Poster: Aiding Java Developers with Interactive Fault Localization in Eclipse IDE

Gergő Balogh, Ferenc Horváth, Árpád Beszédes
Department of Software Engineering, University of Szeged
Dugonics tér 13, H-6720 Szeged, Hungary
{geryxyz,hferenc,beszedes}@inf.u-szeged.hu

***Abstract*—Spectrum-Based ones are a popular class of Fault Localization (FL) methods among researchers due to their relative simplicity. However, recent studies highlighted some barriers to the wider adoption of the technique in practical settings. One possibility to increase the practical usefulness of related tools is to involve *interactivity* between the user and the core FL algorithm. In this setting, the developer interacts with the fault localization algorithm by giving feedback on the elements proposed by the algorithm. This way, the proposed elements can be influenced in the hope to reach the faulty element earlier (we call the proposed approach Interactive Fault Localization, or iFL). With this work, we present our recent achievements in this topic. In particular, we overview the basic approach, our preliminary experimentation with user simulation, and the supporting tool for the actual usage of the method, *iFL for Eclipse*. Our aim is to provide a basis for the investigation of the feasibility and effectiveness of the technique, before moving on to more comprehensive experiments with actual human subjects. We invite researchers for further discussion on the topic, and for that, the method and tool will be made accessible.**

I. Introduction

This work deals with *fault localization* (FL), a debugging subactivity in which the root causes of an observed failure are sought. In particular, we present a technique to aid Spectrum-Based Fault Localization (SBFL), a class of FL methods popular among researchers [1]. The benefit of SBFL is that it relies on two sets of information from test executions, which are typically readily available or easily obtainable in existing projects: detailed code coverage and test outcomes. Based on statistical information about the number of failing and passing test cases exercising different code elements of the system, elements are assigned various *suspiciousness scores* that can then be used to *rank* the code elements, thus aiding the developer in the debugging activity.

There are barriers to the wider adoption of SBFL in programming practice, such as a high number of elements to investigate [2], [3], and other issues [4], [5]. A possibility to increase the practical usefulness of SBFL tools is to involve *interactivity* and hence improve the tool's most crucial performance property, fault localization effectiveness.

In our approach, called Interactive Fault Localization (iFL), we involve the user's previous or acquired knowledge about the system. The developer interacts with the fault localization algorithm by giving feedback on the elements of the prioritized list. This way, the next proposed suspicious elements can be influenced in the hope to reach the faulty element earlier.

II. Interactive Fault Localization

A. Related Work

The developer typically has additional information about the system of which the SBFL engine is not aware. Other researchers have explored the benefits of such information. For example, Li *et al.* [6], [7] reuses the knowledge about passing parameter values in a debugging session, Hao *et al.* [8] asks for feedback about the execution trace, Gong *et al.* [9] asks only for a simple yes/no feedback for a given statement. Lei *et al.* [10] utilize test data generation techniques to produce feedback for interacting with fault localization techniques automatically. To our knowledge, however, contextual information about higher level entities (for instance, statement vs. enclosing function) has not yet been leveraged for interactive SBFL. The contextual knowledge of the user about the next item (e. g., a statement) is exploited in the ranked list, with which larger code entities (e. g., a whole function) can be repositioned in their suspiciousness.

B. Method

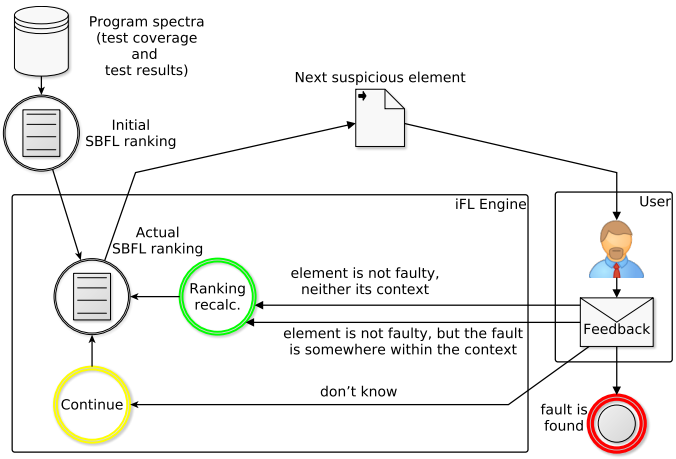

Fig. 1. Basic process of Interactive Fault Localization

Figure 1 shows a conceptual overview of our approach. The process starts by calculating an initial rank based on some traditional SBFL approach (we experimented with Tarantula [11], but any other base method could be used). 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, the iFL engine performs the following actions. In the case of (1), the process terminates, while at (4) it is continued as usual with the next suspicious element (this means that in the worst case when the developer has no background knowledge, the method falls back to the pure SBFL approach). In the remaining two cases, the iFL engine makes adjustments to the suspiciousness scores, recalculates the ranking and shows the next element from the new list to the user in the next iteration.

C. Experiments with Defect4J

As a proof-of-concept, we performed an initial set of experiments with the goal to have a preliminary view of how much improvement can we expect from iFL in terms of improvements in fault localization effectiveness. We implemented the approach from Figure 1 to handle Java systems using simulated users instead of real programmer feedback. The basic code entity in the FL process was a Java *method*, while its *class* was treated as the context. We implemented user simulation so that it automatically gives reliable answers to cases (2) and (3) based on the actually faulty element as follows: in case of (2), the whole context gets 0 score, while for (3) everything but the context is reduced to 0.

TABLE I
iFL IMPROVEMENTS ON DEFECTS4J

Program	Tarantula	iFL	Diff.	Impr.
commons-lang	3.81 (0.19%)	2.00 (0.09%)	-1.81 (-0.10%)	47.50%
commons-math	7.88 (0.17%)	7.21 (0.15%)	-0.67 (-0.02%)	8.50%
joda-time	17.56 (0.43%)	4.70 (0.12%)	-12.86 (-0.31%)	73.23%
Average	**9.75 (0.26%)**	**4.64 (0.12%)**	**-5.11 (-0.14%)**	**43.08%**

The experiment was performed on real defects from the Defect4J repository [12]. For the sake of simplicity, we considered only single method faults, and those faults where the suspiciousness score was not 0. Table I shows the improvements iFL achieved on Defects4J. The performance of the original SBFL algorithm can be seen in column 2 (Tarantula), which we used as the reference to evaluate iFL. We used the Expense measure for this purpose, which is essentially the number of elements that need to be investigated (using the middle position in the case of ties). We present results both in terms of absolute measure expressed in the number of code elements, and a relative version compared to the length of the rank list (shown in parentheses). A summarization row is provided as well with the corresponding average values.

The base SBFL method prioritized the faulty code elements roughly to the 10th place on average, which means that 10 executable code elements must be examined on average to find the faulty one. Column 3 contains the same data for iFL, which produces an Expense of 4.64 (0.12%) on average. This means that in this case a programmer would need only about 5 steps to find the fault on average, which is notably better than for the original algorithm. Column 4 shows the actual difference between the absolute and relative Expense measures. Column 5 of the table contains a summary of improvements in terms of relative changes in the Expense values, expressed in percentage (that is, the difference over the base SBFL value), which is 43.08% on average.

This initial set of experiments gave us confidence that if such notable improvements can be achieved (assuming perfect user responses), an investigation of the approach with real users is worthwhile. As a first step, we implemented a supporting tool in Eclipse as introduced below.

III. iFL – SUPPORTING TOOL FOR ECLIPSE

A. Motivation and Overview

Fault localization is a debugging activity which is, by definition, part of a programmer's work in which she has to interact with the source code of the software being debugged. It follows that this can be performed most effectively through the IDE itself; hence the most logical form of supporting tools is when they are integrated into the IDE.

With this in mind, we present *iFL for Eclipse*, which is an Eclipse plug-in for supporting iFL for Java projects developed in this environment. The plug-in reads the tree of project elements (classes and functions) and lists them in a table showing detailed information about those elements. This information includes, among others, the suspiciousness scores calculated using a traditional SBFL formula, such as Tarantula. This table also enables direct navigation towards the project tree and the contained source code elements.

Interactivity between the tool and the programmer is achieved by providing the capability to send feedback to the FL engine about the next element in the table based on user knowledge. It involves the *context* of the investigated element: in our case, Java classes and methods. This gives an opportunity to change the order of elements in the table and hopefully arrive at the faulty element more quickly.

We have determined two general requirements for our tool: 1) *iFL for Eclipse* should be extensible to make it possible to integrate various already existing SBFL algorithms and future iFL algorithm variants; 2) the usage of any iFL implementation should not generate unnecessary overhead by disturbing the typical workflow of the developer. In this section, we give a short description of the main features of *iFL for Eclipse* from the perspective of the end user and an overview of the main implementation-related details.

iFL for Eclipse is a plug-in supporting Java 10 and later, and Eclipse 2018-12 and later, so it is part of the well-known workspace of developers. It is published via an update site and can be installed using common Eclipse functionality. The current version of *iFL for Eclipse* is available at https://github.com/sed-szeged/iFL4Eclipse.

B. Interactive Fault Localization Session

The interaction with the *iFL for Eclipse* follows a session based process. The main steps of this process are shown in

Figure 2. There could be only one active session, which is tied to the selected (active and loaded) Java project.

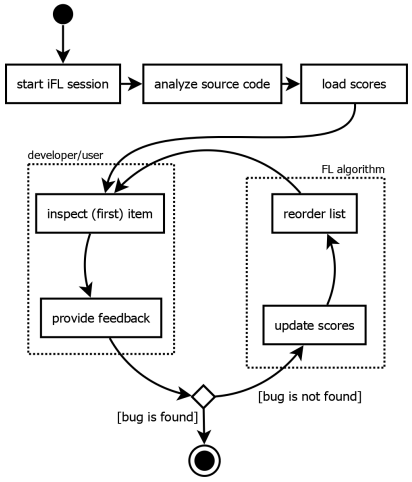

Fig. 2. Overview of an iFL Session

After the user initiates the session, *iFL for Eclipse* begins analyzing the project and listing the source code items currently present with their suspiciousness scores. The user can load scores produced by external tools.

The core cycle of the session is basically an interaction between the user and the iFL algorithm. The algorithm suggests the most likely location of the bug by providing a list of methods ordered by their suspiciousness scores. The user inspects the provided list and gives feedback based on her experiences by choosing one of the provided options. After that, *iFL for Eclipse* reorders the list and the cycle starts over again. The user can break the loop either by forcing the session to terminate or by choosing the option "item is faulty", which indicates that the developer successfully located the bug.

C. Score List

After the initial scan of the source code, *iFL for Eclipse* displays a list of methods currently present in the selected Java project (Figure 3). We use the JDT[1] library to retrieve the associated properties. All of these attributes are given by using the *de facto* standard notation for Java bytecode.

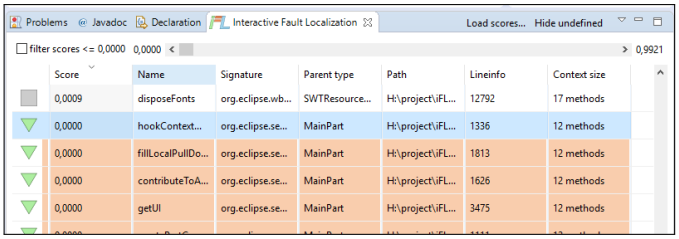

Fig. 3. Score List

There are altogether seven properties per method, and five of them are retrieved directly from the source code. These are their name, signature, enclosing type and the location of the containing file (given as full path and starting offset). The remaining attributes are computed by the iFL algorithm (the scores) or derived from the source code (size of the context, i. e. the count of methods in the enclosing type).

1) Load Scores from External Analysis: The user can load suspiciousness scores from an external source. The information should be in a comma separated data format with headers. The only required columns are `name` and `score`. These features make it possible to use any SBFL algorithm with *iFL for Eclipse*.

2) Navigation to Source Code Item: Currently, we support two easy ways to navigate to the inspected method. The user can either double click on an item or select several of them and then use the context menu option "Navigate to selected" (Figure 4). After that, Eclipse opens the source file with the default editor and navigates to the specific position.

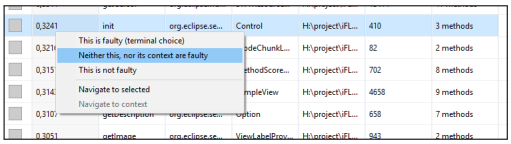

Fig. 4. Context Menu of Score List's Item

3) Acquiring User Feedback: The user can give various kinds of feedback about the inspected method by using the context menu of the list (Figure 4). The possible options depend on the underlying iFL algorithm. The currently implemented ones are: (1) the item is faulty (the process stops), (2) it is not faulty nor its context (they are moved lower in the rank) and (3) it is not faulty (score of the selected item is set to 0). The table also includes an iconographic depiction of the increase or decrease of scores as the result of the interaction with the user.

4) Hide Undefined Scores: There may be methods for which the underlying algorithm is unable to compute initial scores (because none of the test cases cover these or some other shortcomings of the SBFL algorithm). We denote these with "undefined" score value in the list (Figure 3). Because iFL algorithm cannot update these scores, *iFL for Eclipse* provides a feature to exclude these from the list.

D. Integration of New iFL Algorithm Variants

Different variants of the iFL algorithm could be used to leverage the developer's knowledge. When developing *iFL for Eclipse*, we implemented one of these, but the underlying architecture enables easy integration of other variants as well.

The architecture of *iFL for Eclipse* consists of several layers. The main UI of the tool is an Eclipse part, a graphical panel, serving as the front end. It is connected to the back end components, whose purpose is the update of scores and the recalculation of the rank list based on user input. In Figure 5, the layers related to Eclipse infrastructure are located at the bottom (marked with purple), while the layers of the plug-in are placed over these (marked with green). The Standard

[1]https://www.eclipse.org/jdt/

Widget Toolkit (SWT[2]) and Eclipse Java development tools (JDT) provide the connection between them.

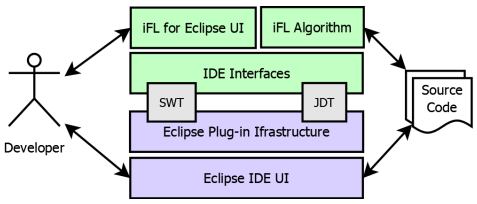

Fig. 5. Overview of the *iFL for Eclipse* Architecture

We separated the usage of the Eclipse Plug-in infrastructure from the rest of the *iFL for Eclipse* by creating an IDE Interfaces layer. Both the UI, whose main task is to provide communication with the user, and the iFL algorithms rely on the functionality of this layer.

This general structure makes it possible to easily replace the underlying iFL algorithm. Currently, new implementations have to be included into the *iFL for Eclipse* code base, but we plan to extract this by using extension points so that other plug-ins could more easily provide their variants of iFL algorithms.

E. Present State

At the present state of our research agenda, the demonstrated tool serves our research purposes: to investigate the feasibility and effectiveness of iFL. As such, it is in a prototype state, not thoroughly tested and validated. The tool will be made open source, along with the results of associated experiments, to enable other researchers its independent validation and further development. Regarding functionality, it currently includes the basic features, and there are many possibilities for further development. Our primary plans are to increase usability and flexibility concerning user feedback actions and the underlying FL computation. Ongoing research is to perform user studies to investigate the effectiveness of the proposed iFL approach and the tool itself.

IV. Conclusions

We present an Interactive Fault Localization approach whose aim is to increase the practical usefulness of Spectrum-Based Fault Localization by introducing the programmer to the FL process in a feedback loop. The programmer's background knowledge about the code elements in the rank list provided by the base FL algorithm is reused for adjusting the ranking scores in the next iteration. We present the results of our initial experiments on a large Java benchmark using simulated users, and also a supporting tool integrated into the Eclipse IDE, which enables the application of the method with real users.

A. Call for Action

At the present state of our research, we have laid down the foundations for experimentation with iFL. However, for that several aspects need to be addressed. The most obvious one, and maybe the most difficult as well, is to choose the correct variation of the underlying algorithm. For example, what are the most meaningful options and actions for the user response? How to define the granularity of the analysis? What constitutes the best context of a source code item?, and so on.

Furthermore, there are several challenges that need to be addressed by iFL related to the everyday workflow of developers while debugging. For example, they usually wander off from the inspected item and evaluate several code constructs simultaneously. Next, the source code may be changed during the fault localization session to try out different ideas to better understand the causes of the error, etc.

A tool that aims to successfully aid programmers in their debugging activity, including *iFL for Eclipse*, must face these challenges. One of the goals of this paper is, admittedly, to engage other researchers in further discussion on this topic.

[2]https://www.eclipse.org/swt/

Acknowledgments

Árpád Beszédes was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Ministry of Human Capacities, Hungary grant 20391-3/2018/FEKUSTRAT is acknowledged. We thank Rita Bártfai for her help with the poster design.

References

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

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

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

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

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

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

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

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

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

[10] Y. LEI, X. MAO, Z. DAI, and D. WEI, "Effective fault localization approach using feedback," *IEICE Transactions on Information and Systems*, vol. E95.D, no. 9, pp. 2247–2257, 2012.

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

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