongs to the Topical Collection: *Software Maintenance and Evolution (ICSME)*

Communicated by: Zhenchang Xing and Kelly Blincoe

✉ Ferenc Horváth  
hferenc@inf.u-szeged.hu

Extended author information available on the last page of the article.

# 1 Introduction

Debugging and related activities are among the most difficult and time consuming tasks in software engineering (Vessey 1986). 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) (Wong et al. 2016; Parmar and Patel 2016; Pearson et al. 2017).

Recent studies highlighted some barriers to the wide adoption of the SBFL methods, including a high number of suggested elements to investigate (Xia et al. 2016; Parnin and Orso 2011), applicability of theoretical results in practice (Le et al. 2013), little experimental results with real faults (Pearson et al. 2017), validity issues of empirical research (Steimann et al. 2013), and so on. (Kochhar et al. 2016) performed a systematic analysis of practitioner's expectations in the field. 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 an 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 (Pearson et al. 2017; Xia et al. 2016; Parnin and Orso 2011; Kochhar et al. 2016). 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 (Wong et al. 2016; Pearson et al. 2017; Xie et al. 2013; Yoo et al. 2017). Researchers proposed 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 (Wong et al. 2016; Pearson et al. 2017). (Xie et al. 2013) examined the equivalence and hierarchy between a number of formulae, while (Yoo et al. 2017) showed that there does not exist a perfect scoring formula which outperforms known techniques found by humans or even by automatic search-based methods. One additional reason an SBFL based method may fail is that these approaches provide only the ranked list of code elements, however this gives little or no information about the context of bugs which makes their comprehension a cumbersome task for developers.

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 our and other researchers' observations, intuitions and experiences presented in details in Section 2.2, and we hypothesize 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. With this intuition, developers can also make a decision ("judge") about the code snippets associated with the item they are currently examining. This allows them to narrow down the search space (i. e., set of the suspicious code elements) more efficiently, which could speed up finding the bug. For example, when users go through the ranked list of suspicious methods, in addition to the examined code element, they could have knowledge about its class, which information can be "fed back" into *iFL* to modify the suspiciousness value of other methods in that class or even exclude items to be examined. 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 (Gong et al. 2012; Hao et al. 2009; Bandyopadhyay and Ghosh 2012; Lei et al. 2012; Li et al. 2016; Li et al. 2018; Lin et al. 2017; Lehmann and Pradel 2018), but to our knowledge, similar contextual information about higher level entities has not yet been leveraged.

We evaluated the approach in three experiments. First, we used *simulation* to predict the effect of interactivity. We simulated user actions during hypothetical fault finding in well-known bug benchmarks, and measured the Expense metric improvements with respect to the following traditional SBFL formulae: Tarantula (Jones and Harrold 2005), Ochiai (Abreu et al. 2009), and DStar (Wong et al. 2014). We relied on two benchmarks: artificial defects from the SIR repository (Do et al. 2005) and real defects from Defects4J (Just et al. 2014). Results show that the method can significantly improve the fault localization efficiency: in both benchmarks, for 32-57% of the faults their ranking position is reduced from beyond the 10th position to between the 1-10th position. Taking into account all the defects, the localization efficiency in terms of Expense improved on average by 71-79%. For reference, we implemented a closely related interactive FL algorithm proposed by (Gong et al. 2012), called TALK, in our simulation framework. We compared the performance of *iFL* to TALK on the real faults from Defects4J, and found that *iFL* has a significant advantage over TALK. We also modelled 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.

In the second stage, we performed a quantitative evaluation of the successfulness of *iFL* usage by *real users*. 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 the third set of experiments, we relied on *think-aloud sessions* to better understand the possible use cases of the approach including benefits and risks. We relied on a group of professional programmers who were experienced in program debugging, and during the sessions we collected all the details about their debugging activities while trying to find bugs in our benchmark programs. Findings show that such an automated approach for fault localization might not be suitable for all programmers but it could be beneficial when the details of the approach and the benefits are carefully introduced. Note that, due the low number of

participants in this set of experiments we cannot support our findings with appropriate statistical data. However, this experiment helped us better understand the developers' thought processes and the weaknesses of the approach, and gave us possible directions for future enhancements.

In summary, our contributions are the following (a preliminary version of this paper was presented in (Horváth et al. 2020a)).

### Extensions for (Horváth et al. 2020a)'s Contributions

1. We expanded the set of evaluated SBFL formulae with Ochiai and DStar. These two additional formulae were utilized during the experiments with simulated users.
2. We expanded the set of evaluated bugs and programs by upgrading our measurement framework to the latest version (v2.0.0) of Defects4J.
3. We compared the results of our *iFL* experiments to a previously defined algorithm specified in (Gong et al. 2012).
4. The user experience and the speed of the Eclipse plug-in were enhanced. For example, we introduced a customizable filtering and sorting feature, which helps the developers to focus their attention on code elements with specific properties.
5. We investigated the usage scenarios of the approach by experienced programmers in great depth using think-aloud sessions, which is a novelty in this field.

### Contributions Presented in (Horváth et al. 2020a)

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.

The paper is organized as follows. Section 2 contains an example to illustrate our approach, which is then discussed in detail in Section 3. In Section 4, we summarize the goals of our empirical assessment, while Sections 5, 6 and 7 present the experimental results for the simulated and the real users, and the think-aloud sessions, respectively. A discussion of our findings is presented in Section 8 with possible threats to validity in Section 9. An overview of related work in Section 10, before concluding with Section 11.

## 2 Motivation and Background

### 2.1 Motivating Example

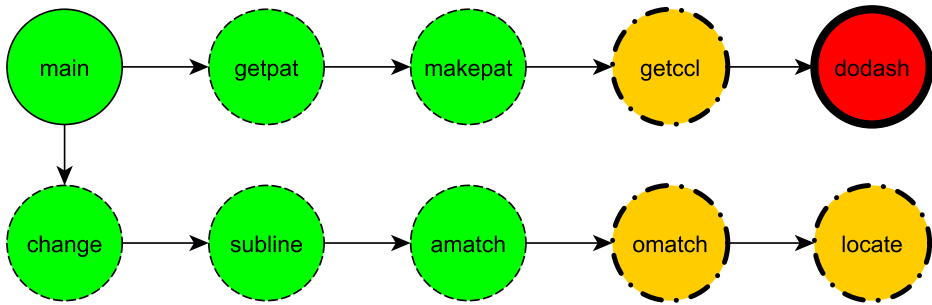
For illustration, consider the example in Table 1. This is part of program `replace` from the SIR benchmark repository, which includes manually seeded faults (this benchmark is often used in SBFL research, although being somewhat outdated). Line 116 is a predicate inside function `dodash`, where an artificial fault is seeded: the relation is changed and the +1 part is

Table 1 Example Code and Fault Localization Process with Seeded Fault

| Source code      |   | Test cases |     |     |     |     | Scores and ranks |              |              |              |  |
|------------------|---|------------|-----|-----|-----|-----|------------------|--------------|--------------|--------------|--|
| Line             | Code  | 557        | 560 | 855 | 857 | 864 | 0. iteration     | 1. iteration | 2. iteration | 3. iteration |  |
| 93               | void dodash(delim, src, i, dest, j, maxset)                   |            |     |     |     |     |                  |              |              |              |  |
| 115              | else if ((isalnum(src[*i - 1])) &&<br>(isalnum(src[*i + 1]))) | •          | •   | •   | •   | •   | 0.658 (23.)      | 0.658 (20.)  | 0.658 (7.)   | 0.658 (5.)   |  |
| 116              | // faulty version &&(src[*i - 1] > src[*i])) {                |            |     |     |     |     |                  |              |              |              |  |
| 116              | // original version // &&src[*i - 1] <= src[*i + 1])) {       | •          | •   | •   | •   | •   | 0.707 (11.)      | 0.707 (9.)   | 0.707 (2.)   | 0.707 (1.)   |  |
| 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            |  |
| Pass/Fail Status |   | P          | F   | F   | F   | P   |                  |              |              |              |  |

Table 1 (continued)

| Source code      |                                  | Test cases |     |     |     | Scores and ranks |              |              |              |              |
|------------------|----------------------------------|------------|-----|-----|-----|------------------|--------------|--------------|--------------|--------------|
| Line             | Code                             | 557        | 560 | 855 | 857 | 864              | 0. iteration | 1. iteration | 2. iteration | 3. iteration |
| 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                |              |              |              |              |



**Fig. 1** Call-graph part for the example program

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 1, while the call-graph in Fig. 1 shows the high level relationship of the four functions. 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 1 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 1 corresponds to the suspiciousness scores computed by the Tarantula method<sup>1</sup> 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 (Jones and Harrold 2005) for a code element  $s$ :

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

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 2 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.

<sup>1</sup>We used Tarantula in this example because it is easy to use in the explanation, but other SBFL techniques provide similar relative ranks.

**Table 2** Basic SBFL Statistics for the Example Program

| Line | <i>ef</i> | <i>ep</i> | <i>nf</i> | <i>np</i> | 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 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 (Masri and Assi 2014) 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.

## 2.2 Developers' Knowledge

In this research, we rely on the knowledge of developers and its application in debugging and fault localization processes. Our intuitions and experiences as professional software developers are the basic building blocks of the idea behind our *iFL* approach. In the following sections, we discuss the foundations of our intuitions and ideas based on examples and related works.

### 2.2.1 The Required Amount of Knowledge

One of these insights about developers' knowledge is that they tend to write whole bodies of source code which encapsulate certain functions under development. (Fritz et al. 2014) found that developers' knowledge models are consistently small:  $4.6 \pm 0.8$  classes. (Corritore and Wiedenbeck 1999) observed that the level of comprehension (knowledge) increased during code inspection tasks. This means that the developers will understand an increasing amount of the code base, hence they presumably will be able to give better feedback. (Fritz et al. 2014) found that developers frequently revisit parts of the source code and take less time if their navigation is more structured. They also show that six of the ten code navigation models centered around one large connected group of classes. In some cases, the models also contained a few additional items with no connections. (Sillito et al. 2006; Sillito et al. 2005) observed that developers first search for an initial focus point and then explore relationships from these points, also revisiting elements.

These findings enforce our intuition that the users of *iFL* could access the required amount of information to provide feedbacks with sufficiently good quality. In other words, developers usually have solid knowledge about a group of classes, but this model constantly changes and it is extended with new information as they investigate more code. We hypothesize that if a context (a collection of classes and functions) is not too large, then it is easily comprehensible and developers could give an opinion about such context. For example, the version of `replace` presented in Table 1 is only about 560 lines long consisting of 21 functions. Among these functions, there are a lot of short ones, like `locate`, and some even simpler utility functions containing only a few statements. These contexts are easy to understand, and developers can judge the suspiciousness with little effort, as in the 1st iteration of the example presented in Section 3. A feedback-based tool could help developers diversify their knowledge of a context by suggesting new unfamiliar code elements and hence modifying their usual search space.



## 2.2.2 Knowledge Acquiring Strategies

Besides the amount of information needed, it is important to inspect the strategies used by developers to acquire the necessary knowledge. Programmers use various multi-level approaches to understand software systems. They switch between program, situation, and domain (top-down) models (von Mayrhauser et al. 1996) frequently. Code navigation models can vary substantially across developers, even for tasks that require only small changes, furthermore, models can differ substantially on class and method level as well (Fritz et al. 2014).

The timing of these information gathering actions could be analyzed by inspecting the order in which the developers enumerate the code elements during a debugging task. (Robillard et al. 2004) found that successful developers re-investigate methods less frequently and most of the time they perform structurally guided searches. In addition, experienced developers visit less methods, thus wasting less time on understanding irrelevant methods (Latoza et al. 2007) e. g. the simpler functions (i. e. `getpat`, `change` and `locate`) from the `replace` program in the example in Section 2.

The wide range of knowledge acquiring strategies may imply that no tool could aid in all of the above cases. However, the presence of structure-based and class-level investigation strategies shows that there are at least some segments of the debugging process where *iFL* could be utilized.

## 2.2.3 The Nature of Knowledge Context

Other than being layered, knowledge models have various types of contexts which differ mostly in the source and the aspects of organizing the information. (Fritz et al. 2014) suggest that by learning from a developer's past a tool might be able to provide better recommendations and also a more accurate representation of a developer's code context model of the past. They also show that developers often, but in particular at the beginning of the investigation, visited those code elements subsequently that share terms within their identifiers. Likewise, it is obvious to organize code chunks according to their functions (Davies et al. 1995). For example, functions in Fig. 1 could be organized into such functionally similar groups (e. g., `getpat`, `makepat` and `amatch`, `omatch`) to help developers understand these parts of the `replace` program more quickly. Our *iFL* methodology does not utilize these aspects of the context yet. We are currently seeking ways to extend our algorithms and their implementations to incorporate these types of additional information.

## 2.2.4 Developers Are Not Omniscient, but They Are Experts

*iFL* requires additional knowledge from the developers about the system. There are two major ways of how this information could be present during the debugging sessions. It is either a priori or posterior, i. e., it is already part of the users' experience or can be synthesized on-demand when the developer inspects the relevant code. It is important to note that based on (Fritz et al. 2010; Fritz et al. 2007) findings we could argue that if developers have a priori knowledge about a class, they have a higher probability to give more accurate feedback for the given context; if they know a method, they have a higher probability to give more accurate feedback for the actual code element. We shall not disregard the fact that developers are not assigned to certain bugs at random. Most of the time, the task of fixing a particular bug is assigned to a developer who is familiar with the context of the specific issue. These two insights are key concepts behind the usability of the *iFL* methodology.

In conclusion, we can see that there is a certain level of experience that is required to decide when and how to apply a tool to increase the effectiveness and efficiency of debugging. The works presented above show that one could acquire the required amount of experience, but not in all cases though. However, in those cases where sufficient knowledge is available *iFL* could help improve the user's process.

### 3 Interactive Fault Localization Utilizing Contextual Knowledge

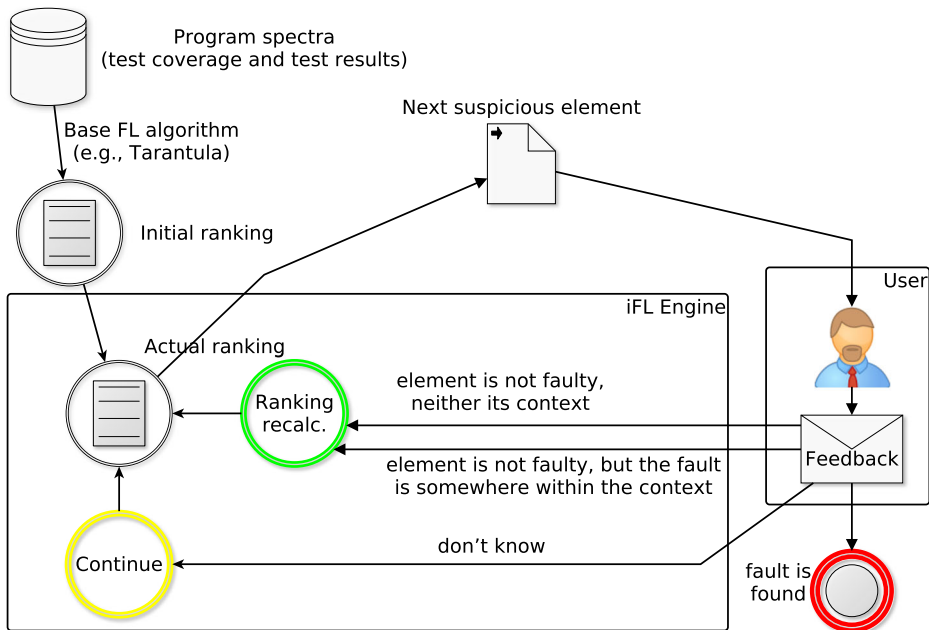
Our approach to improve SBFL is to leverage the background and acquired knowledge of the developer about the system being debugged outside her current focus – the currently investigated code element. We build on our and other researchers' observations, intuitions and experiences presented in Section 2.2, and we hypothesize that a programmer, when presented with a statement from a particular function, in general has an 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.

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 1). She 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 she examined it in a previous debugging session, etc.). 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, statement 116, 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 2 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.

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,



**Fig. 2** Basic process of Interactive Fault Localization

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. Note that answer (4) and its corresponding action could mean different things in different settings, and they could be implemented in various ways depending on the actual setting. For example, in an IDE where the suspicious elements are displayed in a list or a table (4) symbolizes that the developer looks at an element and chooses not to interact with it.

There may be different strategies to make the mentioned 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.

Since there are no increases in the suspiciousness scores, a mistake in the answer made by the developer can move the faulty element towards the end of the ranking which could make the fault finding efficiency even worse than not using the approach. It is safe to assume that developers will not be free of mistakes and completely knowledgeable about the system; in some cases, they may not be able to provide additional information on the context of a code element, or they can make wrong decisions. Therefore, part of our research goals in this paper is to verify what is the performance of the method when the user does exhibit some imperfection properties (this is our **RQ1.4** discussed below).

## 4 Evaluation Goals

We verified the effectiveness of the Interactive Fault Localization approach in four stages. First, we performed an empirical study using *simulated* users (Section 5). Next, we compared *iFL* to a closely related interactive approach, called TALK, proposed by (Gong et al.

2012). For this, we re-implemented the TALK algorithm in our framework, and we evaluated its performance on real faults with simulated users (Section 5.5). These studies were followed by another empirical study involving *real* users (Section 6). Then, we applied a think-aloud method with the help of senior programmers to find out the possible *use cases* of the approach in actual debugging scenarios (Section 7).

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., (Gong et al. 2012; Hao et al. 2009), 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.

The evaluations up to this point provide quantitative data about the usefulness of the approach, but we also wanted to qualitatively understand the possible use cases of the method in actual debugging scenarios, its limitations and possible future enhancements. For this purpose, a field experiment would have been an alternative, but at this stage the *iFL* method and tool are not ready to be deployed in real projects. Instead, in the third stage we performed a very thorough observational study of senior programmers solving a debugging task when the *iFL* was available to them. We decided to use the *think-aloud method*, which is a well-known approach in cognitive psychology (Ericsson and Simon 1980; van Someren et al. 1994), but is relatively rarely used in software engineering research. During the think-aloud sessions, the participants have been asked to orally explain all the details about the current activities, which was complemented by recording the developer's interactions with the development environment on the screen.

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 and an interactive approach?* We have the following Research Questions for this part of the evaluation:

- RQ1.1** What improvement can we observe with *iFL* on artificial faults from the SIR repository?
- RQ1.2** What improvement can we observe with *iFL* on real faults from the Defects4J repository?
- RQ1.3** How *iFL* compares to another interactive approaches on real faults from the Defects4J repository?
- RQ1.4** 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:

- RQ2.1** Is it true that users could find more bugs with *iFL* than users without access to the method?
- RQ2.2** Is it true that users could find bugs more quickly with *iFL* than users without access to the method?
- RQ2.3** How do real users subjectively evaluate the *iFL* method and its implementation in the development environment?

In the third stage, our goal was the following. *How could the iFL approach be made part of a typical debugging routine of experienced programmers? In other words, what could be the most useful use cases of the (present and future state of the) method?* Our Research Questions for this part are presented below. The purpose of the first two was to better understand the actual ways of thinking of the developers for problem solving regardless of the iFL approach. This way, we could better answer the actual question dealing with iFL.

- RQ3.1** What are the typical debugging workflows and associated problem solving strategies of experienced developers when no additional help is available beyond the usual debugging features in their IDEs?
- RQ3.2** What kind of additional information (inside the IDEs or external) experienced developers use during the debugging sessions that help them locate the bugs more quickly, and how do they integrate this information into their debugging workflow?
- RQ3.3** How did the same developers integrate the proposed iFL implementation into their debugging workflow and, apart from these use cases, what could be other potential ones provided iFL is enhanced accordingly?

The data from the think-aloud sessions can also help us verify the initial assumptions about the programmer's ability to decide on larger code entities (required for the context-based feedback) and to determine the actual levels of user imperfections (which is simulated in the first stage).

Eventually, the answers to the questions above could help us design new elements into the iFL approach and new features for the tool implementation.

## 5 Results with Simulated Users

### 5.1 Experiment Setup

To answer research questions **RQ1.1–RQ1.4**, 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 three algorithms: Tarantula (Jones and Harrold 2005), Ochiai (Abreu et al. 2009), and DStar (Wong et al. 2014), which have been reported to be the most successful in different settings (Pearson et al. 2017), and are often referred in literature. With this choice we wanted to verify if the actual algorithm has any impact on the effectiveness of the approach.

Regarding the user responses and iFL engine actions, for **RQ1.1** and **RQ1.2** we follow a relatively simple but strict approach (there are no intermediate, partial or uncertain responses and actions, in other words, we simulate a hypothetical *perfect* user). The perfect user is perfectly sure about her decisions, which means out of the four possible responses explained in Section 3, we will not use the fourth one, “don’t know”. The perfect user always recognizes the faulty method. She also has a perfect knowledge about whether the fault is in the context of the inspected element.<sup>2</sup> Furthermore, the mentioned strategy for the actions will be employed, that is, reducing either the whole context or everything but the context

<sup>2</sup>The exact implementation can be found at [https://github.com/sed-szeged/ifl/blob/09e1d2a0fc32089218b1dbbb9508bc2991e673dd/framework/experiment/experiment\\_one.py#L42](https://github.com/sed-szeged/ifl/blob/09e1d2a0fc32089218b1dbbb9508bc2991e673dd/framework/experiment/experiment_one.py#L42)

to 0. We could reduce the perfection of the simulated user in order to simulate imperfect knowledge about the system under testing. Experiments of user imperfection that answer **RQ1.4**, including the “don’t know” answer, are presented separately in Section 5.6.

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.

### 5.1.1 Implementation Details of TALK

To answer **RQ1.3** We have implemented the TALK algorithm in our simulation framework based on the pseudo codes and descriptions available in (Gong et al. 2012).<sup>3</sup> The core of this algorithm is a loop which waits for user feedback. In the loop two rules are utilized to apply adjustments to the scores of code elements. The first rule is triggered when a code element is labelled as clean by the user (the authors call these elements symptoms). This rule aims to find the root cause of such symptoms. When a root cause is found, a portion of the symptom’s original score is transferred to the root cause. The second rule is applied when a code element is labelled as faulty. In these cases, the score of the code elements which are covered by the smallest (in terms of the number of covered code elements) test case  $t_{\min}$  is increased by a factor  $\mathcal{K}_s$ .  $\mathcal{K}_s$  is set to a value which ensures that in the next few iterations of the algorithm the code elements covered by  $t_{\min}$  are displayed at the beginning of the list that is shown to the user.

Note that our evaluation considers only the first faulty code element that is found. Consequently, our simulation stops when the first faulty code element is found, therefore the second rule is never applied in our experiments.

## 5.2 Evaluation Method

Several strategies have been proposed in literature for measuring the effectiveness of SBFL methods, but they are practically all based on looking at the rank position of the actual faulty element within the list of all possible program elements. One strategy is to express this as the number of elements that need to be investigated by the programmer before finding the fault (Wong et al. 2016), and another is the opposite: elements that need not to be investigated (Renieris and Reiss 2003). This is usually expressed in relative terms compared to the length of the rank list (program size). However, (Parnin and Orso 2011) argued that absolute rankings are more helpful in practical situations.

Another issue with these mechanisms is the handling of ties (Xu et al. 2011), because in many cases it can happen that different program elements get assigned the same suspiciousness scores (as is the case with our example from Table 1 as well). Some approaches select the first (best case), last (worst case) or middle (expected case) element for expressing this value, while others simply treat the elements with the same values as all belonging to one position.

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. We express this in a set of measures called *Expense*, with two variants: an absolute one

<sup>3</sup>The source code of our implementation is available at: <https://github.com/sed-szeged/ifl/tree/feature/gerxyz/gong-experiment>

expressed in the number of code elements ( $E$ ) and a relative version compared to the length of the rank list ( $E'$ ). The following formulae express precisely how to calculate this value (following (Abreu et al. 2007)):

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

where  $N$  is the number of code elements, for  $i \in \{1, \dots, N\}$   $s_i$  is the suspiciousness score of the  $i$ th 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 (Xia et al. 2016; Kochhar et al. 2016). 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 2017) 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.

### 5.3 Results for Seeded Faults

To answer **RQ1.1**, seven small C/C++ programs from the Software-artifact Infrastructure Repository (SIR) (Do et al. 2005) were included in the experiments, which are the so-called

**Table 3** Details of Subject Programs from SIR

| Program      | LOC (CE)      | Functions | Tests  | Faults |
|--------------|---------------|-----------|--------|--------|
| printtokens  | 726 (277)     | 18        | 4 130  | 7      |
| printtokens2 | 570 (262)     | 19        | 4 115  | 10     |
| replace      | 564 (400)     | 21        | 5 542  | 32     |
| schedule     | 412 (225)     | 18        | 2 650  | 9      |
| schedule2    | 374 (198)     | 16        | 2 710  | 10     |
| tcas         | 173 (95)      | 9         | 1 608  | 41     |
| totinfo      | 565 (187)     | 7         | 1 052  | 23     |
| Total        | 3 384 (1 644) | 108       | 21 807 | 132    |



“Siemens” suite. This benchmark contains seeded faults, and both the original and faulty versions are available. The subject programs are listed in Table 3. 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).

Note that, the third column of Table 4 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. The latter issue was present in only a few cases and slightly differently in the three SBFL methods we investigated, which resulted in different number of cases we used for subject `printtokens2`. However, since there were no ways to improve fault localization efficiency in these cases, this did not impact the measurement results. For preparing the raw data for the *iFL* experiments including the code coverage information and test case results, the tools (GCOV 2021) and (SoDA 2021) were used.

Table 4 shows the improvements *iFL* was able to achieve on SIR. The performance of the original SBFL algorithms can be seen in column 4, which we used as the reference to evaluate *iFL*. Both absolute and relative versions of the Expense measure are provided. For each of the SBFL techniques, a summarization line is provided with the corresponding average values. The three techniques achieved similar results, Ochiai being slightly better than the other two. On average, it prioritized the faulty code elements roughly to the 20th place, which means that on average 13% of the executable code elements must be examined to find the faulty one. Ochiai was followed by DStar (19.90 – 13.24%) and then by Tarantula (24.85 – 15.43%).

Column 5 contains the same data for *iFL*. The average Expense measures are notably better than for the original algorithm. Ochiai performs slightly better in this case as well with an Expense of 5.78 (3.75%) on average. This means, that in this case a programmer would need only about six steps (5.78) to find the fault on average. DStar was the second best in this comparison with 5.83 (3.80%), but Tarantula was similar as well 6.86 (4.25%). In terms of relative improvement (Column 6), Tarantula achieved the best results: 17.99 steps (11.19%). It was followed by DStar with 14.11 (9.47%) and Ochiai was last with 14.01 (9.34%). Column 7 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). For all techniques, the improvement is notable, **71-72%**.

The last four columns of Table 4 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, around **49-57%**. More importantly, most of these improvements are those that bring the faulty code element from outside of top 10 into top 10 (Column 8) or, even better, into top 5 (Column 9). 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.



Table 4 *iFL* improvements on SIR

| Alg.      | Program      | Faults | Score          | $E(E')$        |                  | Diff.  | Impr.       | Enabling improvements         |                              |                     | Total       |
|-----------|--------------|--------|----------------|----------------|------------------|--------|-------------|-------------------------------|------------------------------|---------------------|-------------|
|           |              |        |                | <i>iFL</i>     |                  |        |             | [ $\infty$ , 10]<br>→ [10, 5] | [ $\infty$ , 10]<br>→ [5, 1] | [10, 5]<br>→ [5, 1] |             |
| DStar     | 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%)          | 0 ( 0.00%)  |
|           | printtokens2 | 5      | 23.00 ( 8.78%) | 6.30 ( 2.40%)  | -16.70 ( -6.37%) | 72.61% | 1 (20.00%)  | 1 (20.00%)                    | 0 ( 0.00%)                   | 0 ( 0.00%)          | 2 (40.00%)  |
|           | replace      | 22     | 13.05 ( 3.26%) | 3.66 ( 0.91%)  | -9.39 ( -2.35%)  | 71.95% | 1 ( 4.55%)  | 6 (27.27%)                    | 5 (22.73%)                   | 5 (22.73%)          | 12 (54.55%) |
|           | schedule     | 2      | 6.25 ( 2.78%)  | 2.50 ( 1.11%)  | -3.75 ( -1.67%)  | 60.00% | 0 ( 0.00%)  | 0 ( 0.00%)                    | 0 ( 0.00%)                   | 1 (50.00%)          | 1 (50.00%)  |
|           | schedule2    | 4      | 66.62 (33.59%) | 13.25 ( 6.68%) | -53.38 (-26.92%) | 80.11% | 0 ( 0.00%)  | 1 (25.00%)                    | 0 ( 0.00%)                   | 0 ( 0.00%)          | 1 (25.00%)  |
|           | tcas         | 31     | 20.26 (21.32%) | 5.39 ( 5.67%)  | -14.87 (-15.65%) | 73.41% | 11 (35.48%) | 11 (35.48%)                   | 1 ( 3.23%)                   | 1 ( 3.23%)          | 23 (74.19%) |
|           | totinfo      | 18     | 18.78 (10.04%) | 8.06 ( 4.31%)  | -10.92 (-5.84%)  | 58.14% | 7 (38.89%)  | 1 ( 5.56%)                    | 0 ( 0.00%)                   | 0 ( 0.00%)          | 8 (44.44%)  |
|           |              | 83     | 19.90 (13.24%) | 5.83 ( 3.80%)  | -14.11 (-9.47%)  | 70.91% | 20 (24.10%) | 20 (24.10%)                   | 7 (8.43%)                    | 47 (56.63%)         | 47 (56.63%) |
| Ochiai    | 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%)          | 0 ( 0.00%)  |
|           | printtokens2 | 7      | 19.14 ( 7.31%) | 4.79 ( 1.83%)  | -14.36 (-5.48%)  | 75.00% | 1 (14.29%)  | 1 (14.29%)                    | 0 ( 0.00%)                   | 0 ( 0.00%)          | 2 (28.57%)  |
|           | replace      | 22     | 13.09 ( 3.27%) | 3.66 ( 0.91%)  | -9.43 ( -2.36%)  | 72.05% | 1 ( 4.55%)  | 6 (27.27%)                    | 5 (22.73%)                   | 5 (22.73%)          | 12 (54.55%) |
|           | schedule     | 2      | 6.25 ( 2.78%)  | 2.50 ( 1.11%)  | -3.75 ( -1.67%)  | 60.00% | 0 ( 0.00%)  | 0 ( 0.00%)                    | 0 ( 0.00%)                   | 1 (50.00%)          | 1 (50.00%)  |
|           | schedule2    | 4      | 66.62 (33.59%) | 13.25 ( 6.68%) | -53.38 (-26.92%) | 80.11% | 0 ( 0.00%)  | 1 (25.00%)                    | 0 ( 0.00%)                   | 0 ( 0.00%)          | 1 (25.00%)  |
|           | tcas         | 31     | 20.32 (21.39%) | 5.42 ( 5.70%)  | -14.90 (-15.69%) | 73.33% | 11 (35.48%) | 11 (35.48%)                   | 1 ( 3.23%)                   | 1 ( 3.23%)          | 23 (74.19%) |
|           | totinfo      | 18     | 19.00 (10.16%) | 8.28 ( 4.43%)  | -10.92 (-5.84%)  | 57.46% | 8 (44.44%)  | 0 ( 0.00%)                    | 0 ( 0.00%)                   | 0 ( 0.00%)          | 8 (44.44%)  |
|           |              | 85     | 19.74 (13.07%) | 5.78 ( 3.75%)  | -14.01 (-9.34%)  | 70.95% | 21 (24.71%) | 19 (22.35%)                   | 7 (8.24%)                    | 47 (55.29%)         | 47 (55.29%) |
| Tarantula | 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%)          | 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%)                   | 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%)                   | 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%)                   | 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%)                   | 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%)                   | 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%)                    | 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%)         | 42 (49.41%) |

**Answer to RQ1.1:** In the case of SIR programs containing seeded faults, the Ochiai method produced the best results from SBFL techniques. Compared to it, *iFL* achieved **71% improvement** in Expense, and resulted in **47 enabling improvements**, which corresponds to **55% of the faults**.

## 5.4 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. A recent study by (Pearson et al. 2017) investigates existing techniques both on SIR and Defects4J, showing that the latter repository has quite different properties when it comes to the performance of various SBFL techniques. Here, we present our measurement results with *iFL* on defects from the Defect4J repository (Just et al. 2014), to answer **RQ1.2**.

Defects4J is a database and extensible framework which provides a high-level interface to real defects, and has been widely used in software testing research (B Le et al. 2016; Zhang et al. 2017; Just et al. 2014; Shamshiri et al. 2015). The version (v2.0.0) of the dataset that we used contains 17 open source Java programs and 835 bugs in total. For *iFL* experiments, we extended the Defects4J framework with our Java agent-based code-coverage measurement tool<sup>4</sup>. This tool attaches to the JVM and utilizes on-the-fly bytecode instrumentation to collect different levels of coverage information. Defects4J provides the fix for each bug as a patch set. We created a static analyzer tool which uses JavaParser (JavaParser 2021) to analyze the patch sets and provides information about the changed code elements. 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. There are some bugs in this benchmark where the change set becomes empty. The reason for this is that the patch set contains only additions of code elements i. e. the changed elements did not exist in the faulty version of the program. Obviously, non existent methods are not considered by any FL approach, therefore we excluded these cases from our experiments. In addition, there are some bugs where the suspiciousness score of the faulty code elements assigned by the actual SBFL technique was zero or undefined – we excluded these cases too. Note that, different algorithms could produce zero or undefined scores in different settings, therefore the number of excluded bugs could change from algorithm to algorithm. In our experiment we encountered 37 cases where only failing tests exercised the faulty code elements. In these cases, the denominator of the DStar formula ( $ep + (ef + nf) - ef$ ) evaluates to zero ( $0 + (n + 0) - n = 0$ ) and due to the division by zero the score is undefined. The main properties of programs from the Defects4J dataset can be seen in Table 5, and the number of bugs we could use in the experiments is shown in Table 6 (Faults column).

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 Fig. 2) including the responses

<sup>4</sup>The extended framework is available at: <https://github.com/Frenkymd/defects4j/tree/chain>

**Table 5** Main Properties of Programs Used from Defects4J

| Program         | Bugs | Size (LOC) | Tests  | Methods | Classes |
|-----------------|------|------------|--------|---------|---------|
| Chart           | 26   | 96 382     | 2 193  | 5 227   | 472     |
| Cli             | 39   | 1 936      | 94     | 149     | 19      |
| Closure         | 174  | 90 694     | 7 911  | 8 392   | 1151    |
| Codec           | 18   | 2 584      | 206    | 234     | 19      |
| Collections     | 4    | 26 409     | 15 393 | 3 532   | 422     |
| Compress        | 47   | 6 740      | 73     | 437     | 53      |
| Csv             | 16   | 0 806      | 54     | 83      | 11      |
| Gson            | 18   | 5 378      | 720    | 620     | 106     |
| JacksonCore     | 26   | 15 871     | 206    | 795     | 73      |
| JacksonDataBind | 112  | 42 964     | 1 098  | 3 546   | 447     |
| JacksonXml      | 6    | 4 679      | 138    | 293     | 35      |
| Jsoup           | 93   | 2 546      | 139    | 327     | 41      |
| JXPath          | 22   | 19 372     | 308    | 1 287   | 131     |
| Lang            | 64   | 21 778     | 2 291  | 2 353   | 158     |
| Math            | 106  | 84 317     | 4 378  | 6 350   | 818     |
| Mockito         | 38   | 10 517     | 1 378  | 1 454   | 291     |
| Time            | 26   | 27 795     | 4 041  | 3 612   | 204     |
| Total           | 835  | 460 768    | 40 621 | 38 691  | 4 451   |

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 5.3, 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.

With *iFL* we achieved high improvements compared to all the three fault localization metrics. The first part of Table 6 shows average ranking improvements (Diff. column) and its ratio compared to the number of all possible code elements (i. e. Java methods). The Expense measures for the original SBFL methods are quite different than for the SIR programs, they range from 1-98 steps, which is much more than the average on small programs, however, the relative measures are smaller, 1.68-1.74% on average. 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 5 and 6 of the table. The difference is between 30 and 32 positions on average, but given the large total number of elements, the change in percentages is modest. In this case, Ochiai produced the best initial ranking and the best final Expense measures with *iFL* as well. The relative improvement (column 7) is higher than for the SIR benchmark, it is about **79%** for each SBFL technique. Practically, this means that on average the *iFL* approach could potentially save 79% 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 6. Overall, *iFL* had **251-255 (32-33%)** 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

**Table 6** *iFL* improvements on Defects4J

| Alg.  | Program         | Faults | $E(E')$       | Impr.         |                       |        | Enabling improvements         |                              |                         | Total        |
|-------|-----------------|--------|---------------|---------------|-----------------------|--------|-------------------------------|------------------------------|-------------------------|--------------|
|       |                 |        |               | Avg rank      | Avg rank w <i>iFL</i> | Diff.  | $[\infty, 10]$<br>→ $[10, 5]$ | $[\infty, 10]$<br>→ $[5, 1]$ | $[10, 5]$<br>→ $[5, 1]$ |              |
| DStar | Chart           | 22     | 9.14 (0.20%)  | 2.91 (0.06%)  | -6.23 (-0.14%)        | 68.16% | 1 (4.55%)                     | 3 (13.64%)                   | 4 (18.18%)              | 8 (36.36%)   |
|       | Cli             | 37     | 14.45 (5.85%) | 4.62 (1.97%)  | -9.82 (-3.88%)        | 68.01% | 6 (16.22%)                    | 5 (13.51%)                   | 6 (16.22%)              | 17 (45.95%)  |
|       | Closure         | 173    | 89.45 (1.18%) | 15.89 (0.21%) | -73.64 (-0.96%)       | 82.32% | 34 (19.65%)                   | 17 (9.83%)                   | 14 (8.09%)              | 65 (37.57%)  |
|       | Codec           | 16     | 6.25 (1.61%)  | 3.25 (0.81%)  | -3.00 (-0.80%)        | 48.00% | 0 (0.00%)                     | 1 (6.25%)                    | 1 (6.25%)               | 2 (12.50%)   |
|       | Compress        | 46     | 15.73 (1.61%) | 4.71 (0.51%)  | -11.02 (-1.10%)       | 70.08% | 3 (6.52%)                     | 6 (13.04%)                   | 5 (10.87%)              | 14 (30.43%)  |
|       | Csv             | 16     | 5.97 (5.03%)  | 3.06 (2.26%)  | -3.25 (-2.99%)        | 54.45% | 0 (0.00%)                     | 1 (6.25%)                    | 5 (31.25%)              | 6 (37.50%)   |
|       | Gson            | 16     | 18.72 (2.53%) | 8.75 (1.19%)  | -10.09 (-1.35%)       | 53.92% | 3 (18.75%)                    | 0 (0.00%)                    | 0 (0.00%)               | 3 (18.75%)   |
|       | JacksonCore     | 22     | 9.11 (0.94%)  | 3.73 (0.36%)  | -5.39 (-0.58%)        | 59.10% | 2 (9.09%)                     | 2 (9.09%)                    | 4 (18.18%)              | 8 (36.36%)   |
|       | JacksonDatabind | 96     | 60.59 (1.37%) | 10.67 (0.24%) | -49.94 (-1.13%)       | 82.41% | 19 (19.79%)                   | 8 (8.33%)                    | 9 (9.38%)               | 36 (37.50%)  |
|       | JacksonXml      | 4      | 23.00 (7.78%) | 6.00 (2.03%)  | -17.00 (-5.75%)       | 73.91% | 1 (25.00%)                    | 0 (0.00%)                    | 1 (25.00%)              | 2 (50.00%)   |
|       | Jsoup           | 88     | 31.53 (3.25%) | 8.20 (0.81%)  | -23.36 (-2.44%)       | 74.07% | 12 (13.64%)                   | 10 (11.36%)                  | 8 (9.09%)               | 30 (34.09%)  |
|       | JxPath          | 21     | 53.48 (4.08%) | 12.71 (0.97%) | -40.76 (-3.11%)       | 76.22% | 3 (14.29%)                    | 2 (9.52%)                    | 2 (9.52%)               | 7 (33.33%)   |
|       | Lang            | 52     | 4.90 (0.24%)  | 2.40 (0.12%)  | -2.54 (-0.12%)        | 51.76% | 7 (13.46%)                    | 1 (1.92%)                    | 0 (0.00%)               | 8 (15.38%)   |
|       | Math            | 95     | 10.49 (0.27%) | 3.62 (0.10%)  | -6.88 (-0.17%)        | 65.58% | 12 (12.63%)                   | 11 (11.58%)                  | 9 (9.47%)               | 32 (33.68%)  |
|       | Mockito         | 35     | 24.24 (2.24%) | 6.84 (0.60%)  | -17.40 (-1.64%)       | 71.77% | 2 (5.71%)                     | 5 (14.29%)                   | 0 (0.00%)               | 7 (20.00%)   |
|       | Time            | 25     | 19.06 (0.55%) | 4.88 (0.14%)  | -14.46 (-0.42%)       | 75.87% | 2 (8.00%)                     | 1 (4.00%)                    | 3 (12.00%)              | 6 (24.00%)   |
|       |                 | 764    | 39.28 (1.74%) | 8.36 (0.49%)  | -30.96 (-1.26%)       | 78.82% | 107 (14.01%)                  | 73 (9.55%)                   | 71 (9.29%)              | 251 (32.85%) |

Table 6 (continued)

| Alg.   | Program         | Faults | $E(E')$        |                       | Diff.            | Impr.  | Enabling improvements         |                              |                     | Total        |
|--------|-----------------|--------|----------------|-----------------------|------------------|--------|-------------------------------|------------------------------|---------------------|--------------|
|        |                 |        | Avg rank       | Avg rank w <i>iFL</i> |                  |        | [ $\infty$ , 10)<br>→ [10, 5) | [ $\infty$ , 10)<br>→ [5, 1] | [10, 5)<br>→ [5, 1] |              |
| Ochiai | Chart           | 25     | 7.96 ( 0.18%)  | 2.64 ( 0.06%)         | -5.32 ( -0.12%)  | 66.83% | 1 ( 4.00%)                    | 3 (12.00%)                   | 4 (16.00%)          | 8 (32.00%)   |
|        | Cli             | 39     | 14.09 ( 5.74%) | 4.68 ( 2.01%)         | -9.41 ( -3.73%)  | 66.79% | 6 (15.38%)                    | 5 (12.82%)                   | 7 (17.95%)          | 18 (46.15%)  |
|        | Closure         | 173    | 90.25 ( 1.19%) | 15.92 ( 0.21%)        | -74.33 ( -0.97%) | 82.36% | 36 (20.81%)                   | 16 ( 9.25%)                  | 13 ( 7.51%)         | 65 (37.57%)  |
|        | Codec           | 16     | 6.34 ( 1.63%)  | 3.28 ( 0.82%)         | -3.06 ( -0.81%)  | 48.28% | 0 (0.00%)                     | 1 ( 6.25%)                   | 1 ( 6.25%)          | 2 (12.50%)   |
|        | Collections     | 1      | 1.00 ( 0.03%)  | 1.00 ( 0.03%)         | 0.00 ( 0.00%)    | -0.00% | 0 (0.00%)                     | 0 (0.00%)                    | 0 (0.00%)           | 0 (0.00%)    |
|        | Compress        | 47     | 15.72 ( 1.61%) | 4.72 ( 0.52%)         | -11.00 ( -1.09%) | 69.96% | 2 ( 4.26%)                    | 6 (12.77%)                   | 5 (10.64%)          | 13 (27.66%)  |
|        | Csv             | 16     | 5.97 ( 5.03%)  | 3.06 ( 2.26%)         | -3.25 ( -2.99%)  | 54.45% | 0 (0.00%)                     | 1 ( 6.25%)                   | 5 (31.25%)          | 6 (37.50%)   |
|        | Gson            | 16     | 18.91 ( 2.55%) | 8.94 ( 1.22%)         | -10.09 ( -1.35%) | 53.39% | 3 (18.75%)                    | 0 (0.00%)                    | 0 (0.00%)           | 3 (18.75%)   |
|        | JacksonCore     | 25     | 8.48 ( 0.87%)  | 3.80 ( 0.36%)         | -4.68 ( -0.51%)  | 55.19% | 2 ( 8.00%)                    | 1 ( 4.00%)                   | 5 (20.00%)          | 8 (32.00%)   |
|        | JacksonDatabind | 101    | 58.49 ( 1.32%) | 10.62 ( 0.24%)        | -47.89 ( -1.08%) | 81.88% | 18 (17.82%)                   | 8 ( 7.92%)                   | 9 ( 8.91%)          | 35 (34.65%)  |
|        | JacksonXml      | 5      | 18.60 ( 6.29%) | 5.00 ( 1.69%)         | -13.60 ( -4.60%) | 73.12% | 1 (20.00%)                    | 0 (0.00%)                    | 1 (20.00%)          | 2 (40.00%)   |
|        | Jsoup           | 89     | 31.26 ( 3.22%) | 8.03 ( 0.79%)         | -23.25 ( -2.43%) | 74.37% | 14 (15.73%)                   | 9 (10.11%)                   | 8 ( 8.99%)          | 31 (34.83%)  |
|        | JxPath          | 22     | 51.66 ( 3.94%) | 11.05 ( 0.84%)        | -40.61 ( -3.10%) | 78.62% | 4 (18.18%)                    | 3 (13.64%)                   | 2 ( 9.09%)          | 9 (40.91%)   |
|        | Lang            | 61     | 4.51 ( 0.22%)  | 2.23 ( 0.11%)         | -2.31 ( -0.11%)  | 51.27% | 6 ( 9.84%)                    | 1 ( 6.4%)                    | 1 ( 6.4%)           | 8 (13.11%)   |
|        | Math            | 104    | 10.00 ( 0.26%) | 3.51 ( 0.10%)         | -6.50 ( -0.16%)  | 65.02% | 13 (12.50%)                   | 11 (10.58%)                  | 10 ( 9.62%)         | 34 (32.69%)  |
|        | Mockito         | 35     | 24.47 ( 2.25%) | 6.96 ( 0.61%)         | -17.51 ( -1.64%) | 71.57% | 3 ( 8.57%)                    | 5 (14.29%)                   | 0 (0.00%)           | 8 (22.86%)   |
|        | Time            | 26     | 18.40 ( 0.53%) | 4.81 ( 0.14%)         | -13.87 ( -0.40%) | 75.34% | 2 ( 7.69%)                    | 0 (0.00%)                    | 3 (11.54%)          | 5 (19.23%)   |
|        |                 | 801    | 37.93 (1.68%)  | 8.10 (0.48%)          | -29.86 (-1.21%)  | 78.71% | 111 (13.86%)                  | 70 (8.74%)                   | 74 (9.24%)          | 255 (31.84%) |

**Table 6** (continued)

| Alg.      | Program         | Faults | $E(E')$        | Enabling improvements |                                    |                                   |              |             |             |              |
|-----------|-----------------|--------|----------------|-----------------------|------------------------------------|-----------------------------------|--------------|-------------|-------------|--------------|
|           |                 |        |                | Impr.                 | $[\infty, 10] \rightarrow [10, 5]$ | $[\infty, 10] \rightarrow [5, 1]$ | Total        |             |             |              |
|           |                 |        | Avg rank       | Avg rank w $iFL$      | Diff.                              |                                   |              |             |             |              |
| Tarantula | Chart           | 25     | 13.00 ( 0.30%) | 3.16 ( 0.07%)         | -9.84 ( -0.23%)                    | 75.69%                            | 0 ( 0.00%)   | 3 (12.00%)  | 3 (12.00%)  | 6 (24.00%)   |
|           | Cli             | 39     | 15.06 ( 6.09%) | 4.76 ( 2.05%)         | -10.31 ( -4.04%)                   | 68.43%                            | 5 (12.82%)   | 5 (12.82%)  | 7 (17.95%)  | 17 (43.59%)  |
|           | Closure         | 173    | 98.36 ( 1.30%) | 16.35 ( 0.22%)        | -82.01 ( -1.08%)                   | 83.38%                            | 32 (18.50%)  | 17 ( 9.83%) | 14 ( 8.09%) | 63 (36.42%)  |
|           | Codec           | 16     | 6.59 ( 1.68%)  | 3.44 ( 0.85%)         | -3.16 ( -0.83%)                    | 47.87%                            | 0 ( 0.00%)   | 1 ( 6.25%)  | 1 ( 6.25%)  | 2 (12.50%)   |
|           | Collections     | 1      | 1.00 ( 0.03%)  | 1.00 ( 0.03%)         | 0.00 ( 0.00%)                      | -0.00%                            | 0 ( 0.00%)   | 0 ( 0.00%)  | 0 ( 0.00%)  | 0 ( 0.00%)   |
|           | Compress        | 47     | 17.36 ( 1.81%) | 5.04 ( 0.55%)         | -12.32 ( -1.26%)                   | 70.96%                            | 2 ( 4.26%)   | 7 (14.89%)  | 5 (10.64%)  | 14 (29.79%)  |
|           | Csv             | 16     | 5.97 ( 5.03%)  | 3.06 ( 2.26%)         | -3.25 ( -2.99%)                    | 54.45%                            | 0 ( 0.00%)   | 1 ( 6.25%)  | 5 (31.25%)  | 6 (37.50%)   |
|           | Gson            | 16     | 18.88 ( 2.55%) | 9.06 ( 1.23%)         | -10.06 ( -1.35%)                   | 53.31%                            | 3 (18.75%)   | 0 ( 0.00%)  | 0 ( 0.00%)  | 3 (18.75%)   |
|           | JacksonCore     | 25     | 8.34 ( 0.82%)  | 3.66 ( 0.33%)         | -4.68 ( -0.50%)                    | 56.12%                            | 2 ( 8.00%)   | 2 ( 8.00%)  | 3 (12.00%)  | 7 (28.00%)   |
|           | JacksonDatabind | 101    | 58.97 ( 1.33%) | 10.63 ( 0.24%)        | -48.36 ( -1.09%)                   | 82.00%                            | 18 (17.82%)  | 10 ( 9.90%) | 8 ( 7.92%)  | 36 (35.64%)  |
|           | JacksonXml      | 5      | 18.60 ( 6.29%) | 5.00 ( 1.69%)         | -13.60 ( -4.60%)                   | 73.12%                            | 1 (20.00%)   | 0 ( 0.00%)  | 1 (20.00%)  | 2 (40.00%)   |
|           | Jsoup           | 89     | 31.98 ( 3.28%) | 8.22 ( 0.81%)         | -23.76 ( -2.47%)                   | 74.30%                            | 13 (14.61%)  | 10 (11.24%) | 8 ( 8.99%)  | 31 (34.83%)  |
|           | JxPath          | 22     | 42.27 ( 3.22%) | 10.59 ( 0.81%)        | -31.68 ( -2.41%)                   | 74.95%                            | 3 (13.64%)   | 3 (13.64%)  | 4 (18.18%)  | 10 (45.45%)  |
|           | Lang            | 61     | 5.20 ( 0.25%)  | 2.46 ( 0.12%)         | -2.77 ( -0.13%)                    | 53.31%                            | 7 (11.48%)   | 0 ( 0.00%)  | 0 ( 0.00%)  | 7 (11.48%)   |
|           | Math            | 104    | 9.89 ( 0.25%)  | 3.57 ( 0.10%)         | -6.33 ( -0.15%)                    | 64.03%                            | 14 (13.46%)  | 10 ( 9.62%) | 11 (10.58%) | 35 (33.65%)  |
|           | Mockito         | 35     | 27.40 ( 2.52%) | 8.07 ( 0.69%)         | -19.33 ( -1.83%)                   | 70.54%                            | 4 (11.43%)   | 5 (14.29%)  | 0 ( 0.00%)  | 9 (25.71%)   |
|           | Time            | 26     | 19.71 ( 0.57%) | 5.17 ( 0.15%)         | -14.73 ( -0.43%)                   | 74.73%                            | 1 ( 3.85%)   | 0 ( 0.00%)  | 4 (15.38%)  | 5 (19.23%)   |
|           |                 | 801    | 40.07 (1.74%)  | 8.33 (0.49%)          | -31.77 (-1.26%)                    | 79.27%                            | 105 (13.11%) | 74 (9.24%)  | 74 (9.24%)  | 253 (31.59%) |

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 RQ1.2:** In the case of Defects4J experiments with real faults, the Ochiai method produced the best results from traditional techniques. Relative to it, *iFL* achieved **79% improvement** in Expense, and produced **255 enabling improvements**, corresponding to **32% of the faults in this benchmark**.

## 5.5 Results for Real Faults with the TALK Algorithm

In this section, we present the results of the replication study in which we re-implemented the TALK algorithm proposed by (Gong et al. 2012).

Note that we could not replicate the original experiment to its full extent. We did our best to acquire the data and the most detailed description of the experiments from the original paper. However, we could only work with what was published. In order to accommodate the highest number of bugs we used the latest version of Defects4J. Since the approach for coverage data collection was not mentioned in the original paper, we could not reuse the exact same measurement methods. We also used method level granularity, which could have affected the results. In addition, we ran into some issues while executing TALK on the two largest subject programs from Defects4J. Unfortunately, because of the complexity of the algorithm we were not able to execute TALK on Closure and on some bugs of JacksonDataBind.

We followed the same approach to evaluate TALK that we presented at the beginning of Section 5, and we used the same benchmark on which we evaluated *iFL* in Section 5.4. However, as knowledge and confidence cannot be interpreted for the TALK algorithms, we did not check the effect of these factors in the replication experiments.

As published, the baseline performance of TALK in terms of improvement is between -14% and 60% (12.29% on average) with Ochiai, -25% and 46% (11.71%) with Jaccard and -35% and 72% (13.37%) with Tarantula. Table 7 shows the performance of the re-implemented TALK algorithm on the Defects4J benchmark in the same fashion as Table 6 in Section 5.4. As it can be seen, we obtained varying results. The best performance is achieved using DStar. In this setting, improvements range from about -6% up to 10% and 1.56% overall. In the case of Ochiai and Tarantula, TALK is behind its traditional SBFL counterparts by a slight margin, -0.58% and -7.20% respectively. The maximal improvement (16%) is achieved on Chart with Tarantula. However, in the case of some projects TALK performs significantly worse than Ochiai and Tarantula. Also, the number of enabling improvements is very limited compared to *iFL*.

**Answer to RQ1.3:** Regarding Expense, TALK achieves -0.29 (**1.56%**), 0.10 (**-0.58%**) and 1.31 (**-7.20%**) improvement on average compared to DStar, Ochiai and Tarantula respectively, which is worse than *iFL*'s **78.72%**, **78.71%** and **79.27%**. Considering enabling improvements, TALK produces **20-21** such improvements which correspond to the **3.50-3.73%** of the faults, which is significantly less than the 251-255 (**31.59-32.85%**) that *iFL* produces.

Table 7 Talk improvements on Defects4J

| Alg.  | Program         | Faults | $E(E')$       | Avg rank      |                | Diff.  | Impr.     | Enabling improvements              |                                   |                              | Total      |
|-------|-----------------|--------|---------------|---------------|----------------|--------|-----------|------------------------------------|-----------------------------------|------------------------------|------------|
|       |                 |        |               |               |                |        |           | $[\infty, 10] \rightarrow [10, 5]$ | $[\infty, 10] \rightarrow [5, 1]$ | $[10, 5] \rightarrow [5, 1]$ |            |
| DStar | Chart           | 22     | 9.14 (0.20%)  | 8.59 (0.19%)  | -0.55 (-0.02%) | 5.97%  | 0 (0.00%) | 0 (0.00%)                          | 1 (4.55%)                         | 1 (4.55%)                    | 1 (4.55%)  |
|       | Cli             | 37     | 14.45 (5.85%) | 13.97 (5.65%) | -0.47 (-0.20%) | 3.27%  | 1 (2.70%) | 1 (2.70%)                          | 1 (2.70%)                         | 1 (2.70%)                    | 3 (8.11%)  |
|       | Codec           | 16     | 6.25 (1.61%)  | 6.25 (1.60%)  | 0.00 (-0.00%)  | -0.00% | 0 (0.00%) | 0 (0.00%)                          | 0 (0.00%)                         | 0 (0.00%)                    | 0 (0.00%)  |
|       | Compress        | 46     | 15.73 (1.61%) | 16.05 (1.72%) | 0.33 (0.11%)   | -2.07% | 1 (2.17%) | 0 (0.00%)                          | 1 (2.17%)                         | 1 (2.17%)                    | 2 (4.35%)  |
|       | Csv             | 16     | 5.97 (5.03%)  | 5.75 (4.97%)  | -0.22 (-0.06%) | 3.66%  | 0 (0.00%) | 0 (0.00%)                          | 1 (6.25%)                         | 1 (6.25%)                    | 1 (6.25%)  |
|       | Gson            | 16     | 18.72 (2.53%) | 18.38 (2.48%) | -0.34 (-0.05%) | 1.84%  | 1 (6.25%) | 0 (0.00%)                          | 0 (0.00%)                         | 0 (0.00%)                    | 1 (6.25%)  |
|       | JacksonCore     | 22     | 9.11 (0.94%)  | 9.68 (0.98%)  | 0.57 (0.04%)   | -6.23% | 0 (0.00%) | 0 (0.00%)                          | 0 (0.00%)                         | 0 (0.00%)                    | 0 (0.00%)  |
|       | JacksonDatabind | 41     | 32.76 (0.80%) | 29.44 (0.72%) | -3.32 (-0.07%) | 10.13% | 0 (0.00%) | 1 (2.44%)                          | 1 (2.44%)                         | 1 (2.44%)                    | 2 (4.88%)  |
|       | JacksonXml      | 4      | 23.00 (7.78%) | 22.50 (7.61%) | -0.50 (-0.17%) | 2.17%  | 0 (0.00%) | 0 (0.00%)                          | 0 (0.00%)                         | 0 (0.00%)                    | 0 (0.00%)  |
|       | Jsoup           | 88     | 31.53 (3.25%) | 30.46 (3.09%) | -1.07 (-0.16%) | 3.41%  | 0 (0.00%) | 1 (1.14%)                          | 5 (5.68%)                         | 6 (6.82%)                    | 6 (6.82%)  |
|       | JxPath          | 21     | 53.48 (4.08%) | 56.29 (4.29%) | 2.81 (0.21%)   | -5.25% | 0 (0.00%) | 0 (0.00%)                          | 0 (0.00%)                         | 0 (0.00%)                    | 0 (0.00%)  |
|       | Lang            | 52     | 4.90 (0.24%)  | 5.02 (0.25%)  | 0.12 (0.01%)   | -2.35% | 0 (0.00%) | 0 (0.00%)                          | 0 (0.00%)                         | 0 (0.00%)                    | 0 (0.00%)  |
|       | Math            | 95     | 10.49 (0.27%) | 10.69 (0.27%) | 0.21 (0.00%)   | -1.96% | 2 (2.11%) | 0 (0.00%)                          | 1 (1.05%)                         | 3 (3.16%)                    | 3 (3.16%)  |
|       | Mockito         | 35     | 24.24 (2.24%) | 24.10 (2.22%) | -0.14 (-0.02%) | 0.59%  | 1 (2.86%) | 0 (0.00%)                          | 0 (0.00%)                         | 0 (0.00%)                    | 1 (2.86%)  |
|       | Time            | 25     | 19.06 (0.55%) | 19.32 (0.56%) | 0.26 (0.01%)   | -1.36% | 0 (0.00%) | 0 (0.00%)                          | 0 (0.00%)                         | 0 (0.00%)                    | 0 (0.00%)  |
|       |                 | 536    | 18.78 (1.92%) | 18.48 (1.89%) | -0.29 (-0.03%) | 1.56%  | 6 (1.12%) | 3 (0.56%)                          | 11 (2.05%)                        | 20 (3.73%)                   | 20 (3.73%) |



Table 7 (continued)

| Alg.   | Program         | Faults | $E(E')$       |                 | Diff.          | Impr.   | Enabling improvements         |                          |            |
|--------|-----------------|--------|---------------|-----------------|----------------|---------|-------------------------------|--------------------------|------------|
|        |                 |        | Avg rank      | Avg rank w TALK |                |         | $[\infty, 10]$<br>→ $[10, 5]$ | $[10, 10]$<br>→ $[5, 1]$ | Total      |
| Ochiai | Chart           | 25     | 7.96 (0.18%)  | 6.76 (0.15%)    | -1.20 (-0.03%) | 15.08%  | 0 (0.00%)                     | 0 (0.00%)                | 1 (4.00%)  |
|        | Cli             | 39     | 14.09 (5.74%) | 14.60 (6.10%)   | 0.51 (0.36%)   | -3.64%  | 0 (0.00%)                     | 2 (5.13%)                | 1 (2.56%)  |
|        | Codec           | 16     | 6.34 (1.63%)  | 6.22 (1.55%)    | -0.12 (-0.08%) | 1.97%   | 1 (6.25%)                     | 0 (0.00%)                | 0 (0.00%)  |
|        | Collections     | 1      | 1.00 (0.03%)  | 1.00 (0.03%)    | 0.00 (0.00%)   | -0.00%  | 0 (0.00%)                     | 0 (0.00%)                | 0 (0.00%)  |
|        | Compress        | 47     | 15.72 (1.61%) | 16.23 (1.74%)   | 0.51 (0.13%)   | -3.25%  | 0 (0.00%)                     | 1 (2.13%)                | 1 (2.13%)  |
|        | Csv             | 16     | 5.97 (5.03%)  | 6.44 (5.58%)    | 0.47 (0.54%)   | -7.85%  | 0 (0.00%)                     | 0 (0.00%)                | 1 (6.25%)  |
|        | Gson            | 16     | 18.91 (2.55%) | 19.00 (2.56%)   | 0.09 (0.01%)   | -0.50%  | 1 (6.25%)                     | 0 (0.00%)                | 0 (0.00%)  |
|        | JacksonCore     | 25     | 8.48 (0.87%)  | 9.56 (0.96%)    | 1.08 (0.09%)   | -12.74% | 0 (0.00%)                     | 0 (0.00%)                | 2 (8.00%)  |
|        | JacksonDatabind | 44     | 31.52 (0.77%) | 29.48 (0.72%)   | -2.05 (-0.04%) | 6.49%   | 0 (0.00%)                     | 1 (2.27%)                | 1 (2.27%)  |
|        | JacksonXml      | 5      | 18.60 (6.29%) | 18.20 (6.16%)   | -0.40 (-0.13%) | 2.15%   | 0 (0.00%)                     | 0 (0.00%)                | 0 (0.00%)  |
|        | Jsoup           | 89     | 31.26 (3.22%) | 30.11 (3.09%)   | -1.15 (-0.12%) | 3.67%   | 0 (0.00%)                     | 2 (2.25%)                | 5 (5.62%)  |
|        | JxPath          | 22     | 51.66 (3.94%) | 56.14 (4.28%)   | 4.48 (0.34%)   | -8.67%  | 0 (0.00%)                     | 0 (0.00%)                | 0 (0.00%)  |
|        | Lang            | 61     | 4.51 (0.22%)  | 4.79 (0.23%)    | 0.28 (0.02%)   | -6.18%  | 0 (0.00%)                     | 0 (0.00%)                | 0 (0.00%)  |
|        | Math            | 104    | 10.00 (0.26%) | 10.72 (0.28%)   | 0.72 (0.02%)   | -7.16%  | 0 (0.00%)                     | 0 (0.00%)                | 1 (0.96%)  |
|        | Mockito         | 35     | 24.47 (2.25%) | 24.61 (2.26%)   | 0.14 (0.01%)   | -0.58%  | 1 (2.86%)                     | 0 (0.00%)                | 1 (2.86%)  |
|        | Time            | 26     | 18.40 (0.53%) | 18.81 (0.54%)   | 0.40 (0.01%)   | -2.19%  | 0 (0.00%)                     | 0 (0.00%)                | 0 (0.00%)  |
|        |                 | 571    | 17.95 (1.83%) | 18.05 (1.88%)   | 0.10 (0.05%)   | -0.58%  | 3 (0.53%)                     | 6 (1.05%)                | 20 (3.50%) |

**Table 7** (continued)

| Alg.      | Program         | Faults | $E(E')$       | Avg rank w TAL K |                | Diff.   | Impr.     | Enabling improvements         |                              |            |
|-----------|-----------------|--------|---------------|------------------|----------------|---------|-----------|-------------------------------|------------------------------|------------|
|           |                 |        |               | Avg rank         |                |         |           | $[\infty, 10]$<br>→ $[10, 5]$ | $[\infty, 10]$<br>→ $[5, 1]$ | Total      |
| Tarantula | Chart           | 25     | 13.00 (0.30%) | 10.92 (0.24%)    | -2.08 (-0.06%) | 16.00%  | 0 (0.00%) | 0 (0.00%)                     | 1 (4.00%)                    | 1 (4.00%)  |
|           | Cli             | 39     | 15.06 (6.09%) | 15.96 (6.66%)    | 0.90 (0.58%)   | -5.96%  | 1 (2.56%) | 1 (2.56%)                     | 0 (0.00%)                    | 2 (5.13%)  |
|           | Codec           | 16     | 6.59 (1.68%)  | 6.38 (1.65%)     | -0.22 (-0.04%) | 3.32%   | 1 (6.25%) | 0 (0.00%)                     | 0 (0.00%)                    | 1 (6.25%)  |
|           | Collections     | 1      | 1.00 (0.03%)  | 1.00 (0.03%)     | 0.00 (0.00%)   | -0.00%  | 0 (0.00%) | 0 (0.00%)                     | 0 (0.00%)                    | 0 (0.00%)  |
|           | Compress        | 47     | 17.36 (1.81%) | 18.02 (1.94%)    | 0.66 (0.13%)   | -3.80%  | 0 (0.00%) | 1 (2.13%)                     | 1 (2.13%)                    | 2 (4.26%)  |
|           | Csv             | 16     | 5.97 (5.03%)  | 6.44 (5.43%)     | 0.47 (0.40%)   | -7.85%  | 0 (0.00%) | 0 (0.00%)                     | 1 (6.25%)                    | 1 (6.25%)  |
|           | Gson            | 16     | 18.88 (2.55%) | 20.56 (2.77%)    | 1.69 (0.22%)   | -8.94%  | 1 (6.25%) | 0 (0.00%)                     | 0 (0.00%)                    | 1 (6.25%)  |
|           | JacksonCore     | 25     | 8.34 (0.82%)  | 12.02 (1.15%)    | 3.68 (0.33%)   | -44.12% | 0 (0.00%) | 0 (0.00%)                     | 2 (8.00%)                    | 2 (8.00%)  |
|           | JacksonDatabind | 42     | 28.26 (0.70%) | 29.99 (0.74%)    | 1.73 (0.05%)   | -6.11%  | 0 (0.00%) | 1 (2.38%)                     | 1 (2.38%)                    | 2 (4.76%)  |
|           | JacksonXml      | 5      | 18.60 (6.29%) | 16.20 (5.48%)    | -2.40 (-0.82%) | 12.90%  | 0 (0.00%) | 0 (0.00%)                     | 0 (0.00%)                    | 0 (0.00%)  |
|           | Jsoup           | 89     | 31.98 (3.28%) | 31.85 (3.28%)    | -0.13 (-0.00%) | 0.40%   | 1 (1.12%) | 4 (4.49%)                     | 2 (2.25%)                    | 7 (7.87%)  |
|           | JxPath          | 22     | 42.27 (3.22%) | 50.41 (3.84%)    | 8.14 (0.62%)   | -19.25% | 0 (0.00%) | 0 (0.00%)                     | 0 (0.00%)                    | 0 (0.00%)  |
|           | Lang            | 61     | 5.20 (0.25%)  | 5.82 (0.29%)     | 0.62 (0.03%)   | -11.99% | 0 (0.00%) | 0 (0.00%)                     | 0 (0.00%)                    | 0 (0.00%)  |
|           | Math            | 104    | 9.89 (0.25%)  | 12.45 (0.32%)    | 2.56 (0.07%)   | -25.91% | 0 (0.00%) | 0 (0.00%)                     | 1 (0.96%)                    | 1 (0.96%)  |
|           | Mockito         | 35     | 27.40 (2.52%) | 28.59 (2.61%)    | 1.19 (0.09%)   | -4.33%  | 1 (2.86%) | 0 (0.00%)                     | 0 (0.00%)                    | 1 (2.86%)  |
|           | Time            | 26     | 19.71 (0.57%) | 20.94 (0.61%)    | 1.23 (0.04%)   | -6.24%  | 0 (0.00%) | 0 (0.00%)                     | 0 (0.00%)                    | 0 (0.00%)  |
|           |                 | 569    | 18.13 (1.88%) | 19.43 (2.00%)    | 1.31 (0.12%)   | -7.20%  | 5 (0.88%) | 7 (1.23%)                     | 9 (1.58%)                    | 21 (3.69%) |

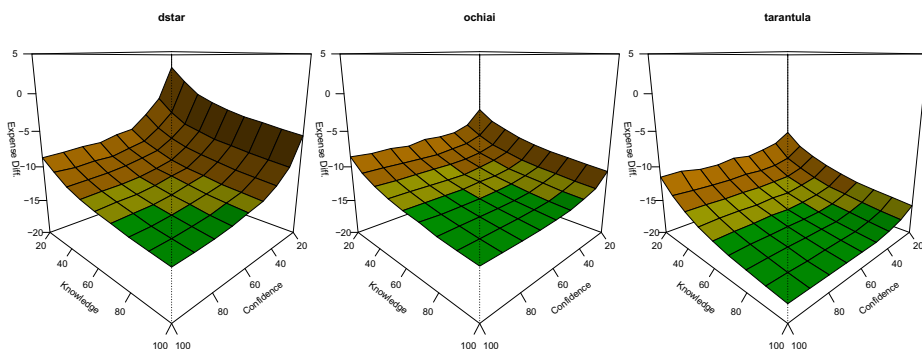
## 5.6 Effect of User Imperfections

To answer **RQ1.4**, 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. 2009) 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. 2016; Li et al. 2018) 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. Both studies consider a limited range (5–30%, 50%) of the various factors that impact the effectiveness, and they conclude that various factors modelling user imperfections indeed had impact on the effectiveness results, but not that significant which would invalidate their findings. We provide a similar, but more detailed analysis of user imperfection by experimenting with two factors that may influence the validity of simulated users:

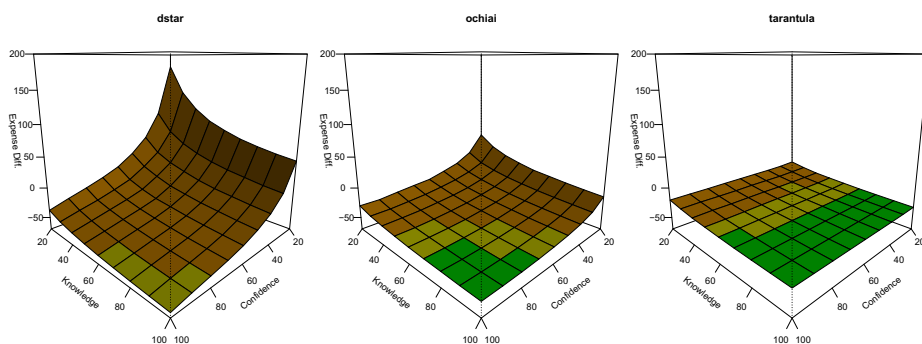
**Confidence Level** This factor indicates how much confidence we have in the user for providing reliable answers. We model confidence by applying a proportional decrease of the scores instead of nullation as with the base algorithm. The *iFL* engine was modified to scale down the suspiciousness of appropriate 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 about the context. Thus, knowledge is modelled 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.

These two factors were designed and implemented solely for the experiments for answering **RQ1.4**. If both factors are set to 1, we will obtain the base approach we used for research



**Fig. 3** *iFL* improvement with different knowledge and confidence levels on SIR



**Fig. 4** *iFL* improvement with different knowledge and confidence levels on Defects4J

questions **RQ1.1-RQ1.3**. Any combination of values 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 Figs. 3 and 4. 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. Results show that both knowledge and confidence affect the performance of *iFL*, though to various extent. The algorithm is sensitive to both factors when DStar is used as the base suspiciousness score, however it is relatively stable with Ochiai and Tarantula. We can observe that confidence and knowledge have different effects. While the algorithm scales almost linearly with knowledge, lack of confidence causes performance loss on a near exponential rate. 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.

Table 8 shows the approximate lowest knowledge and confidence levels allowable to limit the performance loss of *iFL* to at most 10 or 20% (reduction to 80 or 90% of the gain of the base *iFL* algorithm). These requirements were calculated by setting one of the factors

**Table 8** Requirements for keeping 80-90% of improvement

|     |           | 90%   |       | 80%   |       |
|-----|-----------|-------|-------|-------|-------|
|     |           | Know. | Conf. | Know. | Conf. |
| SIR | DStar     | 60%   | 60%   | 40%   | 40%   |
|     | Ochiai    | 60%   | 40%   | 40%   | 30%   |
|     | Tarantula | 60%   | 30%   | 40%   | 20%   |
| D4J | DStar     | 70%   | 90%   | 50%   | 80%   |
|     | Ochiai    | 60%   | 70%   | 50%   | 50%   |
|     | Tarantula | 60%   | 20%   | 40%   | 20%   |

to 100%, e. g., the last column in the last row means that confidence can be as low as 20% (while knowledge is set to 100%), but *iFL* still provides 80% of its benefits compared to the best case scenario on Defects4J using Tarantula (except DStar on Defects4J).

User imperfection is more realistic when both factors are changed at the same time. For the minimum gain levels 80–90%, consider the areas of the surfaces in Figs. 3 and 4 marked with different shades of yellow and green. Points satisfying the 80 or 90% gain level are positioned in the lower part of the surface. To keep the desired gain level, a tradeoff between confidence and knowledge may be made but when both aspects of user imperfection exceed the roughly 30–40% levels, the overall gain will be above the 80% of the gain of the perfect user.

**Answer to RQ1.4:** User imperfection, as modelled by our experiments, affects the results of the context aware *iFL* algorithm differently. *iFL* is sensitive to the knowledge and the confidence factors as well. Our experiments show that even very low confidence (20–30%) and knowledge levels (30–40%) suffice to keep 80% of the improvements.

## 6 Quantitative Results with Real Users

### 6.1 Experiment Setup

To answer research questions **RQ2.1–RQ2.3**, 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 2011) and (Le et al. 2019).

**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. 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. 23 participants had at least 1 year programming experience working in industry. 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 (Balogh et al. 2019; Balogh et al. 2019) 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 feedback, 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 project. One important consideration in the selection was to have bugs where the rank of the faulty code element is small (math-53) and slightly larger (ship-3), however, other aspects were also considered. Such characteristics were the following:

- (i) the number of faulty methods, i. e. there should be a bug where one method is faulty (math-5) and also one where there are multiple faulty methods (time-9)
- (ii) the complexity of the bugfix, i. e. the fix should not be too complicated/complex (since students participated in the experiment) and
- (iii) the similarity of ranks, i. e. the FL algorithms should produce the same result in terms of the ranks of the faulty methods.

These properties ensured that we control the most variables during the experiment. In 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 9 shows the task assignments (I: *iFL* used, N: *iFL* not used).

**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

**Table 9** 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 |

**Table 10** Number of All and Completed Tasks

| Bug         | Overall |           | with <i>iFL</i> |           | no <i>iFL</i> |           |
|-------------|---------|-----------|-----------------|-----------|---------------|-----------|
|             | 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%) |

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.

## 6.2 Results for Bug Finding Efficiency

Results regarding the number of completed tasks are presented in Table 10.<sup>5</sup> 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).<sup>6</sup> 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).

The overall success was 64% but it was highly varying 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).

<sup>5</sup>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.

<sup>6</sup>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 11** Task Completion Time in Hours, Minutes and Seconds

| Bug         | with <i>iFL</i> | no <i>iFL</i> | 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%) |

**Answer to RQ2.1:** *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.

Table 11 shows the results we collected about the time required to localize the fault, which was needed for **RQ2.2**.

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 RQ2.2:** 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%**).

### 6.3 Subjective Evaluation by the Participants

For answering **RQ2.3**, 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 12 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*.

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



Table 12 Subjective Evaluation of *iFL* per Bug

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

| Not useful | Little useful | Moderately useful | Very useful | Extremely useful |
|------------|---------------|-------------------|-------------|------------------|
| 3<br>(8%)  | 9<br>(25%)    | 11<br>(31%)       | 7<br>(19%)  | 6<br>(17%)       |

**Fig. 5** Overall usefulness of the *iFL* approach

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. 5. **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.

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 RQ2.3:** 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.

## 7 Think-Aloud Sessions

This section deals with the third stage of our experiments, the think-aloud sessions. The goal of this stage was to gain a better understanding of possible use cases of the *iFL* approach, and not to quantify the success rate of using the tools (which was the goal of previous stages). Hence, in this set of experiments we concentrated on the actual activities of real programmers who have been given access to the *iFL* tool, but we did not explicitly ask them to use it. Participation was completely voluntary and all the data was anonymized.

### 7.1 Why Think-Aloud Sessions?

In this set of experiments, we aimed at involving real, experienced developers who have solid knowledge of basic debugging techniques but who are not biased in any ways by knowing the purpose of the research. This way, we could investigate their spontaneous interaction with *iFL*. However, to be able to achieve this, a very detailed information was required about their activities during problem solving. We aimed at relatively few participants in the experiments but with very deep understanding of their workflows.

There are several potential ways to probe the underlying processes leading to concrete actions of people, but they all have to be indirect. Developers are used to do their job, not to explain it. If they do try to explain (orally or in written form) how they go about their debugging or fault localization workflows, it is quite possible that their account of it will be incomplete or even incorrect. This is because they construct this account from memory. They may be inclined to describe the work process neatly in terms of some formal descriptions that they acquired during their professional training, but the actual work process deviates from these. Psychologists, like (van Someren et al. 1994), have demonstrated that such accounts are not very reliable. Another possibility is to look at the results of their work: the fixes they made and the tools they used. But, in this case, we are looking at the products of the thought processes of these developers, and not at the thought processes themselves.

To overcome these issues we conducted a so-called *think-aloud experiment*, which was explored in detail by (Ericsson and Simon 1980). Informally, the think-aloud process is a methodology which allows to access more reliable information about the thought processing by instructing the participants to verbalize their thoughts during solving the given tasks. We also utilized the practical results of (van Someren et al. 1994) during the implementation of our experimental setup. In this section we will use the terminology found in this article to describe our variation of the think-aloud process.

Note that this approach is a well-known technique in psychology, but comparably more rarely applied in software engineering research (Gopstein et al. 2020; Yamashita et al. 2018; Rojas et al. 2015; Prabhakararao et al. 2003; Wallace et al. 2002).

### 7.2 Experiment Setup

Overall, we recorded the verbalization of solving debugging problems of *six participants* in an empirical assessment consisting of three phases. The whole experiment took five days but the work of the participants was not continuous; the task solving sessions were interrupted with retrospections and preprocessing steps conducted by three authors of this paper without the presence of the participants.

The general overview of the experiment is shown in Fig. 6. There were three phases consisting of one day each (Monday, Wednesday and Friday) when the participants had to solve debugging tasks. The tasks were simply to locate and fix a known bug in a given Java

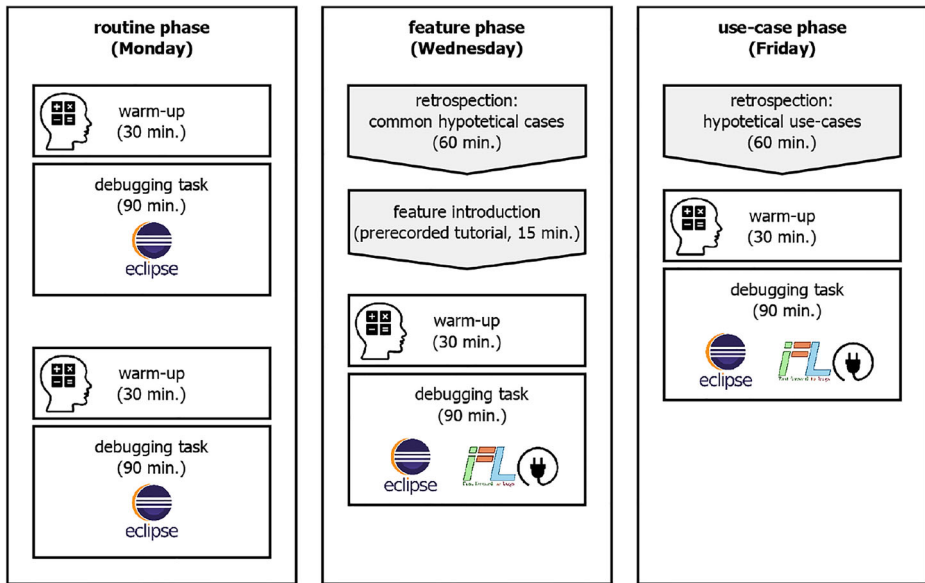


Fig. 6 Overview of the think-aloud sessions

program using the Eclipse IDE, where the bugs were demonstrated by failing test cases. The first phase was the *routine phase*, which consisted of two debugging sessions. The purpose of this phase was to collect data about the unmodified workflow of the participants (not influenced by any factor that related to the experimentation, for example, only a bare IDE was available).

During the second phase, the *feature phase*, we added three tools to the IDE (Eclipse plug-ins) that could potentially help during the debugging and fault localization process, among them *iFL*. The other two tools were a code coverage analyzer and a graph-based static code visualizer. We did not tell the participants any details about the experiment, such as the goal, the purpose of the tools, and naturally, about *iFL* either. Our goal with this was to minimize the bias of the developers and to be able to detect their more reliable responses. In this phase, we briefly explained the main features of the three tools but we did not give them any hints about their potential use cases or actual benefits for fault finding.

Finally, in the *use-case phase* we gave the participants more details about the possible use cases of each tool, while reflecting on their intuitive usage from *feature phase*. This enabled us to observe an evolution of debugging practices of the developers during the course of the three phases, and also to understand their overall workflow routines. Eventually, this data helped us assess how intuitive is to use *iFL*, and in what ways can it help the debugging process.

The *routine phase* included two debugging sessions. We considered the first one as an introductory session with a simple bug only, in which the participants could get used to the target project and the development environment. The second session included a regular bug and served as the comparison basis for the other two phases, which included one debugging session each.

### 7.2.1 Participants

The selection of the participants was conducted in two phases. First one was a voluntary sign-up phase in which we asked the candidates to supply the necessary professional information required for the selection. The selection itself was purely based on experience in software development, in particular Java and Eclipse IDE knowledge. The minimum requirement was a currently professional affiliation as a developer with at least three years of practical experience in the mentioned fields.

During the first phase, we advertised the experiment on various social media platforms, and collected the surveys filled by candidates. We did not disclose any information about the concrete goals of the experiment, we just mentioned that it is about professional developer's daily workflows and gave an overview of the schedule. We selected the six participants after evaluating the survey data, and contacted the participants to discuss the technical details. We also provided a single complementary payment to the participants as compensation for the time spent with the experiment. The amount of payment received was the same for all participants disregarding their performance.

In the following, we will refer to the participants as *Participant 1–6* or P1–P6, as appropriate.

### 7.2.2 Tool Support

We prepared two versions of the Java Eclipse IDE development environment for the participants. During the *routine phase* they used the most recent stable version of Eclipse without any modification. In the case of the *feature phase* and *use-case phase*, we equipped the IDE with 3 additional tools, which were (iFL4Eclipse 2021), (Atlas 2021) and (EclEmma 2021). We used the current stable versions of all of these tools.

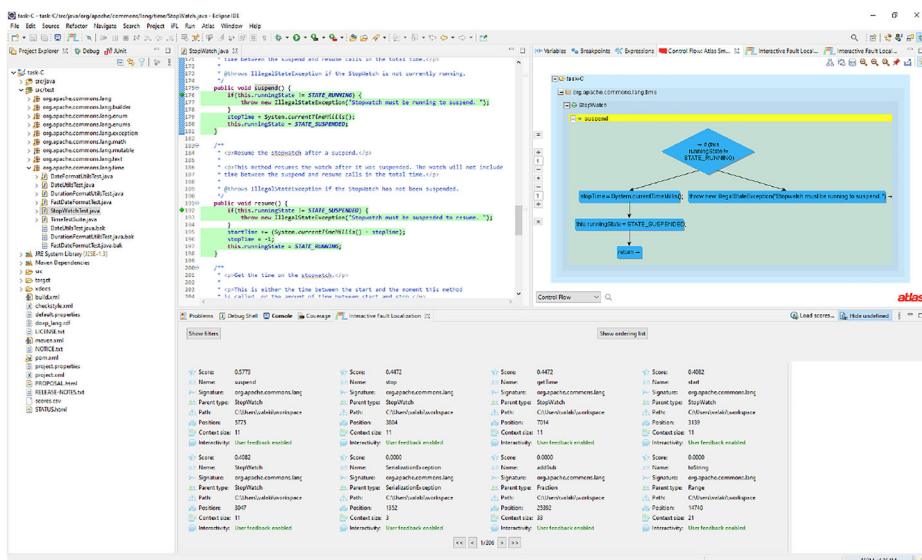
iFL4Eclipse is the implementation of *iFL* as an Eclipse plug-in. Atlas is a tool that lets Java and C users deeply explore their code bases by using visualization of the relationships in the code, files, classes, etc. EclEmma is a free Java code coverage tool for Eclipse which shows code coverage analysis results directly in the Eclipse workbench on the source code. The screenshot in Fig. 7 shows the Eclipse IDE with all three tools activated.

As mentioned, it was not mandatory for the participants to use any of these tools, and in fact, we did not tell them why we included exactly these tools in the IDE at all. Of course, our main focus was to collect information about the *iFL* methodology and its implementation but we did not want this to be evident for the participants, so we included the other two tools “as a distraction” to avoid that the participants give a biased opinion.

### 7.2.3 Task Assignment

As mentioned, each participant had to solve four different tasks during the debugging sessions. The maximum time-frame for each session was 90 minutes, but the participants were allowed to leave earlier if they finished the task. In the debugging tasks themselves we relied on the Defects4J bug database (Just et al. 2014), but we used different bugs than in the previous stages of the experimentation in this paper. In particular, we prepared specific versions of the *commons-lang* project by injecting the failing test cases and set everything up in the Eclipse IDE before the experiments. Table 13 shows the bug identifiers used during various phases.

We relied on our experiences to assess the difficulty of the bugs and select those which could be solved in the specified time frame. The difficulty of the bugs was similar, except



**Fig. 7** Eclipse IDE equipped with additional tools

the first (introductory) task which was very simple (Lang-7b). Since all debugging sessions took place in parallel in the first phase, we used the same bugs for each participant. However, in the second and third phases, the participants were divided in two groups of three and worked in two different parts of the day, but they worked isolated in separate rooms. So, we used two sets of bugs in these two days to prevent that the participants discuss any yet unsolved bugs.

## 7.2.4 Think-Aloud Sessions

We provided the participants with the basic information of the debugging task and asked them to verbalize as much of their thoughts as possible in a continuous flow while solving the problem. During the think-aloud sessions the developer was left alone in an empty room and we checked them periodically (ca. in 10 minutes intervals) to ensure the constant flow of verbalisation. The participants' activities and their immediate surroundings were recorded using screen capture software and a camera with microphone.

We used a two monitor setup (Fig. 8): one for the tasks and another for checking the recoding. During the experiment we tried to minimize the interactions with the participants and did not disturb them while solving their tasks. The only time when we interacted with

**Table 13** Bugs Used during Debugging Sessions

| Phase    | Task(s)           |     |                   |
|----------|-------------------|-----|-------------------|
| Routine  | Lang-7b (Task-A)  | and | Lang-3b (Task-B)  |
| Feature  | Lang-55b (Task-C) | or  | Lang-61b (Task-D) |
| Use-case | Lang-22b (Task-E) | or  | Lang-50b (Task-F) |



**Fig. 8** Dual monitor setup during the experiment

them was in the case of silent work in which case we briefly reminded them to continue speaking.

All of the debugging sessions were preceded with a warm-up session whose goal was to allow the participants to prepare their mindset for thinking aloud. In this step, the participants had to calculate a set of elementary expressions containing basic arithmetic operators and whole numbers using pen and paper. The data collected during the warm-up sessions was not evaluated anyhow.

At the beginning of each debugging session we gave printed handouts to the participants which contained the following information:

- ID of the task (A, B, etc.)
- Failing test case(s)
- Short description (one sentence) of the failure
- Execution instruction about the test suite

Before warm-up and debugging session of the *feature phase* we provided further information about the main features of the three IDE plug-ins, which were introduced during this phase. The participants were asked to watch a 15 minutes long pre-recorded tutorial about the three tools. In this video, we did not give any preferred or discouraged use-cases of the tools, just a brief overview of their features. At the beginning of the debugging sessions in the *feature phase* and *use-case phase*, the subjects got an additional printed handout that



contained a one page brief summary about the features seen in the video as a reminder. These handouts are available at the webpage of the research (iFL 2021)<sup>7</sup>.

We defined a task to be “successfully completed” when the developers found the cause of the failing test cases and managed to fix them. In other words, the participants had to use the Eclipse IDE to debug and fix the programs until all presented unit tests passed. We did not allow them to change the tests, so all fixes had to be made in the system under test. Windows 10 operating systems were used with unrestricted internet connection, but we applied a single restriction that they were not supposed to search the exact bug in the official issue or bug tracking database of the project (these are open source projects). Apart from that any other information source was allowed to be used.

### 7.2.5 Retrospective sessions

With the two retrospective sessions our goal was to gradually “inject” additional information about the *iFL* methodology and the implementation into the participants’ awareness. This means that we did not want to explicitly train them with our views about the approach. Instead, we tried to lead the developers to discover pieces of the approach by themselves. During these sessions we presented various hypothetical cases to the participants and gave them the opportunity to elaborate their thoughts and opinions about these. Six cases were presented to each participant during each of the sessions.

The cases were selected from several predefined templates and then adapted based on the previous responses of the participants. The templates included various references to code elements with some relationships to other parts of the IDE and we also took into account the actions made by the participants during preceding sessions. The researcher tried to be as natural as possible during the dialogue, which meant in practice that we gave lexical knowledge only if the participant explicitly asked for it. We always gave evasive answers to questions about our sentiment or opinion. We also frequently used the interrogative mode to avoid unwanted suggestions (“What do you think?”).

More information on retrospection can be found in Appendix A.

### 7.2.6 Data Preparation and Processing

We processed the collected data in three phases: anonymization, preprocessing/categorization and summarization. The information we obtained was then used to answer our Research Questions, which we present in the subsequent sections.

**Anonymization** From all sessions together, we obtained video material of total length of 33 hours and 57 minutes. Each video frame was composed of two parts: the participants screen area and video footage (Fig. 9).

We created a transcription of each recorded video in form of subtitles (SRT format) with additional information about gestures and other activities of the participants, such as relevant changes of facial expression, gesticulation, and other environment factors (this is similar to subtitles for the deaf and hard of hearing for movies). After this step, all the videos could be edited to remove identifying information, that is video and voice of the participant, and leave only the screen capture part with the transcript. This format allowed us also to gain

---

<sup>7</sup>The direct URL is the following: <https://interactivefaultlocalization.github.io/pages/raw-data.html#handouts>



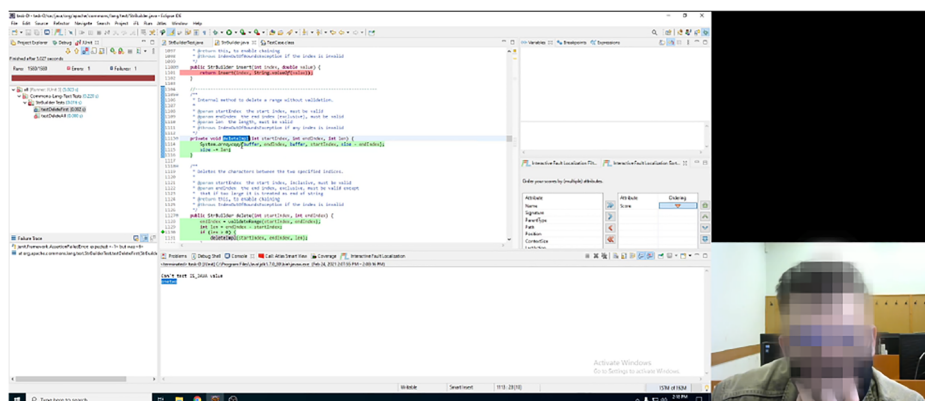


Fig. 9 Sample screen capture during debugging session

access to the information in a searchable way for further evaluation. The transcript files are available at the homepage of our think-aloud research project (iFL 2021).<sup>89</sup>

**Preprocessing/Categorization** Once the transcripts were available, we performed a pre-processing step to obtain the relevant parts of the data for this research.<sup>10</sup> The preprocessing phase started with the manual inspection of the collected data: we watched the videos and determined the aspects and categories based on the witnessed behavior. In the next step we systematically watched all the recordings again, and assigned the appropriate categories to each parts of the experiment. The aspects and categories will be described in the respective sections below.

**Summarization** The categorization results were then summarized with a semi-automatic Python script, which converted the data into easily manageable formats (CSV and GRAPHML files) and to produce the diagrams and tables used in this paper. Both the script and its intermediate results are available at the webpage of the research (iFL 2021)<sup>11,12</sup>.

Finally, we combined the results of the previous preprocessing steps and our notes we made while watching the videos to reach the findings and answers to our Research Questions.

### 7.3 Debugging Workflows and Problem-solving Strategies

In this section, we discuss our findings related to general debugging workflows we observed with the participants (RQ3.1). We emphasize that, in this phase, we were not interested in any particular usage of tools, just the generic debugging routines.

<sup>89</sup>The direct URL is the following: <https://interactivefaultlocalization.github.io/pages/raw-data.html#transcription-of-the-videos>

<sup>90</sup>The researchers' and the participants' mother tongue is Hungarian, so all the raw data is available in this language, while some parts are translated to English.

<sup>10</sup>We plan to use this data in other research projects as well, and not all data are relevant for the present one.

<sup>11</sup>Categorization of the Behaviors: <https://interactivefaultlocalization.github.io/pages/raw-data.html#categorization-of-the-behaviors>

<sup>12</sup>Graphs of Categorization: <https://interactivefaultlocalization.github.io/pages/raw-data.html#graphs-of-categorization>

### 7.3.1 Analysis Aspects

To be able to describe our findings uniformly, we used several different aspects with fixed possible categories for each. We relied on some well-known debugging literature such as the books (Zeller 2009) and (Spinellis 2016), as well as our own experience. Based on the videos from the experiment, the authors discussed what behaviors were visible and which categories these fell into and then made consensual community decisions about categorization, thereby reducing potential subjectivity.

**Debugging strategies** This refers to the basic debugging strategies when trying to locate the causes of a bug. We identified the following categories in this aspect:

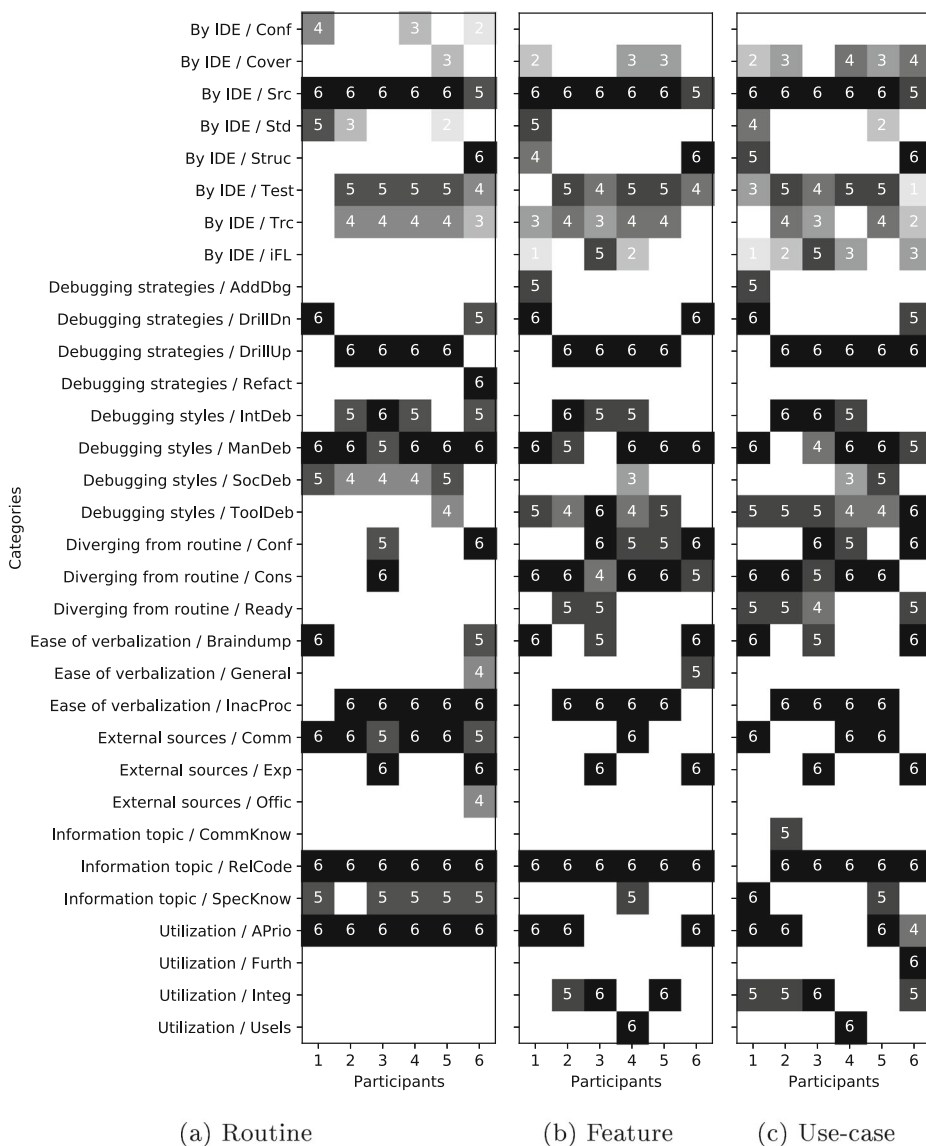
- DRILLUP ► *Drill Up from the Problem to the Bug*: investigating from the failing assertion and roll up the action from there
- DRILLDN ► *Down from the Program's Start to the Bug*: investigating from the beginning of the test case
- DIFFGOOD ► *Find the Difference between a Known Good System and a Failing One*: trying out the method under test with other known-to-be-good inputs
- ADDDBG ► *Add Debugging Functionality*: diverging the data or control flow to force the execution of a certain code chunk to check it
- ADDLOG ► *Add Logging Statements*: temporarily modifying the code by writing out important data
- MINDIFF ► *Minimize the Differences between a Working Example and the Failing Code*: finding the most similar pair of input or control flow, which pass and fail and comparing them
- REFACT ► *Refactoring during debugging*: actually, improving the code during the debugging task

**Debugging styles** This aspect describes the basic approach of the developers to debugging, whether they use only some manual techniques or if they rely on debugging tools and even external help. The following are the main categories for Debugging styles:

- MANDEB ► *Manual Debugging*: debugging without debugger, reading the code, using *println* to retrieve information
- INTDEB ► *Interactive Debugging*: debugging with debugger, using breakpoints and traces
- TOOLDEB ► *Tool-Aided Debugging*: using various tools, which are integrated into the IDE to retrieve additional information
- SOCDEB ► *Socially-Aided Debugging*: using web-searches and other external sources of information during debugging

**Diverging from routine** During the course of the three phases of our think-aloud sessions, the developers were gradually introduced to the possible usage of various debugging tools, including *iFL*. With this aspect, we were interested to see how open the participants are in trying out new methods and tools, and how much they rather stick to their routine workflows. Thus, this aspect mostly relates to our research methodology rather than the debugging workflow itself. We identified the following main categories:

- **READY** ► *Ready to Assimilate New Tools and Information*: eager to learn and experience new methods and technologies, read the handouts as soon as possible, critically evaluate and integrate the new information into the problem-solving
- **CONS** ► *Conservative*: general mistrust towards new information, read the handouts at the beginning, but cannot (or will not) use the information
- **CONF** ► *Conformist*: read the handout as soon as possible, try to follow the instruction precisely using exactly the same steps as instructed



**Fig. 10** Analysis of think-aloud sessions - Category manifestation

Apart from Diverging from routine, there is another aspect which affected the think-aloud methodology, *Ease of verbalization*, which refers to the way the participants were able to express their thoughts during the experiment. This is important for interpreting the information we extract from the think-aloud sessions, and because it may limit the successfulness of this process, we list and detail this aspect as a possible threat to the validity in Section 9.

### 7.3.2 Analysis Results

We will use the diagram in Fig. 10 to give an overview of the analysis results of the session recordings. It contains the information for all aspects and their categories, not only the ones from this section, and the structure of the figure is the following. The y-axes represent various aspects and their categories, delimited with a slash, e.g. By IDE / Cover. The x-axis lists the participant IDs. Each cell represents the frequency of one of the possible categories of a particular aspect of analysis for one of the participants. Note that a particular participant can have more categories assigned from a given aspect if all of them are applicable. The gray-scale level of a dot represents the relative frequency of a given category: black is the most dominant, while light-gray is almost absent during the workflow of that particular participant. The numbers in the cells denote the same thing as the shades: 6 is the darkest, hence most common, while 1 is the lightest, hence rarest. The diagram has three panels arranged vertically, which correspond to the three phases of the experiment: routine workflow with no tools demonstrated, second assignment after tools have been demonstrated, and the final assignment after the use cases have been discussed.

We can observe that there is only a slight change in the overall categorization across the three phases, which indicates that the participants typically did not change their debugging habits throughout the experiment. Table 14 reflects this as well in which we can see the counts of Diverging from routine aspect for each of the phases. Conservative is the most frequent category, especially in the second and third phases, and only slight increase in Ready to Assimilate New Tools and Information can be observed by the end of the experiments.

It is also visible that some of the debugging strategies and styles were very common among the participants, such as Drill Up from the Problem to the Bug and Manual Debugging. Table 14 presents these occurrences in detail. As expected, in the second and third phases Tool-Aided Debugging cases increased notably, but at the same time Socially-Aided Debugging was typical only in the first phase.

The most typical debugging strategies and styles applied by the participants are summarized in the first and second columns of Table 15. The third column shows the categories of Diverging from routine. We analyzed each participant's responses in detail, and we were able to form a rough image of each of them, which is summarized in Appendix B.

**Answer to RQ3.1:** We basically identified two behavior types during debugging and problem solving. The first is when the participant was open to try new approaches and tools, and the second is when he was reluctant to try out new approaches. It seems that in both cases, developers mostly stayed with the familiar routine. This may be caused by a cognitive bias known as *anchoring*. In some cases, the participants did not use new tools but gave them a try just out of curiosity.

**Table 14** Count of categories during various phases in several aspects

| aspects  | Diverging from routine |      |       | Debugging strategies |         |         | Debugging styles |        |        |        |         |
|----------|------------------------|------|-------|----------------------|---------|---------|------------------|--------|--------|--------|---------|
|          | Conf                   | Cons | Ready | AddDbg               | DrillDn | DrillUp | Refact           |        |        |        |         |
|          |                        |      |       |                      |         |         |                  | IntDeb | ManDeb | SocDeb | ToolDeb |
| routine  | 2                      | 1    |       |                      | 2       | 4       | 1                | 4      | 6      | 5      | 1       |
| feature  | 4                      | 6    | 2     | 1                    | 2       | 4       |                  | 3      | 5      | 1      | 5       |
| use-case | 3                      | 5    | 4     | 1                    | 2       | 5       |                  | 3      | 5      | 2      | 6       |

**Table 15** Debugging strategies and information usage

| Debugging            |         | Diverging from routine   | Information usage       |                   |                   |
|----------------------|---------|--------------------------|-------------------------|-------------------|-------------------|
| Debugging strategies |         | Debugging style          | By IDE                  | External sources  |                   |
| Debugging style      |         |                          |                         | Information topic |                   |
| P1                   | DrillDn | ManDeb, ToolDeb          | Src, Struc / Std, (iFL) | Comm              | RelCode, SpecKnow |
| P2                   | DrillUp | IntDeb, ManDeb           | Src, Test, (iFL)        | Comm              | RelCode           |
| P3                   | DrillUp | IntDeb, ToolDeb          | Src, iFL                | Exp               | RelCode           |
| P4                   | DrillUp | ManDeb, IntDeb           | Src, Test, Trc, (iFL)   | Comm              | RelCode           |
| P5                   | DrillUp | ManDeb, ToolDeb / SocDeb | Src, Test               | Comm              | RelCode           |
| P6                   | DrillDn | ManDeb, ToolDeb / IntDeb | Struc, Src, (iFL)       | Exp               | RelCode           |

## 7.4 Information Sources Utilized during Debugging

In this section, we investigate the information sources developers used during problem solving (RQ3.2). We emphasize that in this part we also noted the usage of *iFL* tools, but in this section we do not present the related results. Rather, we will summarize *iFL* related aspects in the next section.

### 7.4.1 Analysis Aspects

Similarly to the previous section, we identified several aspects with possible categories for each. We composed these based on our previous experience and on the outcomes of the debugging sessions as well as the categorization based on behaviors (experienced in videos) similarly to the Section 7.3. We basically divide the information sources by the IDE or external sources. If the developer successfully performed a task without any of these categories we assume that this knowledge was *a priori*.

**By IDE.** This aspect covers basically all information that is accessible within the Eclipse IDE including the built-in tools and our three additional tools:

- SRC ► *Static Source Code*: everything that could be retrieved from the source code editor parts without executing the code
- STRUC ► *Static Structure*: structural data about the static source code such as package and folder hierarchies, e.g., Eclipse’s project explorer and Atlas’s diagrams without executing
- STD ► *stdout, stderr and Log Files*: data which is sent to the standard output or error (e.g., `println`) or any log files
- TRC ► *Stack Traces*: sequences of methods or classes, which were executed one after another, and stack trace during debugging
- COVER ► *Code Coverage*: source code parts that are covered by executing test cases
- TEST ► *Test Results*: information from executing test cases, such as assertions executed, messages and types of exceptions (except trace related information which is covered by previous categories)
- ORDER ► *Ordered Lists*: any ordering over the code elements, e.g., alphabetical list of methods, list of classes by LOC or any other metrics except *iFL* score
- iFL ► *iFL Rank Order*: ordering (rank list) according to *iFL* score
- CONF ► *Settings and Configuration of the IDE or Tools*: for example, classpath, version of the JVM

**External sources.** Information the developers gathered from external sources:

- PERS ► *Personal Connection*: colleague, friend, etc.
- COMM ► *Community*: forums, chats, articles, etc.
- OFFIC ► *Official Sources*: official project web-pages, dedicated user guides, etc.
- EXP ► *This Experiment*: information that the participant obtained during the scope of the experiment and this was explicitly referenced, e.g., developer referring to the video demo or hypothetical cases, reading the handout or the instructions

**Information topic.** Apart from the sources, sometimes it was useful to know the topic itself of the information, which was categorized according to the following:

- COMMKNOW ► *Common Knowledge about Software Engineering*: for example, what is code coverage, how much bits does an integer holds, etc.
- UG ► *User's Guide*: usage details about tools or the IDE
- SPECKNOW ► *Special Knowledge Required for the Task*: this relates to the specific domain of the debugging task, e.g. how regex works or what does “E” means in scientific notation, etc.
- RELCODE ► *Information about Related Source Code*: this category refers to the case when the developer investigated an information that was closely related to the code item in question such as its class, call information, control flow, etc.

An important aspect we observed was that the developers not always utilized the information retrieved despite it could have been useful. This might be related to insufficient knowledge or uncertainty, and could affect the validity of the results. This aspect, called *Utilization*, is discussed in more detail in Section 9.

## 7.4.2 Analysis Results

The results of this analysis can be observed in Fig. 10. The first thing to notice is that there were many occurrences of Static Source Code and Community. Regarding the information topic, Information about Related Source Code was frequent. The said categories did not notably change over the three phases, but an interesting observation is that Special Knowledge Required for the Task was present in the first phase, and then the developers did not use this information. Table 16 provides more details about the actual number of occurrences of By IDE, External sources and the counts of categories under Information topic aspects, respectively.

The last three columns of Table 15 offer the most important categories of information usage for these developers. The findings for the individual participants are summarized in Appendix B.

**Answer to RQ3.2:** The primary source of information developers used during the debugging sessions was unquestionably the source code itself. They started the investigation from the source and used additional information subsequently. So, if any additional information would be available closely from the code (e.g., by various highlights) it would attract the developer's attention. There were three kinds of developer behaviors in terms of how much extra information they used: either using a large context, relying on instructions or using little information only. The kind of context information was mostly the related package structure, call information, and trace data.

## 7.5 Detected Changes of Routine Debugging Workflow due to *iFL*

In this section, we discuss our findings related to how the introduction of *iFL* influenced the debugging routines of the participants. In other words, how much did they manage to use the tool and in which way. This information will help us design further enhancements to the approach and to the tool. As introduced earlier, we added two additional tools to the Eclipse IDE, and we did not tell the participants the purpose of the experiments, which helped reduce the bias of influenced tool usage. As the purpose of this experiment was not to investigate the other two tools, in this section we will concentrate only on *iFL*.



**Table 16** Count of categories during various phases in several aspects

| aspects  | By IDE |       |     |     | External sources |      |     |     | Information topic |     |       |                   |          |
|----------|--------|-------|-----|-----|------------------|------|-----|-----|-------------------|-----|-------|-------------------|----------|
|          | Conf   | Cover | Src | Std | Struc            | Test | Trc | iFL | Comm              | Exp | Offic | Information topic |          |
|          |        |       |     |     |                  |      |     |     |                   |     |       | CommKnow          | SpecKnow |
| routine  | 3      | 1     | 6   | 3   | 1                | 5    | 5   |     | 6                 | 2   | 1     | 6                 | 5        |
| feature  |        | 3     | 6   | 1   | 2                | 5    | 5   | 3   | 1                 | 2   |       | 6                 | 1        |
| use-case |        | 5     | 6   | 2   | 2                | 6    | 4   | 5   | 3                 | 2   |       | 1                 | 2        |

**Table 17** Think-aloud sessions' task successfulness and completion time in minutes (– denotes that the bug was not found)

|            | P1 | P2 | P3 | P4 | P5 | P6 |
|------------|----|----|----|----|----|----|
| 1st task   | A  | A  | A  | A  | A  | A  |
| time (min) | 70 | 18 | 34 | 46 | 13 | 21 |
| 2nd task   | B  | B  | B  | B  | B  | B  |
| time (min) | 81 | –  | 38 | 84 | 66 | 80 |
| 3rd task   | D  | D  | C  | D  | C  | C  |
| time (min) | –  | –  | 18 | –  | 27 | 17 |
| 4th task   | E  | F  | F  | F  | E  | E  |
| time (min) | 86 | 41 | 37 | 59 | 24 | 24 |

**Changes in the workflow** First, we investigate how much the debugging workflow of participants changed in general. Figure 10 shows this information in the form of changes between the different phases of the experiments. Here, we can examine the transition from one category to the other (or the change of frequency) by comparing the columns related to the same participant in each of the panels. We can observe that in most of the cases, the aspect categories did not change notably (as we mentioned for RQ3.1 above related to the anchoring cognitive bias). In Appendix B, we list the notable transitions observed with the participants.

**Usage of *iFL*** About the participants' usage of *iFL*, we can say that three out of the six participants used it successfully in various phases (P1, P3 and P6), two tried it but did not rely on its outputs (P2, P4), and one participant did not use it at all (P5). Only one participant used the interactive features of the tool. However, we do not see this as a very strong critique for the approach given that SBFL in general and the current tool support are mostly unknown to the general public. Column four of Table 15 shows this information (marginal usage is indicated by parentheses). Details for each participant are listed in Appendix B.

**Task solving successfulness** In Section 7.2, we explained that the participants had to solve four debugging tasks, two in the first phase and one in the second and third phases each. Below is a summary of the successfulness of the actual fault finding process of each participant and task. Note that this information is not strictly part of the related Research Question, but might be interesting when comparing the qualitative analysis of the *iFL* method usage.

Table 17 shows the overview of task successfulness for each of the four assignments and each participant. If the task has not been completed, we marked this case with a dash (–), otherwise the completion time is provided in minutes. Unfortunately, task D turned out to be more difficult than anticipated because none of the participants managed to find the bug in time. Apart from this, there was only one case when a developer did not complete the task.

We are aware that this is too few data to generalize, but we can observe a promising trend that the participants who used *iFL* managed to solve the tasks more quickly compared to their colleagues who did not use it. Compare, for example, for the 4th task Participant 3 to Participants 2 and 4 who all had been assigned bug F.

**Summary of findings related to *iFL*.** Based on the above information and on additional discussions with the experiment participants, we can summarize the findings related to *iFL* in the think-aloud sessions as follows.

1. The concept of *iFL* could easily be learned, but in order to be productively used, additional education is required with concrete use cases and explanation of the FL ranking, scores and the interactive features. The experiment participants often expressed verbally that the tool would be useful and they understand it, yet they did not use it properly. Currently, we do not fully understand the reasons for this.
2. *iFL* is most often useful as a “preliminary study” to pre-filter the relevant code parts for investigation.
3. A consequence of the previous point is that since the tool does not integrate very deeply into the usual workflow where the source code editor is the center, often it will be later on neglected by the developer. A possible solution would be to mark the tool outputs directly within the source code.
4. Without proper understanding of the underlying mechanisms (such as the score calculation) the developers will not trust the results of the tool.
5. The developers are reluctant to exclude code parts in the interactive part of *iFL*, even more bigger context parts, if they do not fully understand and believe in the underlying mechanisms of the *iFL* process. For example, it was not clear why would changing the order help in finding the bug more quickly (this could be mitigated using more and better use cases).
6. Often a quite large context could be excluded with *iFL* (not just a class but a complete package), however these kinds of feedbacks require extensive knowledge about the system. This kind of wider knowledge is typically available to system architects and programmers who have a greater overview of the system (which was typical of Participant 6 in our experiment).
7. We learned in all phases of the experiment (even when *iFL* was not available) that the developers excluded some methods during debugging but these were smaller ones, and for bigger ones they looked at them in multiple turns before excluding them.
8. We observed multiple times that the rank order in *iFL* was not really important among the first 3-5 methods. The developers made no distinction between these, and it was only important that they belong to this range. Perhaps another way of presenting the results and not strictly using the order would be more beneficial (while concentrating only on one element at a time).

Overall, the basic concept of *iFL* seems to be acceptable by the developers. But, due to the fact that automated fault localization in general is not known and accepted in an everyday programmer’s work, it requires much more education and training based on real use cases and easily usable tools. This can be contrasted to using a tool like code coverage measurement during debugging, which is much more known to an average programmer, as we learned from this experiment. Note, that this is not a particular problem of *iFL* but of automated fault localization in general. In a similar context, for algorithmic debugging, the authors of the survey (Caballero et al. 2017) say: “Moreover, they are often offered as a separate tool, and the user must switch from their traditional debugger to start algorithmic debuggers. We think that, on the contrary, algorithmic debuggers should aim to integrate with existing debuggers, for instance, allowing the user to switch from a trace debugger to the algorithmic debugger easily, while maintaining the same environment and appearance.” We believe that more research is required about how to facilitate a more open reception of these tools by practitioners, which much better integrate into their usual workflow.

**Answer to RQ3.3:** *iFL* helped the developers to reduce the search space, however, not all participants managed to integrate it into their routine workflows. After suitable training and explanation via use cases the openness increased towards the approach, but it is essential that all the details of inner workings are understood by the developers. Since automated fault localization in general is not widespread, the disbelief in such tools can be high. A number of enhancement possibilities to the approach are available.

## 8 Discussion

This section contains our general views on the interpretation of the empirical data from previous sections.

### 8.1 Conceptual Evaluation Based on Simulations

As discussed, *iFL* achieved notable improvements on both benchmarks; Table 18 summarizes the most important results. For reference, the TALK algorithm proposed by (Gong et al. 2012) achieves about 10-16% improvement at best on some projects from Defects4J, and it produces 20-21 enabling improvements. DStar and Ochiai behaves very similarly and they produce slightly different results compared to Tarantula. However, DStar is very sensitive to user imperfections. Results are comparable for the two benchmarks, despite significant differences in their properties including size and the number of tests compared to the number of program elements.

We confirmed the effectiveness of *iFL* with paired t-test and signed Wilcoxon test to check if the differences in the Expense measures between *iFL* and the basic SBFL over the entire dataset are statistically significant. These tests showed that *iFL* results are significantly better than SBFL at 99% confidence level for each SBFL technique and benchmark.

### 8.2 Possible Improvement Based on Experiments with Real Users

The previously mentioned results concerned the validity and usefulness of the *iFL* methodology from a conceptual point of view. However, there are at least two main practical issues which could hinder its application in real life situations. Namely, do developers have enough information about the context (i. e. the surrounding code base) to be able to give adequate feedback for the *iFL* algorithm? Furthermore, developers have to integrate the proposed methodology and its implementation into their daily workflows. The latter could affect several other factors, not only the raw efficiency as we used during the simulation.

Based on the findings in Sections 6 and 7, we were able to articulate several concrete improvements to the *iFL* methodology and current implementation in the Eclipse

**Table 18** Main results of the iFL algorithm

| Study     | Best FL method | Fault type | All faults | Improvement | Enabling improvement |
|-----------|----------------|------------|------------|-------------|----------------------|
| SIR       | Ochiai         | seeded     | 85         | 71%         | 47 (55%)             |
| Defects4J | Ochiai         | real       | 801        | 79%         | 255 (32%)            |

IDE to address the above issues. We categorize these into three groups: improvements to the methodology, functional enhancements to the implementation and design and other technical improvements to the tool.

In addition to these issues, several participants explicitly expressed their opinion that without accurate background knowledge of the underlying method (such as score calculation) they would be less trustful in the outcomes of the tool. These kinds of issues are related to multiple aspects of the *iFL* methodology and its implementation. Technical details should be explained in the documentation and calculation details need to be accessible from the tool on demand. We could solve these issues in several different ways. For example, we could use various Help features in the IDE, or clarify the concept and principle of score calculation in a detailed training session.

### 8.2.1 Methodology

The main improvements regarding the *iFL* methodology include lowering the granularity level and experimenting with different kinds of contextual data. Currently, in the case of large programs the *iFL* engine works on method level. However, we have seen that users often demand more detailed information during debugging. Since *iFL* is designed to cope with different granularity levels, changing the granularity to line/statement level is fairly straightforward after we eliminate the technical limitations implied by the current coverage measurement approach. As our experiments have shown, users usually incorporate multiple information sources into their debugging process. Along this idea *iFL* itself could be extended to utilize additional data, e. g., function call contexts, as contextual information to improve the effectiveness.

### 8.2.2 Functional enhancements

We could aid the integration of *iFL* into the developers' workflow by implementing features which enhance the user experience of the methodology. For example, a possible improvement is automatic score calculation after test execution. So, by running the tests during or after the (bug)repair, we can modify (upgrade) the score values and get a real-time picture of the change in suspiciousness values. These changes can provide useful information for locating the bug or identifying other faulty (i.e., not yet inspected and corrected) methods.

The degree of trust in the tools or the users' own decisions could impact the willingness of the tools' usage. There may be cases when users approve an interaction and then they regret it afterwards. This could be due to, for example, a "false-click" or because the users have changed their previous judgment or opinion and therefore they want to revoke it. This would be supported by the Undo and Redo functions, which allow the developers to rollback any *iFL* score changes.

Currently, we do not support other IDEs besides Eclipse, so another important goal is to create a plugin for other environments such as IntelliJ.

### 8.2.3 Tool design and technicalities

Experience has shown that the participants' main source of information is the code itself; hence it would be more useful to "bring the information closer" to the source code. One possible way to do this is to visualize it in the code-editor window, for example by code highlight (cf. code coverage). This would greatly help in understanding the *iFL*'s results and

increase usability. This feature may also help us to reduce the overcrowding of method rank lists and the information in them.

## 9 Threats to Validity

### 9.1 Simulation

The main threat to the validity of the results related to **RQ1** is the set of projects and bugs that we used during our experiments. We selected the SIR and Defects4J benchmarks because they are popular in the scene of fault localization research, hence we maximize the comparability of our results to the related works. However, some researchers argue that the projects and bugs in these benchmarks are not representative of real life projects and bugs. We restricted the experiments in this paper to single-fault cases, however this is a current technical restriction, the proposed method itself is not limited in this regard. In addition, multi-fault cases would not have an effect on the results considering the Expense metrics, since (as usual in these experiments) only the faulty code element with the highest rank is considered during the calculation of the evaluation metrics. Considering the experiments related to user imperfections, multi-fault cases may yield different results, because the algorithm may have more ways to modify the scores of faulty and non-faulty code elements as well.

The next threat is that the feedback model of our method is relatively simple, which might not be suitable in a real life scenario. For example, the user might give some more complex feedback, like combining code elements from other contexts or pointing to code elements not suggested in the rank list. Also, as new knowledge is obtained, changes to earlier decisions could be possible. We plan to address this topic in future work.

### 9.2 Quantitative User Study

The first threat that we identified which could impact the validity of the results of our user studies in **RQ2** is the time to perform the tasks. We set the time limit to 25 minutes for each task, which may have an effect on the participants' performance, especially if they have less experience. Also, a longer period of time would have affected the performance as well, since the developers could lose concentration as they get tired.

It is also a possibility that the rank of the faulty code element may have influenced our results. Although, differently ranked faults may yield different results, investigating the correlation between the position of faulty code elements and the success rate or time needed to fix the bugs was not an explicit goal of our experiments. In addition, we tried to select bugs with as different ranks as possible while keeping in mind the distribution of the complexity of the tasks. Also, during the design of the tasks we investigated various projects and we selected bugs for which the developers do not need specific domain knowledge. In other words, we simulated the scenario in which the developer has to debug an unknown program. However, some of the participants may have had knowledge about the projects or bugs, which could have influenced their performance.

An other threat is the so called Hawthorne effect (Franke and Kaul 1978), i. e., the participants might have changed their behavior in response to the awareness of being observed. They may try to figure out the goals of the experiment in which they participate. To mitigate this threat we minimized the amount of information that might have helped the participants

determine the goal of the experiment. However, there is still a chance that they were able to deduce the purpose of the experiment.

A further threat is related to the participants of the studies. They were undergraduate and graduate students, and professional developers. We tried to diversify the list of participants by inviting people with different levels of experience, yet they may not be a representative sample of the population of developers. In addition, we tried to design the experimental groups and assign the tasks to them in such a way that the distribution of participants through the groups is balanced.

### 9.3 Think-Aloud Sessions

We identified several threats which could impact the validity of the think-aloud sessions (RQ3). It is a well known fact that any experimental scenario could have an impact on the behavior of the participants. In our case this means that the unfamiliar working environment and the mandatory development environment could change the preferred actions of the developers during the debugging sessions. We tried to minimize the number of interruptions while the participants were solving their tasks, but they might have noticed that we occasionally checked their verbalization. Many people feel awkward or tense while they are being recorded. We did not find any documented cases when these interruptions occurred, but they still could have affected the participants and our conclusions about their workflows, especially during the *routine phase*.

There are two kinds of biases which we cannot eliminate. The participants may discussed their experiences while they were not inspected. One of the participants showed signs of pressure to perform, even though we explicitly stated that we will not evaluate their performance.

Psychologists have shown that retrospections are not very reliable (van Someren et al. 1994). For this reason, we only used retrospection to inject further information into the subjects' workflow. The main source of information during the evaluation was the recorded think-aloud debugging sessions. Our goal was to use only those types of verbal reports that required the least amount of mental effort from the participants, but also allowed to access their development workflows. These were described as Level 2 verbalization by (Ericsson and Simon 1980), or Intermediate recording into verbal code. This occurs when the internal representation in which the information is originally encoded is not in verbal code, so it has to be translated into that form. The most prominent effect of this kind of verbalization is the slowing down of the task execution. Hence, this could affect the validity of the data presented in Table 17.

**Ease of verbalization** There were a small number of cases when the participants failed to use the required type of verbalization. They either generalized (which could be as much unreliable as retrospections) or they were not able to verbalize certain thoughts. We used the following categories to assess this aspect.

- BRAINDUMP ► *Unprompted "Braindump"*: braindump without instructing to do so, no summation/aggregation/abstraction, continuous flow of information
- INACPROC ► *Inaccessible Processes*: developer speaks continuously but concentrates precisely on the currently available information (reading the code, UI items, etc.) and routine tasks are difficult to verbalize; if the participant stops to think about something, the verbalization is suspended

- **GENERAL** ► *Generalization and Second-Hand Information*: participant provides larger chunks of information but this is not continuous and not precise; the information provided is often that what is generally expected with little to no concrete data, and a higher level of generalization is typical

As seen in Fig. 10, the amount of Generalization and Second-Hand Information is negligible, but almost all participants had some issues with Inaccessible Processes. These cases occurred when the tasks reached a certain level of complexity, which vary from participant to participant. We managed to clarify these cases using the screen captures, but it is possible that we misinterpreted their actions.

**Utilization** We classified the availability and the quality of collected data from a different aspect too. This aspect represents how well the participants are able to utilize the information provided by various sources, such as *iFL*. We defined the following categories.

- **IGN** ► *Ignoring*: the developer ignored the information, e. g., the list of suspicious code elements provided by *iFL*
- **USELS** ► *Consider Useless*: the developer checked the new information, but did not use it based on his own decision, e. g., because did not trust the score value, or all recommended methods seemed equally suspicious
- **FURTH** ► *Need Further Information or Feature*: the developer checked the new information, but did not use it, developers acknowledge the usefulness, but miss some key features or information to utilize
- **INTEG** ► *Integrated Usage*: the developer used the new information during the debugging and changed his routine workflow to accommodate the new possibilities
- **APRIO** ► *A Priori Usage*: the developer has previous experience with the information or tool

Figure 10 shows that most of the participants act according to the category of A Priori Usage, even in the *feature phase* and *use-case phase* after they already had access to the new tools. This means that they rarely integrated new information into their routine workflow. It is beyond doubt, that the integrated usage of *iFL*, was far from dominant, but we could still identify several relevant cases. Almost all of the participants understood the basic concepts of *iFL*, and some of them were able to partially integrate it into their workflow. We cannot assess the distribution of these cases in general from the collected data, but our data clearly shows that there are developers who are able and willing to utilize the *iFL* approach.

Finally, a threat to the generalization of our findings is that, due the low number of participants in the think-aloud experiment we cannot support our findings with appropriate statistical data. However, we aimed at better understanding the developers' thought processes and this experiment also helped us identifying the weaknesses of the approach, and gave us possible directions for future enhancements.

## 10 Related Work

SBFL methods are still finding their way to be employed in practice (Le et al. 2013; Steimann et al. 2013; Kochhar et al. 2016; Abreu et al. 2009). For instance, most studies are carried on using artificial faults (Pearson et al. 2017), and still the faulty element is usually placed far from the top of the rankings (Xia et al. 2016; Parnin and Orso 2011). (Le et al. 2013) showed in their study that there is a gap between theoretical and practical results.



Since SBFL heavily relies on the coverage and the pass/fail information, test suite properties directly affect fault localization (Baudry et al. 2006; Yu et al. 2008). The Zoltar (Janssen et al. 2009) and GZoltar (Ribeiro and Abreu 2010; Campos et al. 2012) tools provide the ranked list of diagnosis candidates to help the user in practice. The FLINT method by (Yoo et al. 2013) improves the effectiveness of fault localization using an entropy-based approach. This approach resembles our idea of trying to involve the “statistical dispersal” of the test cases in terms of code coverage to find the most appropriate tests. Both approaches improve the fault localization process by carefully selecting the next test cases to execute after each step.

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. 2016; Wong and Debroy 2009) and (Parmar and Patel 2016).

Some debugging approaches are loosely related to the topic of this article, where user feedback may be incorporated, for example the works of (Zeller 2009) and (Kiss et al. 2017) on Delta Debugging, as well as algorithmic debugging and testing (Silva 2011).

The closest related works from the other Interactive Fault Localization methods (Gong et al. 2012; Hao et al. 2009; Bandyopadhyay and Ghosh 2012; Lei et al. 2012; Li et al. 2016; Li et al. 2018) to our approach are the ones that change the ranking of program elements based on the user feedback iteratively (Gong et al. 2012; Hao et al. 2009; Lei et al. 2012). Other than utilizing user feedback, these papers also incorporate the Siemens suite from SIR into their set of subject programs. 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, however, we managed to re-implement the work of (Gong et al. 2012).

For reference, (Lei et al. 2012) 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. 2009) 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. 2012), 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 (Abreu et al. 2009). For a more precise comparison which includes real bugs, we reimplemented the TALK algorithm proposed by Gong *et al.* in our simulation framework and re-executed our experiments. We found that TALK improves the rank of the faulty elements by 1-3 positions which translates to 2-6% improvement on Defects4J.

(Li et al. 2016; Li et al. 2018) uses a concept of contextual knowledge that is similar to ours. 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 2012) proposed a method to iteratively predict and remove coincidentally correct test cases based on user feedback.

(Fry and Weimer 2010) 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.

In the last phase of our experimentation, we used think-aloud protocols. These techniques have a notable background in psychology and other areas. (Ericsson and Simon 1980) showed that verbalizing information is shown to affect cognitive processes only if the instructions require verbalization of information that would not otherwise be attended to. The inaccurate reports found by other research are shown to result from requesting information that was never directly heeded, thus forcing the participants to infer rather than remember their mental processes. In our case this means that we can safely rely on the data collected during the previously described think-aloud protocols. (van Someren et al. 1994) gives a detailed description about these processes and also a guide for practical implementation for social scientists who want to use the think aloud method for research on cognitive processes and knowledge engineers who want to use the method for knowledge acquisition. Our research can be viewed as a specialized case for the latter: we wish to capture various details about the developers' workflow to partially integrate their expert knowledge into our *iFL* methodology.

Think-aloud techniques are often used to study other mental processes. For example, learning processes in the context of problem-solving are studied by (Anzai and Simon 1979). In this study the think aloud method is used to identify changes in the knowledge during repeated problem-solving on a single task. Since debugging involve new knowledge acquisition during frequently occurring tasks, this work is a good example, which supports the usage of think-aloud protocols in these areas.

Nonetheless, in software engineering, think-aloud sessions are comparably rarely used. The most important field where this approach is successfully applied is user interface design and usability testing (Lewis 1982; Fan et al. 2020; Nørgaard and Hornbaek 2006; McDonald et al. 2012; Olmsted-Hawala et al. 2010). For debugging, practitioners have developed the so-called “rubber duck debugging” techniques that can ease the verbalization of the debugging process, which can result in hitting upon the solution in the process of explaining the problem (rubberduck 2021). Examples where think-aloud sessions in empirical software engineering research have been used include program comprehension (Gopstein et al. 2020; Yamashita et al. 2018), and test generation (Rojas et al. 2015).

The most related approaches to our think-aloud method we found are (Prabhakararao et al. 2003; Wallace et al. 2002), but this relates to a different domain and fault localization methods. Understanding debugging processes is usually achieved by monitoring IDE usage (Shynkarenko and Zhevaho 2020; Yamashita et al. 2018; Schröer and Koschke 2021). For understanding the mental processes during debugging and specifically fault localization with think-aloud sessions, we think that our approach is the first.

## 11 Conclusions

In this work, we presented *iFL* – a new form of an Interactive Fault Localization approach. This approach extends 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.

With the three-staged approach to empirical evaluation of the proposed approach, we tried to investigate the potential in the *iFL* techniques from many different angles, but this might still not represent real life scenarios fully. However, in all three stages we obtained promising results. 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. In addition to non-interactive approaches, *iFL* also outperformed TALK, a closely related interactive FL algorithm by a huge margin. Although *iFL* does not seem to help in localizing more bugs when it comes to real users, the fault localization times were reduced significantly (except for the very simple cases), and subjective feedback was also mostly positive about our experimental implementation. Finally, in our think-aloud sessions we found that experienced developers understood the benefits of the approach and some of them were able to successfully integrate the approach into their workflows. However, further research is required to find out how the methodology involving the interactivity should be made more intuitive, and the supporting tools more directly related to the source code, the centerpiece of the developer's attention during debugging. This would enable better acceptance by practitioners without extensive training and explanation through use cases.

The interested reader can find more information about the data and software used in this paper on GitHub (*iFL4Eclipse* 2021; *iFL* 2021) and in the following archives: (Horváth et al. ; 2020b).

## Appendix : A Details about Retrospective Sessions

Psychology has shown that the data obtained by retrospection is not always valid. People may not report thoughts that they have clearly had before and they will also report false memories: thoughts that they cannot have had at the time (van Someren et al. 1994). For this reason we only used retrospection to inject further information into the participant's workflow. The main source of information during the evaluation was the recorded think-aloud debugging sessions.

### A.1 Discussion of *iFL* Methodology Related Hypothetical Cases

During the retrospection of the *feature phase* we presented various general and software engineering related hypothetical cases. On average, the participants had 10 minutes to reflect on each of these. The cases were composed along the following informal properties.

#### Assumable sentiment about the changed situation

- Positive
- Natural
- Negative

#### Assumable usefulness for debugging

- Harmful
- Irrelevant
- Helpful

The cases were assigned randomly to each participant but the algorithm we used tried to cover as much of the above properties as possible.

The goal of this session, hence the main focus of these cases, was to inject information about the methodology of the *iFL* without revealing any details about its implementations. Consider, for example, this case (translated version): “*Let us suppose that before debugging on Monday (incorrect parsing of double from string), you heard that the NumberUtils.createNumber() method was written by a junior college, who has very little experience.*”. This is very similar to the situation when the *iFL* algorithm suggests (with a high score) that the inspected method is suspicious.

## A.2 Discussion of Use-case Related Hypothetical Cases

During the second retrospection (in the *use-case phase*) we presented six hypothetical use-cases about the plug-ins. Note that at this point the participants already had some knowledge of the features of these tools, but they did not get any information about the recommended or discouraged usage of them.

Similarly to the previous hypothetical cases in the *feature phase*, we could categorize the use-cases according the following properties.

### Assumable sentiment about the changed situation

- Positive
- Natural
- Negative

### Related tool

- ECL Emma
- Atlas
- iFL

Similarly to the above, the cases were assigned randomly to each participant but the algorithm we used tried to cover as much of the above properties as possible.

We wanted to avoid that the participants just automatically recite the use-cases we present. To achieve this we refrained of declaring any use-cases as good or bad, and the participants did not have to choose either. In practice, this means that most of the use-cases were open-ended. For example, “*Let us suppose that the methods were sorted according to their scores in descending order. You found the NumberUtils.createNumber methods at the first place. After inspecting it, it seems to be correct but you are not sure about this decision. So you decrease its score by 66% but later you discover it to be the faulty method.*” As this example illustrates, our goal was to “force” the participants to think about the possible use-cases, which could have helped them to integrate the new information into their workflow, without explicitly telling them our versions.

## Appendix : B Participant Details in Think-Aloud Sessions

In this section, we provide the analysis of participant responses according to the criteria set in the corresponding parts in Section 7. In addition, to make it easier to relate our findings to the workflows exhibited by the participants, we assigned short but descriptive names to workflows we observed. With these names our goal was not to try to generalize debugging

styles, but we merely wanted to summarize the connection between our findings and actions of the participants. This will help us explain the usage of *iFL* given the observed workflows.

## Participant 1

**Debugging Workflows and Problem-solving Strategies** This participant mostly performed manual debugging with some internet search and very light tool usage. His favorite strategy was Down from the Program's Start to the Bug. The developer is mostly conservative and believes that debugging functionalities offered by the IDE and additional tools provide little additional benefits over a println-based approach. The participant tried to use the tools only when no other ideas worked.

**Information Sources Utilized during Debugging** This developer investigated the source code a lot, and took usage of code coverage information frequently as well. As an external information, he used programmers' forums a couple of times. The topic was mostly about the related code.

**Changes in the workflow** Regarding debugging styles, this participant soon started using Tool-Aided Debugging, however did not rely on the tools much. In terms of information sources, he started using *iFL* Rank Order in the second phase, and moved from stdout, stderr and Log Files to Static Structure and from Information about Related Source Code to Special Knowledge Required for the Task.

**Usage of *iFL*** This participant already tried to use *iFL* in the second phase, but at first he did not find it useful compared to his own intuition. In the third phase, the first step of the debugging process was to look at the results *iFL* provided (but without the interactivity). In essence, this developer uses *iFL* as a pre-selection before investigating the methods in more detail. He expressed his excitement when a new candidate method appeared on the *iFL* list, which was not known to him. In another case, he used the tool as a confirmation of a method, which he found suspicious intuitively.

**Summary of Workflows** We named this workflow the "*un-industrialized comfort zone*", which is shared with Participant 4. These developers used a debugging technique heavily based on static and manual code analysis. These represent the most challenging scenarios for the *iFL* tool and methodology, and we assume that for these participants this would be the same for any new tools or information sources. The fact that Participant 4 used the *iFL* tool only marginally supports this assumption. Any new tools would need to have very well pronounced advantages in order for these developers to abandon their well established practices.

## Participant 2

**Debugging Workflows and Problem-solving Strategies** The participant started with mostly manual approach (using println, commenting code, etc.), but quickly switched to using the IDE-provided interactive debugging aids and the tools provided in the experiment. Debugging strategy was mostly Drill Up from the Problem to the Bug. He was eager to use the tools provided but always fell back to the manual approach.

**Information Sources Utilized during Debugging** Most of the information this developer gained was from the source code and the related JavaDoc documentation. Apart from this, he used a little internet search as well. The topics of information were code call information and coverage.

**Changes in the workflow** This participant used various debugging styles in the first two phases but mostly Socially-Aided Debugging, from which he gradually changed to Tool-Aided Debugging. The developer started using iFL Rank Order in the third phase.

**Usage of iFL** This developer tried to use *iFL* right at the beginning of the session but failed at first because he could not interpret the error message he received (the project was not selected before applying the tool). He tried to make use of the rank list but did not try the interactive features. Overall, the tool did not help him, and he expressed his distrust towards the whole approach because he did not understand how the scores and the ranks are calculated.

**Summary of Workflows** Most of the actions of this participant could be interpreted as a consequence of using “*dynamic analyses*”, like interactive debugging, breakpoints and watch-windows. These tools usually involve the integrated usage of several tools (including default ones in the IDE). We anticipate that the presence of these tools could be one of the reasons that this participant was more ready to use a complex methodology and tool like *iFL* (regardless of the successfulness).

### Participant 3

**Debugging Workflows and Problem-solving Strategies** This developer applied various debugging styles, but he used mostly interactive approach with some internet search and other manual techniques, with Drill Up from the Problem to the Bug style. In terms of diverging from routine, he was mostly conformist but was open to try new tools, especially *iFL*.

**Information Sources Utilized during Debugging** This participant heavily relied on the source code and information from the test runs (assertions and traces). He also used the experiment handouts in much detail. This developer investigated an extensive context related to the code element, and tried to understand it.

**Changes in the workflow** The programmer gradually changed his debugging style from Manual Debugging to Interactive Debugging then to Tool-Aided Debugging. His Diverging from routine gradually changed from Conservative to Conformist and from Conformist to Ready to Assimilate New Tools and Information, but he also showed signs of the first two categories in the later phases as well. This participant started using iFL Rank Order already in the second phase and continued it in the third one as well. The usage of *iFL* was very pronounced as well.

**Usage of iFL** This participant managed to make use of *iFL* to a large extent. The workflow he applied was exactly the intended one: he looked at the methods in the ranking order one by one and investigated those methods in this order. Alas, the interactive features did not help this developer. Also, the usage of some features was not clear for him such as the way of reordering the elements.

**Summary of Workflows** During the course of solving the tasks, this developer made slight turns to use of various tools and new information sources, but he almost always returned to the static inspection of the source code. We named this behavior the “*explorations around the source code*”. During these turns the developer briefly looked at tools like the interactive debugger and visualizations, but these did not play a major role in his workflow. Our expectation (supported by actual usage) is that the integration of the *iFL* approach could be aided by placing well selected features either close to “home” (source code) or along the main workflow paths (e. g. automatically start the *iFL* inspection after test executions).

## Participant 4

**Debugging Workflows and Problem-solving Strategies** This participant also used a combination of manual and interactive debugging with Drill Up from the Problem to the Bug style. He tried to use all the tools provided, even searched for some additional information on the internet, but did not rely on the results provided.

**Information Sources Utilized during Debugging** This developer tried to use the outputs provided by the tools but then did not use them. However, the use of community information and internet searches was very frequent with this participant.

**Changes in the workflow** This participant changed his Debugging styles from Socially-Aided Debugging to Tool-Aided Debugging. In terms of information sources, he changed from Stack Traces to Code Coverage and from Code Coverage to Ordered Lists in the third phase. The developer tried to use *iFL* Rank Order in the second phase, which was a bit more enhanced in the third one.

**Usage of *iFL*** This participant looked at *iFL* and spent some time on the ranking list but did not use it effectively (nor the interactive features). However, this was true for all of the three additional tools in Eclipse. It seemed that he investigated the tools only from curiosity and did not bother to change his usual workflow.

**Summary of Workflows** This workflow is shared with Participant 1, “*un-industrialized comfort zone*”, whose description can be found there.

## Participant 5

**Debugging Workflows and Problem-solving Strategies** The participant mostly relied on manual techniques (commenting out code, manual results calculation) with some internet search and very light tool usage. This developer was very conservative, and mostly relied on the same approach over all three phases.

**Information Sources Utilized during Debugging** This participant relied on the source code and information from the test runs (assertions and traces), but also used information written to output and log files with a little internet searches.

**Changes in the workflow** This participant changed his Debugging styles from Socially-Aided Debugging to Tool-Aided Debugging then back to Socially-Aided Debugging.



**Usage of *iFL*** This participant did not use *iFL* at all (he looked at one of the other tools but only briefly). Interestingly, the developer mentioned in one of the interviews that he understood the purpose of the tool and that he would use it.

**Summary of Workflows** The presumed key concept during task resolution of this participant was his loyalty to known tools and practices: “*industrialized comfort zone*”. Despite that this developer did not use *iFL*, this workflow already utilizes several tools and methodologies, which could serve as a basis to introduce and finally integrate more complex techniques like *iFL*.

## Participant 6

**Debugging Workflows and Problem-solving Strategies** This participant used many different approaches for problem solving, he even made some refactoring on the code before trying to find the bug. The developer tried to understand the system at great depth. He first used manual approach and switched to more advanced interactive features or tools only when needed. His favorite strategy was Down from the Program’s Start to the Bug, but used others as well. In terms of diverging from routine, this participant was mostly conformist but in his case the routine means to try out as much tool as possible and to see if they can bring some benefit.

**Information Sources Utilized during Debugging** The developer first tried to understand a large extent of the code, including its overall structure, before debugging the code part in question. He also used community forums and the provided handouts as external sources. The most important type of information this participant relied on was the structure of the related packages and their contents.

**Changes in the workflow** This participant transitioned to Tool-Aided Debugging from Manual Debugging in the third phase, and was the only one who changed his debugging strategy (from Refactoring during debugging to Down from the Program’s Start to the Bug and then to Drill Up from the Problem to the Bug). His Diverging from routine changed in the third phase from Conservative to Ready to Assimilate New Tools and Information and his information sources changed from Test Results to Code Coverage. The developer started using *iFL* Rank Order very actively in the third phase.

**Usage of *iFL*** This participant was the most active user of *iFL*. In fact, he was the only one who followed the intended usage including the interactive features. In particular, he used this part to exclude the non-suspicious methods. This developer focused on a relatively large context, including almost the whole package, and he first tried to understand the context as much as possible before proceeding to the actual debugging task. He also listed several improvement suggestions for the tool and the process. An important observation is that the developer started using the tool more actively after he understood the details of how the scores and ranks are calculated.

**Summary of Workflows** We found that the workflow of this developer was similar to the previous one but employed more advanced features, so we refer to it as “*high-tech architect*”. During debugging, he already utilized several concepts and tools usually related to the system or project level view on software engineering. The developer actively sought new tools to aid his tasks, which could provide a fertile ground for new tools like *iFL* (which



he integrated during the *use-case phase*), even if these tools employ lesser known concepts. However, these high level techniques sometimes stir this workflow away from the solution, which could be possibly solved by more careful tool design. One of our long-term goals with the *iFL* project is to try to integrate different other contextual information so that they could bridge this gap.

**Acknowledgements** We would like to thank Rita Bártfai and Dávid Horváth for their contribution to the development of the *iFL* for Eclipse. We would also like to thank Rita Bártfai, Filip Opauszki, Sándor József Savanya and Marianna Stiller for their help during the preparation of the audio and video data of think-aloud sessions.

László Vidács was also funded by the János Bolyai Scholarship of the Hungarian Academy of Sciences. Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. The research was supported by the Ministry of Innovation and Technology NRD Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004).

**Funding** Open access funding provided by University of Szeged.

## Declarations

**Conflict of interest** The authors declare that they have no conflict of interest.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- iFL (2021) Research about interactive fault localization. <https://interactivefaultlocalization.github.io/>. Accessed 01 April 2021
- Abreu R, Zoetewij P, Golsteijn R, van Gemund AJC (2009) A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* 82(11):1780–1792
- Abreu R, Zoetewij P, van Gemund AJC (2007) On the accuracy of spectrum-based fault localization. In: *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pp 89–98
- Anzai Y, Simon HA (1979) The theory of learning by doing. *Psychol Rev* 86(2):124–140. <https://doi.org/10.1037/0033-295X.86.2.124>. <https://psycnet.apa.org/record/1979-27801-001>
- Atlas (2021) Ensoft — atlas for java and c - understand code someone else wrote!. <https://www.ensoftcorp.com/atlas/>. Accessed 12 March 2021
- B Le T-D, Lo D, Le Goues C, Grunske L (2016) A learning-to-rank based fault localization approach using likely invariants. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, pp 177–188
- Balogh G, Horváth F, Beszédés A (2019) 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*, pp 371–374
- Balogh G, Schnepfer Lacerda V, Horváth F, Beszédés A (2019) iFL for Eclipse – a tool to support interactive fault localization in Eclipse IDE
- Bandyopadhyay A, Ghosh S (2012) Tester feedback driven fault localization. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp 41–50

- Baudry B, Fleurey F, Le Traon Y (2006) Improving test suites for efficient fault localization. In: 28th international conference on Software engineering. ICSE '06. ACM, pp 82–91
- Caballero R, Riesco A, Silva J (2017) A survey of algorithmic debugging. *ACM Comput. Surv.* 50, 4
- Campos J, Ribeiro A, Perez A, Abreu R (2012) 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, p 378
- Corritore CL, Wiedenbeck S (1999) Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies* 50(1):61–83. <https://doi.org/10.1006/IJHC.1998.0236>
- Davies SP, Gilmore DJ, Green TRG (1995) Are Objects That Important? Effects of Expertise and Familiarity on Classification of Object-Oriented Code. *Human-Computer Interaction* 10(2-3):227–248. <https://doi.org/10.1080/07370024.1995.9667218>
- Do H, Elbaum S, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Emp. Softw. Eng.*, 10,4
- EclEmma (2021) Eclemma - java code coverage for eclipse. <https://www.eclemma.org/>. Accessed 23 March 2021
- Ericsson KA, Simon HA (1980) Verbal reports as data. *Psychol Rev* 87(3):215–251. <https://doi.org/10.1037/0033-295X.87.3.215>
- Fan M, Wu K, Zhao J, Li Y, Wei W, Truong KN (2020) VisTA: Integrating Machine Intelligence with Visualization to Support the Investigation of Think-Aloud Sessions. *IEEE Trans Vis Comput Graph* 26(1):343–352. <https://doi.org/10.1109/TVCG.2019.2934797>
- Franke RH, Kaul JD (1978) The hawthorne experiments: First statistical interpretation. *Am Sociol Rev* 43(5):623–643. <http://www.jstor.org/stable/2094540>
- Fritz T, Murphy GC, Hill E (2007) Does a programmer's activity indicate knowledge of code? In: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2007, pp 341–350. <http://doi.acm.org/10.1145/1287624.1287673>
- Fritz T, Ou J, Murphy GC, Murphy-Hill E (2010) A degree-of-knowledge model to capture source code familiarity. In: Proceedings - International Conference on Software Engineering, vol 1, pp 385–394. [www.nongnu.org/cvs](http://www.nongnu.org/cvs)
- Fritz T, Shepherd DC, Kevic K, Snipes W, Bräunlich C (2014) Developers' code context models for change tasks. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering 16-21-Nove:7–18. <https://doi.org/10.1145/2635868.2635905>
- Fry ZP, Weimer W (2010) A human study of fault localization accuracy. In: 2010 IEEE International Conference on Software Maintenance, pp 1–10
- GCOV (2021) gcov—a test coverage program. <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>, Last visited: 2020-08-13
- Gong L, Lo D, Jiang L, Zhang H (2012) Interactive fault localization leveraging simple user feedback. In: IEEE International Conference on Software Maintenance, ICSM. IEEE, pp 67–76
- Gopstein D, Fayard AL, Apel S, Cappos J (2020) Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion. ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 605–616
- Hao D, Zhang L, Xie T, Mei H, Sun J-S (2009) Interactive Fault Localization Using Test Information. *J Comput Sci Technol* 24(5):962–974
- Horváth F, Beszédes A, Vancsics B, Balogh G, Vidács L, Gyimóthy T Data for Experiments with Interactive Fault Localization Using Simulated and Real Users. <https://doi.org/10.5281/zenodo.4658314>
- Horváth F, Beszédes A, Vancsics B, Balogh G, Vidács L, Gyimóthy T (2020) Experiments with interactive fault localization using simulated and real users. In: Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME'20), pp 290–300
- Horváth F, Beszédes A, Vancsics B, Balogh G, Vidács L, Gyimóthy T (2020) Supplemental Material for Experiments with Interactive Fault Localization Using Simulated and Real Users, figshare
- iFL4Eclipse (2021) iFL 4 Eclipse. <https://github.com/InteractiveFaultLocalization/iFL4Eclipse>. Accessed 01 April 2021
- Janssen T, Abreu R, van Gemund ArjanJC (2009) Zoltar: A toolset for automatic fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, pp 662–664
- JavaParser (2021) Javaparser - homepage. <https://javaparser.org/> Accessed 25 October 2021
- Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proc. of International Conference on Automated Software Engineering. ACM, pp 273–282


- Just R, Jalali D, Ernst MD (2014) 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, pp 437–440
- Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G (2014) Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, pp 654–665
- Kiss A, Hodován R, Gyimóthy T (2017) Coarse hierarchical delta debugging. In: Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 194–203
- Kochhar PS, Xia X, Lo D, Li S (2016) Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016. ACM Press, New York, USA, pp 165–176
- Latoza TD, Garland D, Herbsleb JD, Myers BA (2007) Program comprehension as fact finding. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2007. 361–370
- Le T-DB, Thung F, Lo D (2013) Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization. In: 2013 IEEE International Conference on Software Maintenance, pp 380–383
- Le X-BD, Bao L, Lo D, Xia X, Li S, Pasareanu C (2019) On reliability of patch correctness assessment. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp 524–535, <https://doi.org/10.1109/ICSE.2019.00064>
- Lehmann D, Pradel M (2018) 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. ESEC/FSE 2018. ACM, New York, NY, USA, pp 610–620
- Lei Y, Mao X, Dai Z, Wei D (2012) Effective fault localization approach using feedback. IEICE Trans Inf Syst E95.D(9):2247–2257
- Lewis C (1982) Using the “thinking Aloud” Method in Cognitive Interface Design. IBM Research Report, RC-9265 9265:12. [https://books.google.it/books/about/Using\\_the\\_thinking\\_Aloud\\_Method\\_in\\_Cogni.html?id=F5AKHQAAAJ&redir\\_esc=y](https://books.google.it/books/about/Using_the_thinking_Aloud_Method_in_Cogni.html?id=F5AKHQAAAJ&redir_esc=y)
- Li X, d'Amorim M, Orso A (2016) Iterative User-Driven Fault Localization. Springer International Publishing, Cham, pp 82–98
- Li X, Zhu S, d'Amorim M, Orso A (2018) Enlightened debugging. In: Proceedings of the 40th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2018). ACM
- Lin Y, Sun J, Xue Y, Liu Y, Dong J (2017) Feedback-based debugging. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp 393–403
- Masri W, Assi RA (2014) Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Trans. Softw. Eng. Methodol. 23(1):8:1–8:28
- McDonald S, Edwards HM, Zhao T (2012) Exploring Think-Alouds in Usability Testing: An International Survey, vol 55
- Nørgaard M, Hornbaek K (2006) What Do Usability Evaluators Do in Practice?. An Explorative Study of Think-Aloud Testing. Tech. rep.
- Olmsted-Hawala EL, Murphy ED, Hawala S, Ashenfelter KT (2010) Think-Aloud Protocols: A Comparison of Three Think-Aloud Protocols for use in Testing Data-Dissemination Web Sites for Usability. <http://www.census.gov>
- Parmar P, Patel M (2016) Software fault localization: A survey. Intl. Journal of Computer Applications 154(9):6–13
- Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, pp 199–209
- Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B (2017) Evaluating and improving fault localization
- Prabhakararao S, Cook C, Ruthruff J, Creswick E, Main M, Durham M, Burnett M (2003) Strategies and behaviors of end-user programmers with interactive fault localization. In: Proceedings - 2003 IEEE Symposium on Human Centric Computing Languages and Environments, HCC 2003, pp 15–22. <https://ieeexplore.ieee.org/abstract/document/1260197/>
- Renieris M, Reiss SP (2003) Fault localization with nearest neighbor queries. In: Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE 2003). IEEE Computer Society, pp 30–39
- Ribóira A, Abreu R (2010) The GZoltar Project: A Graphical Debugger Interface. In: Testing: Academia-Industry Collaboration, Practice and Research Techniques. Springer, Berlin, Heidelberg, pp 215–218
- Robillard MP, Coelho W, Murphy GC (2004) How effective developers investigate source code: An exploratory study. IEEE Trans Softw Eng 30(12):889–903. <https://doi.org/10.1109/TSE.2004.101.https://ieeexplore.ieee.org/abstract/document/1377187/>

- Rojas JM, Fraser G, Arcuri A (2015) Automated unit test generation during software development: a controlled experiment and think-aloud observations. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM, New York, NY, USA, <https://doi.org/10.1145/2771783.2771801>
- rubberduck (2021) Rubber duck debugging. [https://en.wikipedia.org/wiki/Rubber\\_duck\\_debugging](https://en.wikipedia.org/wiki/Rubber_duck_debugging). Accessed 29 March 2021
- Schröer M, Koschke R (2021) Recording, visualising and understanding developer programming behaviour. In: Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'21)
- Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A (2015) Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, pp 201–211
- Shynkarenko V, Zhevaho O (2020) Development of a toolkit for analyzing software debugging processes using the constructive approach. Eastern-European Journal of Enterprise Technologies, (107) pp 29–38
- Sillito J, De Volder K, Fisher B, Murphy G (2005) Managing software change tasks: an exploratory study. pp 10
- Sillito J, Murphy GC, De Volder K (2006) Questions programmers ask during software evolution tasks. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp 23–34, <https://doi.org/10.1145/1181775.1181779>
- Silva J (2011) A survey on algorithmic debugging strategies. Adv. Eng. Softw. 42(11):976–991
- SoDA (2021) SoDA library. <https://github.com/sed-szeged/soda>. Accessed 13 August 2020
- Sohn J, Yoo S (2017) FLUCCS: Using code and change metrics to improve fault localization. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2017. ACM, pp 273–283
- Spinellis D (2016) Effective Debugging: 66 Specific Ways to Debug Software and Systems. Addison-Wesley Professional (1)
- Steimann F, Frenkel M, Abreu R (2013) 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, pp 314–324
- van Someren MW, Barnard YF, Sandberg JAC (1994) The think aloud method: A practical guide to modelling cognitive processes. Academic Press, London. [https://doi.org/10.1016/0306-4573\(95\)90031-4](https://doi.org/10.1016/0306-4573(95)90031-4)
- Vessey I (1986) Expertise in debugging computer programs: An analysis of the content of verbal protocols. IEEE Transactions on Systems, Man, and Cybernetics 16(5):621–637
- von Mayrhauser A, International AMVansProceedingssofth, 1994 U (1996) Comprehension processes during large scale maintenance. ieeexplore.ieee.org. <https://ieeexplore.ieee.org/abstract/document/296764/>
- Wallace C, Cook C, Summet J, Burnett M (2002) Assertions in End-User Software Engineering: A Think-Aloud Study. Tech. rep.
- Wong WE, Debroy V (2009) A survey of software fault localization. Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45, 9
- Wong WE, Debroy V, Gao R, Li Y (2014) The dstar method for effective software fault localization. IEEE Trans. Reliability 63:290–308
- Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Trans Softw Eng 42(8):707–740
- Xia X, Bao L, Lo D, Li S (2016) “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, pp 267–278
- Xie X, Chen TY, Kuo F-C, Xu B (2013) A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. 22(4):31:1–31:40
- Xu X, Debroy V, Wong WE, Guo D (2011) Ties within fault localization rankings: Exposing and addressing the problem. Int J Softw Eng Knowl Eng 21:803–827
- Yamashita A, Petrillo F, Khomh F, Guéhéneuc Y-G (2018) Developer interaction traces backed by ide screen recordings from think aloud sessions. In: Proceedings of the 15th International Conference on Mining Software Repositories. MSR '18. Association for Computing Machinery, New York, NY, USA, pp 50–53, <https://doi.org/10.1145/3196398.3196457>
- Yoo S, Harman M, Clark D (2013) Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. ACM Trans Softw Eng Methodol 22(3):1
- Yoo S, Xie X, Kuo F-C, Chen TY, Harman M (2017) Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. ACM Trans. Softw. Eng. Methodol. 26(1):4:1–4:30

- Yu Y, Jones JA, Harrold MJ (2008) An empirical study of the effects of test-suite reduction on fault localization. In: International Conference on Software Engineering (ICSE). ACM, pp 201–210
- Zeller A (2009) Why Programs Fail, Second Edition: A Guide to Systematic Debugging, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
- Zhang M, Li X, Zhang L, Khurshid S (2017) Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2017. ACM, New York, NY, USA, pp 261–272

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

## Affiliations

Ferenc Horváth<sup>1</sup>  · Árpád Beszédes<sup>1</sup> · Béla Vancsics<sup>1</sup> · Gergő Balogh<sup>1</sup> · László Vidács<sup>1,2</sup> · Tibor Gyimóthy<sup>1,2</sup>

Árpád Beszédes  
beszedes@inf.u-szeged.hu

Béla Vancsics  
vancsics@inf.u-szeged.hu

Gergő Balogh  
geryxyz@inf.u-szeged.hu

László Vidács  
lac@inf.u-szeged.hu

Tibor Gyimóthy  
gyimi@inf.u-szeged.hu

<sup>1</sup> Department of Software Engineering, University of Szeged, Szeged, Hungary

<sup>2</sup> MTA-SZTE Research Group on Artificial Intelligence, University of Szeged, Szeged, Hungary