Interactive Fault Localization for Python with CharmFL

Attila Szatmári
szatma@inf.u-szeged.hu
Department of Software Engineering,
University of Szeged
Szeged, Hungary

Qusay Idrees Sarhan
sarhan@inf.u-szeged.hu
Department of Software Engineering,
University of Szeged
Szeged, Hungary
Department of Computer Science,
University of Duhok
Duhok, Iraq

Árpád Beszédes
beszedes@inf.u-szeged.hu
Department of Software Engineering,
University of Szeged
Szeged, Hungary

ABSTRACT

We present a plug-in called "CharmFL" for the PyCharm IDE. It employs Spectrum-based Fault Localization to automatically analyze Python programs and produces a ranked list of potentially faulty program elements (i.e., statements, functions, etc.). Our tool offers advanced features, e.g., it enables the users to give their feedback on the suspicious elements to help re-rank them, thus improving the fault localization process. The tool utilizes contextual information about program elements complementary to the spectrum data. The users can explore function call graphs during a failed test. Thus they can investigate the data flow traces of any failed test case or construct a causal inference model for the location of the fault. The tool has been used with a set of experimental use cases.

CCS CONCEPTS

• **Software and its engineering** → Dynamic analysis; *Software maintenance tools*; *Integrated and visual development environments*; **Software testing and debugging**; • **Human-centered computing** → Interactive systems and tools.

KEYWORDS

Debugging, spectrum-based fault localization, Interactive Fault Localization, CharmFL, Python, PyCharm

ACM Reference Format:
Attila Szatmári, Qusay Idrees Sarhan, and Árpád Beszédes. 2022. Interactive Fault Localization for Python with CharmFL. In *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST '22), November 17–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software systems and applications cover many aspects of our day-to-day activities. However, they are still far from being free of faults. Software faults may cause critical undesired situations, including life loss. Therefore, various software fault localization techniques have been proposed over the last few decades, including Spectrum-based fault localization (SBFL) [6]. In SBFL, the probability of each program element (e.g., statements) being faulty is calculated based on program spectra obtained by executing test cases. However, SBFL is not yet widely used in the industry because it poses a number of issues [4]. One such issues is that most SBFL tools currently target programs written in C/C++ and Java. Thus, there is a lack of SBFL tools that help developers debug their programs that are written in other programming languages, including Python, which is also considered to be one of the most popular programming languages.

In our previous paper [9], we presented a framework called "CharmFL" to support the fault localization process in the PyCharm IDE. However, it does not employ interactive fault localization for its users.

SBFL approaches usually compute program elements' suspiciousness scores without consulting the user, which is considered one of the main issues that decrease its applicability [14]. As a result, the user's prior knowledge of the subject program is not used to increase fault localization accuracy. By including the users and taking into account their feedback on the suspicious elements or an element's context and ranks, the fault localization process can be improved. We introduce "close" and "far" contexts for each program statement. "Close" context refers to the method that contains the statement and "far" context refers to the methods that call or have been called by the method that contains the investigated statement. In this paper, we introduce "CharmFL" with an interactivity feature. Another feature it provides is the display of a static call graph, which adds context that helps users to provide their feedback on the suspiciousness of code elements, and in particular, changes the scores based on the calling distance from the investigated element.

2 SPECTRUM BASED FAULT LOCALIZATION (SBFL)

Fault localization is a time-consuming part of the software debugging process, therefore, the need for automation is very important. There are several approaches implementing the process[15]. Using the program's spectra (i.e., program elements, per-test coverage, and test results), SBFL can help programmers find the faulty element in the target program's code easier. The code coverage matrix is a two-dimensional matrix used to represent the relationship between the test cases and the program elements. Its rows demonstrate the test cases and its columns represent the program elements. An element of the matrix is 1 if it is covered by the test case, otherwise it is 0. In another matrix vector, the test results are stored, where

A-TEST '22, November 17–18, 2022, Singapore, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

0 means the test case passed and 1 means it failed. Using these matrices, the following four basic statistical numbers are calculated for each program element e:

- ef: number of failed tests covering e
- ep: number of passed tests covering e
- nf: number of failed tests not covering e
- np: number of passed tests not covering e

Then, our tool uses these four numbers with an SBFL formula such as Tarantula [11] or Ochiai [1], etc. to provide a ranked list of program elements. Whichever element ranked the highest in the list, is the most suspicious of containing a bug.

3 RELATED WORKS

"Whyline", a standalone debugging tool for Java programs, was proposed by Ko and Myers [12]. The tool uses static and dynamic slicing to generate why and why not questions, which are then displayed in a graphical and interactive style. Each question helps the user to have some useful information about a selected program element and also the tool collects the user's answer to each question to include or exclude an element from the displayed list.

Horvath et al. [8] proposed an SBFL tool called "iFL" for Java developers using Eclipse. The tool provides a ranked list to the user, and while debugging they flag elements that are buggy or not. However, choosing the latter they have two options; either they say the element's context is suspicious or not. Given this, the tool will recalculate the ranks with the additional information.

In [5, 7], the authors also presented an interactive fault localization approach that relies on straightforward user feedback. The user can interact with their approach by deciding whether or not a recommended suspicious element is valid. Following that, the proposed approach takes the simple user feedback and re-orders the rest of the suspect program elements based on it, with the goal of putting actually faulty elements at the top of the ranking list.

In [13], the authors proposed an approach called "Enlighten". It uses dynamic program slicing to form a Dynamic Dependence Graph (DDG) for every failed test in the test suite. This information will then be used to create queries. Each query consists of a method invocation with its input and output values, which the user can mark as correct or incorrect. This approach in each iteration updates the debugging data and the ranking list based on the user feedback until the fault is found.

In [2], the authors suggested an interactive method for estimating the number of coincidentally correct test cases (those that execute faulty statements but do not cause failures) based on user comments about the correctness of a set of statements. Thus, helping them exclude such test cases and improve the fault localization process.

Janssen et al. [10] and Campos et al. [3] proposed a fault localization tool that adopts SBFL and it is available as a command-line tool called "Zoltar" and as an Eclipse plug-in called "Gzoltar". It also uses colors to mark the execution of program entities from red to green based on their suspiciousness scores.

Compared to the tools mentioned above, our tool offers many unique features such as: (a) it targets Python programs. (b) it supports interactivity from a new context perspective, i.e. via "close" and "far" contexts. (c) it supports different types of coverage spectra. (d) it can be used as a command-line tool or as a plug-in tool. (e) it supports a hierarchical navigation of program elements.

4 INTERACTIVITY IN SBFL

Our tool implements interactivity similar to iFL4Eclipse [8], the fault localization tool for Java programs. Via interactivity, the developer can give feedback on the elements to the tool, and it will help the developer by recalculating the elements' suspicious scores.

Horváth et. al. [8] defined the developer feedback as: either the investigated element is definitely not buggy but its context may be, or the element and its context do not contain the bug. Otherwise, the user finds the bug. Based on their findings, including the elements' context can help the developer, however, context may vary for different program structures. This may be true for Python as well. In our tool, we extend the context by investigating the static call graph and update the element's callers' and callees' scores by a certain amount. (i.e. "recalculation factor")

Our tool provides the possibility to generate a static call graph for the investigated Python program. Whenever the user selects a program element its callers and callees will be highlighted.

This is useful for developers, since they can visualize the context, thus it helps to efficiently find the buggy element. When developers are debugging, they investigate the "close" and "far" contexts as well. One context might seem buggier than the other, i.e. we know the bug is not in the method but the callee seems faulty or the other way around. Table 1 presents an example code with a seeded fault. It has two methods and an exception handler class. The test cases check if the returned text equals "You guessed X right!", where X is a provided integer. We can see the iterations of how the user's feedback alternates the scores (we used Tarantula for demonstration). The first iteration represents the basic suspiciousness scores for each element. We set the **recalculation factor** for "close" context to 1.5 and "far" context to a lower number, 1.2. The user of the tool is provided with the option to modify these values as needed.

Following the example output from Table 1, the developer investigates the first element with the highest score (i.e., the 10th statement). During debugging they are not sure about the "close" context, but they can tell that the statement and the called Error class are not faulty so they set the statement and its "far" context to non-buggy. This results in multiplying the scores of the close context elements. They investigate the next element with the highest score and its context. The "close" context contains an expression assignment, if-else, and print statements. They conclude that the statement is not buggy, neither is its "close" context. In the next iteration, the 5th statement is the only element left with a suspiciousness score. They investigate it and find the bug.

In this simple example, they were able to find the bug only in 3 iterations, instead of following the whole original list of suspicious elements, which would have taken at least 7 iterations. The more complex the program, the more iterations the interactive approach can save in finding the bug. The difference highly depends on the developer's expertise. However, in the worst-case scenario, they use "CharmFL" as a basic SBFL tool, i.e. following the provided ranked list while debugging.

Table 1: Example code and fault localization process with seeded fault

	Source code	Test cases					Scores		
Line	Code	tc1	tc2	tc3	tc4	tc5	0. iteration	1. iteration	2. iteration
1	class Error(Exception):						-	-	-
2	pass	•	•			•	0.5	0	0
3							-	-	-
4	def congratulations_for(i_num):						-	-	-
5	return "".join("You guessed",i_num,"right!") # error; should be separated by white spaces			•	•		0.37	0.45	**0.54**
6							-	-	-
7	def guess_the_num(i_num):						-	-	-
8	number = 10	•	•	•			0.5	0.75	0
9	if i_num <number:	•	•	•			0.5	0.75	0
10	raise Error	•					**1**	0	0
11	elif i_num >number:		•	•			0,37	0.56	0
12	raise Error		•				**1**	**1.5**	0
13							-	-	-
14	return congratulations_for(i_num)			•			0	0	0
	Pass/Fail status	F	F	P	F	P			

Test cases: tc1 = guess_the_num(9); tc2 = guess_the_num(11); tc3 = guess_the_num(10); tc4 = congratulations_for(10); tc5 = Error();

5 TOOL'S OVERVIEW

5.1 Architecture

Figure 1 shows the architecture of our tool. The CharmFL engine runs tests, collects coverage, creates static call graphs, and does fault localization. To collect coverage, the engine uses one of the most popular coverage frameworks for Python called coverage.py [1]. The list of options is:

```
python main.py -fl -d <projects_directory>-alg tarantula -r <rank_mode>   To start the basic fault localization
python main.py -cg -d <projects_directory>                                 To generate static call graph
python main.py -c <filename>                                               To get class coverage
python main.py -m <filename>                                               To get method coverage
python main.py -s <filename>                                               To get spectrum
```

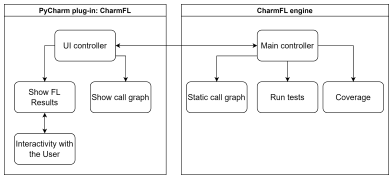

Figure 1: CharmFL architecture

The PyCharm plug-in gets the data from the engine and displays the outcome of fault localization and the call graph. The users can interact with this part, they can modify the list according to the methodology presented in Section (4). They can view the corresponding context of the element. After the user gives their insight, the UI will rearrange the list accordingly and let the user continue debugging.

5.2 Graphical User Interface (GUI)

The tool's user interface is an IDE-specific plug-in for PyCharm. After installing the plug-in, i.e. drag and drop the zip file into the IDE, the user can either run fault localization or generate the static call graph.

The hierarchical ranked list (tree) of suspicious elements is provided in the results table (Figure 2). The Action button can be used to hide/show the elements inside each level of the hierarchy or to jump to a specific element via clicking on the element. Even though this is a compact view, developers find the separated lists useful. Thus, we made three new table views, that show the ranked list of suspicious classes, methods, and statements separately (Figure 3).

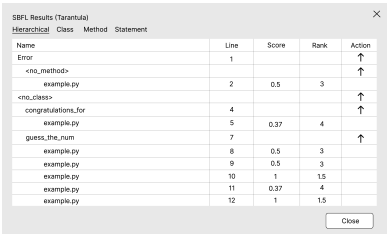

Figure 2: CharmFL ranking list output

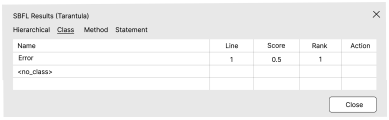

Figure 3: CharmFL ranked list of suspicious classes

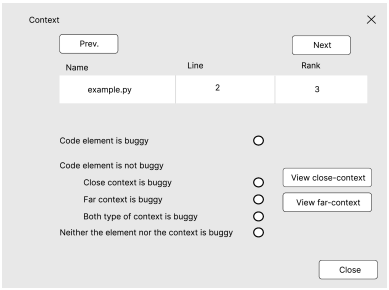

Figure 4: CharmFL interactivity window

Right-clicking any element will result in a pop-up window (Figure 4), where the user can interact with the fault localization algorithm. The user has several options to choose from:

[1] https://coverage.readthedocs.io/en/6.4.1/

- the element is buggy
- only the "close" context seems buggy
- only the "far" context seems buggy
- all contexts seem buggy
- neither the element nor its context seem buggy

The user can view both "close" and "far" contexts. Clicking the "View close context" button, the PyCharm IDE navigates the user to the code element. Clicking the "View far context", the PyCharm IDE opens the static call graph highlighting the callers and callees of the element as shown in Figure 5.

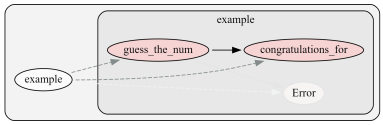

Figure 5: Static call graph.

6 APPLICABILITY SCENARIOS

When developers notice bugs in their Python programs, they have several options while debugging. In order to find the bug, they can run test cases to see which ones are failing or they can start understanding and debugging the code. Our tool can be used in either situation to locate faults using test results, coverage, and contextual information. In this section, we will give two scenarios that show how our tool can be used.

1) The developers can use the tool as a basic SBFL tool. The users start the fault localization process and the tool provides the hierarchical tree of suspicious elements. Checking the first element with the highest rank, they click on the element in the tree and the tool navigates them to the element. They investigate the code element and decide whether it caused the fault or not. If it did, the task terminates. However, if the element did not cause the fault they can move on to the next element in the tree based on the ranks. The user goes through the elements until they find the one that is causing the fault.

2) The developer can interact with the algorithm. The same steps apply as in the first scenario until the user is sure the investigated code element is not buggy. They have a few options at this point. The developer knows the element, e.g. statement, is not causing the fault, however its context might, e.g. method, caller, callee, etc. They are prompted with the opportunity to view these contexts and decide on whether they are suspicious. Let us say the investigated code element was in a "getter" method. The developer knows this one is not faulty but the method that called it could be. In this case they can say the far context is suspicious and the program will recalculate the list accordingly. After this point, the developer repeats this scenario until they find the buggy element.

7 CONCLUSION

The present state of the tool is a research prototype, which we are using in our SBFL-related research activities in our department. It is also a framework in which various experimental ideas (including student works) are tried. We provide the tool [2] as open-source for the research community to be able to conduct related experiments.

[2] https://interactivefaultlocalization.github.io/

For future work, we would like to add different types of interactivity to enable the user to give their feedback on the suspicious elements to help re-rank them, thus improving the fault localization process. Assessing the tool with real users and in real-world scenarios would be a valuable next step as well. Our ultimate goal is to improve the usability of implemented features and test on various subjects and scenarios to attain a state in which Python developers can be effectively supported in their debugging tasks.

ACKNOWLEDGEMENTS

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.

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*. 39–46. https://doi.org/10.1109/PRDC.2006.18

[2] Aritra Bandyopadhyay and Sudipto Ghosh. 2012. Tester Feedback Driven Fault Localization. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 41–50. https://doi.org/10.1109/ICST.2012.84

[3] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: An eclipse plug-in for testing and debugging. *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings* (2012), 378–381.

[4] Higor A. de Souza, Marcos L. Chaim, and Fabio Kon. 2016. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. (jul 2016), 1–46. arXiv:1607.04347 http://arxiv.org/abs/1607.04347

[5] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. 2012. Interactive fault localization leveraging simple user feedback. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 67–76. https://doi.org/10.1109/ICSM.2012.6405255

[6] C. Gouveia, J. Campos, and R. Abreu. 2013. Using HTML5 visualizations in software fault localization. In *First IEEE Working Conference on Software Visualization (VISSOFT)*. 1–10. https://doi.org/10.1109/VISSOFT.2013.6650539

[7] Dan Hao, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. 2009. Interactive Fault Localization Using Test Information. *Journal of Computer Science and Technology* 24, 5 (2009), 962–974. https://doi.org/10.1007/s11390-009-9270-z

[8] F Horváth, V S Lacerda, Á Beszédes, L Vidács, and T Gyimóthy. 2019. A New Interactive Fault Localization Method with Context Aware User Feedback. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. 23–28. https://doi.org/10.1109/IBF.2019.8665415

[9] Qusay Idrees Sarhan, Attila Szatmari, Rajmond Toth, and Arpad Beszedes. 2021. CharmFL: A Fault Localization Tool for Python. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 114–119. https://doi.org/10.1109/SCAM52516.2021.00022

[10] Tom Janssen, Rui Abreu, and Arjan J.C. Van Gemund. 2009. Zoltar: A spectrum-based fault localization tool. *SINTER'09 - Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution at Runtime* (2009), 23–29.

[11] J. A. Jones, M. J. Harrold, and J. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. 467–477. https://doi.org/10.1145/581396.581397

[12] Andrew J. Ko and Brad A. Myers. 2008. Debugging reinvented: Asking and answering why and why not questions about program behavior. *Proceedings - International Conference on Software Engineering* (2008), 301–310.

[13] Xiangyu Li, Shaowei Zhu, Marcelo D'Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, New York, NY, USA, 82–92. https://doi.org/10.1145/3180155.3180242

[14] Qusay Idrees Sarhan and Arpad Beszedes. 2022. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access* 10 (2022), 10618–10639. https://doi.org/10.1109/ACCESS.2022.3144079

[15] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (aug 2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368