CharmFL: A Fault Localization Tool for Python

Qusay Idrees Sarhan[1,2], Attila Szatmári[1], Rajmond Tóth[1], Árpád Beszédes[1]
[1] Department of Software Engineering, University of Szeged, Szeged, Hungary
[2] Department of Computer Science, University of Duhok, Duhok, Iraq
{sarhan, szatma, beszedes}@inf.u-szeged.hu
raymonddrakon@gmail.com

Abstract—**Fault localization is one of the most time-consuming and error-prone parts of software debugging. There are several tools for helping developers in the fault localization process, however, they mostly target programs written in Java and C/C++ programming languages. While these tools are splendid on their own, we must not look over the fact that Python is a popular programming language, and still there are a lack of easy-to-use and handy fault localization tools for Python developers. In this paper, we present a tool called "CharmFL" for software fault localization as a plug-in for PyCharm IDE. The tool employs Spectrum-based fault localization (SBFL) to help Python developers automatically analyze their programs and generate useful data at run-time to be used, then to produce a ranked list of potentially faulty program elements (i.e., statements, functions, and classes). Thus, our proposed tool supports different code coverage types with the possibility to investigate these types in a hierarchical approach. The applicability of our tool has been presented by using a set of experimental use cases. The results show that our tool could help developers to efficiently find the locations of different types of faults in their programs.**

Index Terms—**Debugging, fault localization, spectrum-based fault localization, Python, CharmFL.**

I. 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) [1]. In SBFL, the probability of each program element (e.g., statements) of being faulty is calculated based on program spectra obtained by executing a number of test cases. However, SBFL is not yet widely used in the industry because it poses a number of issues [2]. One of such issues is that most of the SBFL tools currently target programs written in C/C++ and Java. Thus, there is lack in SBFL tools that help developers debug their programs that are written in other programming languages including Python which is considered also as one of the most popular programming languages.

In this paper, we present a tool called "CharmFL" as a plug-in for the PyCharm IDE, a popular Python development platform, to automate the software fault localization process. Our tool utilizes SBFL to assist Python developers in automatically analyzing their programs and producing useful data at run-time that can then be used to generate a ranked list of potentially faulty program elements. To determine whether a statement is faulty or not, developers examine each statement in turn, beginning at the top of the list (the most suspicious element). Several experiments with Python projects were conducted to assess the applicability of our tool. The results indicate that the tool is useful for locating faults in various types of programs and that it is simple to use.

The remainder of the paper is organized as follows. Section II briefly introduces the background of SBFL and its main concepts. Section III presents an overview of the most related works. Section IV provides a theoretical overview on the used techniques in our tool. Section V presents our proposed software fault localization tool. Section VI discusses the applicability of our tool in different practical contexts. Finally, we provide our conclusions and possible future works in Section VII.

II. Background of SBFL

Fault localization is a time consuming part of the software debugging process, therefore the need for automating it is incredibly important. There are several approaches to implement the process [3], however we focus on SBFL due to its simple but powerful nature, i.e. using only code coverage and test results. There have been several surveys written [3]–[5] and various empirical studies [6], [7] performed on this topic.

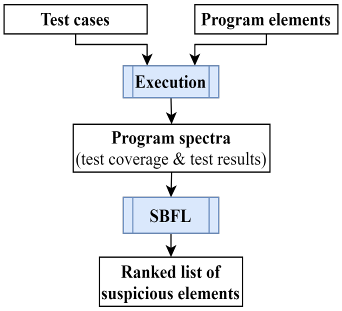


Fig. 1. SBFL process

Figure 1 shows the SBFL process. Using the program's spectra (i.e., program elements, per-test coverage, and test results), SBFL can help the programmer to 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, whose rows demonstrate the test cases and columns represent the program elements. An element of the matrix is 1, if it is covered by test case, otherwise it is 0. In another matrix vector, the test results are stored where 0 means the test case is passed and 1 when it is failed. Using these matrices, the following four basic statistical numbers are calculated for each program element ϕ:

- ϕ_{ep}: number of passed tests covering ϕ
- ϕ_{ef}: number of failed tests covering ϕ
- ϕ_{np}: number of passed tests not covering ϕ
- ϕ_{nf}: number of failed tests not covering ϕ

Then, our tool uses these four numbers with the formula in Equation 1, Tarantula [8]; Equation 2, Ochiai [9]; Equation 3, DStar [10]; Equation 4, Wong2 [11] to provide a ranked list of program elements as an output. Whichever element ranked the highest in the list, it is the most suspicious of containing a bug.

$$\text{Tarantula} = \frac{\frac{\phi_{ef}}{\phi_{ef}+\phi_{nf}}}{\frac{\phi_{ef}}{\phi_{ef}+\phi_{nf}} + \frac{\phi_{ep}}{\phi_{ep}+\phi_{np}}} \tag{1}$$

$$\text{Ochiai} = \frac{\phi_{ef}}{\sqrt{(\phi_{ef} + \phi_{nf}) * (\phi_{ef} + \phi_{ep})}} \tag{2}$$

$$\text{Dstar} = \frac{{\phi_{ef}}^2}{\phi_{ep} + \phi_{nf}} \tag{3}$$

$$\text{Wong2} = \phi_{ef} - \phi_{ep} \tag{4}$$

III. Related Works

There are many software fault localization tools implemented and proposed in the literature. This section briefly presents them. Jones et al. [8] proposed a standalone software fault localization tool called “Tarantula” to help C programmers to debug their programs. The tool assigns different colors to program statements based on how suspicious they are, ranging from red (most suspicious) to green (not suspicious). Besides, the tool displays varying brightness levels based on how frequently the tests execute a statement. The brightest statements are those that are most commonly executed. However, the tool does not run test cases and record their results; it takes as input a program’s source code and the results of executing a test suite on the program. Furthermore, the tool’s only supported metric is the Tarantula metric.

Chesley et al. [12] proposed an Eclipse plug-in tool called “Crisp” that helps developers identify the reasons for a failure that occurs due to code edits by constructing intermediate versions of a program that is being edited. For example, if a test case fails, the tool will identify parts of the program that have been changed and caused the failing test. Thus, developers can concentrate only on those affecting changes that were applied.

Ko and Myers [13] proposed a standalone debugging tool called “Whyline” for Java programs. The tool employs both static and dynamic slicing to formulate why and why not questions, which are then presented in a graphical and interactive way to help developers in understanding the behavior of a program under test. It also records program execution traces and the status of each used class whether it is executed or not. Using the tool also allows the user to load the execution trace of a program and select a program entity at a specific point during its execution. Then he or she can click on the selected entity to bring up a pop-up window containing a set of questions that include data values gathered during the execution as well as information about the properties of the selected entity.

Hao et al. [14] proposed an Eclipse plug-in tool called “VIDA” for programs written in Java. The tool extracts statements hit spectrum from the target programs, executes JUnit tests and based on their results, it calculates suspiciousness. It also provides a list of the ten most suspicious statements as potential breakpoints. It displays the history of breakpoints including the developers’ previous estimates of the correctness of the breakpoint candidates as well as their current suspiciousness. Moreover, it employs colors to distinguish between the developers’ estimations, ranging from red (wrong) to green (correct), and suspiciousness, ranging from black (very suspicious) to light gray (less suspicious). And, it provides the users with the ability to extract static dependency graphs from their programs to assist developers with their estimations and also to help them understand the relationships among different program entities.

Janssen et al. [15] and Campos et al. [16] 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”. The tool provides a complete infrastructure to automatically instrument the source code of the programs under test in order to generate runtime data, which is then used to return a ranked list of faulty locations. It also uses colors to mark the execution of program entities from red to green based on their suspiciousness scores. The tool only employs the Ochiai metric to compute suspiciousness.

Wang et al. [17] proposed a fault localization tool called “FLAVS” for developers using Microsoft Visual Studio platform. The tool provides an automatic instrumentation mechanism to record program spectrum information during the execution. It also provides a user with two options either automatically or manually to mark the result of each used test case; whether it is successful or not. Additionally, it monitors each test environmental factors of the running program such memory consumption, CPU usage, and thread numbers. For example, the developer can notice that there is something wrong when the CPU time drops to zero and never gets increased again during the running of a test case. The tool provides different levels of granularities for fault localization analysis such as statement, predicate, and function. Using the tool allows the users to examine the correct positions in the source code files by clicking on the suspicious units, which are displayed and highlighted in different colors also. The functionalities of “FLAVS” have been extended by Chen and

Wang [18] in another tool called “UnitFL”. The tool uses program slicing to decrease the program execution time. Besides, it provides different levels of granularities for fault localization analysis to provide different aspects of execution during the program analysis. And, it shows fault-related elements with different colors based on their suspiciousness; ranging from green to red.

Ribeiro et al. [19] proposed a SBFL tool called “Jaguar” for Java developers. The tool supports two advanced spectra types which are control-flow and data-flow. Also, it visualizes suspicious program elements where the user can easily inspect suspicious methods, statements, or variables. Although the data-flow spectrum provides more information, it is not adopted widely in SBFL because of the high costs of execution. To overcome this issue, the tool utilizes a lightweight data-flow spectrum coverage tool called “ba-dua”. This enables the tool to be used for testing large-scale programs at affordable execution costs. The tool can be used as an Eclipse plug-in or as a command-line tool.

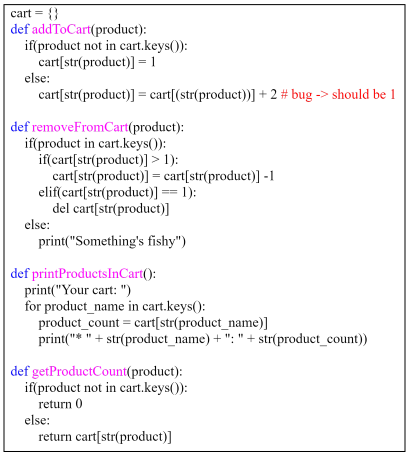
All the previous tools target programs written in Java and C/C++ programming languages. Tools for helping Python developers in their debugging process have not been previously proposed in the literature by other researchers. However, two open-source fault localization tools for Python’s pytest testing framework are available, namely, Fault-Localization [20] and PinPoint [21]. In this paper, we propose a tool called “CharmFL” with more features to target programs written in Python; which is considered one of the most popular programming languages nowadays. Compared to the other two tools, our proposed tool supports different types of code coverage (i.e., class, method, and statement), displays the fault localization results in different ways, provides a graphical user-friendly interface to examine the suspicious elements, and enabling the user to smoothly examine any suspicious element via clickable links to the source code. Table I summarizes the features of our proposed tool compared to the others.

TABLE I
COMPARISON AMONG PYTHON FAULT LOCALIZATION TOOLS

Features	Fault-Localization	PinPoint	CharmFL
Statement hit coverage	Yes	Yes	Yes
Method hit coverage	No	No	Yes
Class hit coverage	No	No	Yes
Supported SBFL metrics	1	5	4
Shows ranking	Color-based	Value-based list	Both
Shows suspicious scores	Yes	No	Yes
Ties ranking	No	No	Min, Max, or Average
GUI interface	No	No	Yes
Command-line interface	Yes	Yes	Yes
Elements investigation	Flat	Flat	Hierarchy
Elements navigation	No	No	Via clickable links to each element in the source code
Tool type	Option for pytest framework	Option for pytest framework	Plug-in for PyCharm IDE
Current version	0.1.6	0.3.0	0.1

IV. METHODOLOGY

In this section, we give a theoretical overview on the used techniques in our tool. We will use an example project for demonstration purposes. The selected project has four methods as shown in Figure 2 and four test cases to test them as shown in Figure 3. For simplicity, we will represent the four test cases through the paper as T1, T2, T3, and T4 according to their order in the figure.



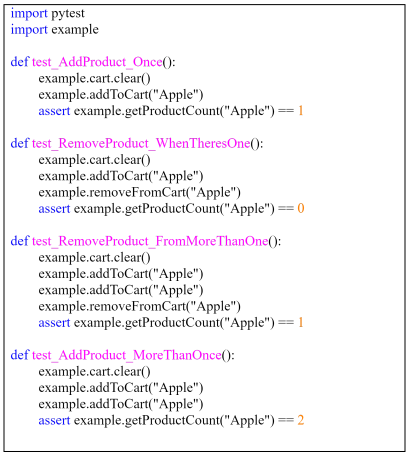
```
cart = {}
def addToCart(product):
  if(product not in cart.keys()):
    cart[str(product)] = 1
  else:
    cart[str(product)] = cart[(str(product))] + 2 # bug -> should be 1

def removeFromCart(product):
  if(product in cart.keys()):
    if(cart[str(product)] > 1):
      cart[str(product)] = cart[str(product)] -1
    elif(cart[str(product)] == 1):
      del cart[str(product)]
  else:
    print("Something's fishy")

def printProductsInCart():
  print("Your cart: ")
  for product_name in cart.keys():
    product_count = cart[str(product_name)]
    print("* " + str(product_name) + ": " + str(product_count))

def getProductCount(product):
  if(product not in cart.keys()):
    return 0
  else:
    return cart[str(product)]
```

Fig. 2. Running example – program code



```
import pytest
import example

def test_AddProduct_Once():
    example.cart.clear()
    example.addToCart("Apple")
    assert example.getProductCount("Apple") == 1

def test_RemoveProduct_WhenTheresOne():
    example.cart.clear()
    example.addToCart("Apple")
    example.removeFromCart("Apple")
    assert example.getProductCount("Apple") == 0

def test_RemoveProduct_FromMoreThanOne():
    example.cart.clear()
    example.addToCart("Apple")
    example.addToCart("Apple")
    example.removeFromCart("Apple")
    assert example.getProductCount("Apple") == 1

def test_AddProduct_MoreThanOnce():
    example.cart.clear()
    example.addToCart("Apple")
    example.addToCart("Apple")
    assert example.getProductCount("Apple") == 2
```

Fig. 3. Running example – test cases

Our tool provides the opportunity to measure statement, method, and class coverage levels. This is achieved by employing the “zooming in/out” hierarchy approach, where the user can examine the suspicious elements from the highest level in the hierarchy (i.e., classes) to lower levels in the hierarchy and repeat the steps above, until s/he reaches the lowest level, which is the statements level. This is better than only one level of granularity as the developer can exclude methods or even classes from the ranking list, thus saving time spent on the debugging process.

We can see in this example that the highest granularity is method level; in this case, the class level coverage is absent. Table II presents the method level coverage matrix, and the basic statistical numbers. Running any SBFL algorithm, e.g. Tarantula, we get a list of elements with suspiciousness scores as presented in Table III. For the lack of space, we will not show the statement granularity, but the overarching principle is the same; we investigate the elements with highest scores until we find the bug. We can see that the “addToCart” method has the highest score according to Tarantula. Using the “zooming in/out” technique, we need to investigate the statements in the “addToCart” method first. This saves the developers time since they do not have to go through all the statements in the suspicousness list.

TABLE II
METHOD HIT SPECTRUM (WITH FOUR BASIC STATISTICS)

	T1	T2	T3	T4	ef	ep	nf	np
addToCart	1	1	1	1	2	2	0	0
removeFromCart	0	1	1	0	1	1	1	1
printProductsInCart	0	0	0	0	0	0	2	2
getProductCount	1	1	1	1	2	2	0	0
Test results	0	0	1	1				

TABLE III
TARANTULA SUSPICOUSNESS SCORES

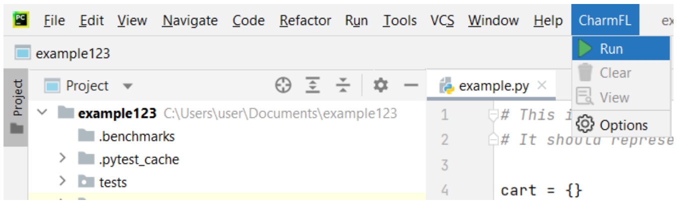
Method	Score
addToCart	0.58
removeFromCart	0.41
printProductsInCart	0.00
getProductCount	0.48

V. CHARMFL TOOL

In this section, we give an overview about our tool’s architecture, data processing, and user interface. Our tool can be divided into two parts; front-end and the back-end framework. The first part is the actual plug-in for the PyCharm IDE, which the user can interact with and use during debugging. We detail this part in Section V-A. The second part is a framework that gives the opportunity for developers to integrate fault localization in other IDEs. We give details on its architecture and usage in Section V-B.

A. GUI

The front-end part of the tool, shown in Figure 4, is an IDE specific plug-in using the CharmFL engine for the PyCharm IDE. After installing the plug-in and opening the Python project in the IDE, the user can run the fault localization process to get the list of program suspicious elements.



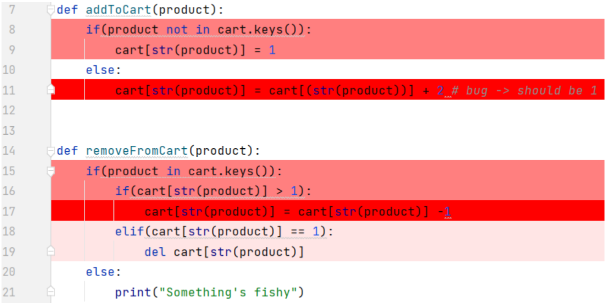
```
# This i
# It should represe

cart = {}
```

Fig. 4. CharmFL GUI

Additionally, the corresponding program elements are highlighted with different shades of red color based on the suspicious scores as shown in Figure 5. The darker the color is, the most suspicious the element is. If the user accidentally closes the results table, s/he can reopen it again by clicking on the View button in the CharmFL menu.

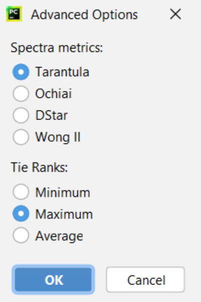
There is a set of advanced options for researchers too which appears via clicking the Options button of the menu as shown in Figure 6. Such options enable them to select different metrics for comparison and to apply different tie-breaking techniques to the elements sharing the same score in the ranking list.



```
def addToCart(product):
    if(product not in cart.keys()):
        cart[str(product)] = 1
    else:
        cart[str(product)] = cart[(str(product))] + 2 # bug -> should be 1


def removeFromCart(product):
    if(product in cart.keys()):
        if(cart[str(product)] > 1):
            cart[str(product)] = cart[str(product)] -1
        elif(cart[str(product)] == 1):
            del cart[str(product)]
    else:
        print("Something's fishy")
```

Fig. 5. Highlighted statements based on suspicious scores



Advanced Options

Spectra metrics:

- ● Tarantula
- ○ Ochiai
- ○ DStar
- ○ Wong II

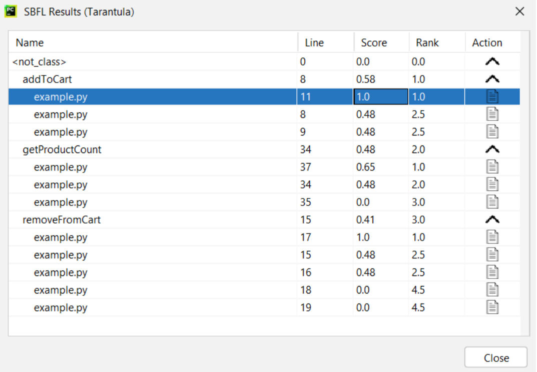
Tie Ranks:

- ○ Minimum
- ● Maximum
- ○ Average

Fig. 6. CharmFL advanced options

When the user selects multiple metrics, there will be a table for each metric, that way they can compare the elements side-by-side. This is especially good for researchers who would like to compare the efficiency of the supported SBFL metrics.

The SBFL results table (Figure 7) shows the program elements hierarchically, next to them there are their positions in the source code, their ranks, and their scores. Also, the Action button can be used to hide/show the elements inside each level of the hierarchy or to jump on a specific element via clicking on its corresponding document icon.



SBFL Results (Tarantula)

Name	Line	Score	Rank	Action
<not_class>	0	0.0	0.0	^
addToCart	8	0.58	1.0	^
example.py	11	1.0	1.0	🗎
example.py	8	0.48	2.5	🗎
example.py	9	0.48	2.5	🗎
getProductCount	34	0.48	2.0	^
example.py	37	0.65	1.0	🗎
example.py	34	0.48	2.0	🗎
example.py	35	0.0	3.0	🗎
removeFromCart	15	0.41	3.0	^
example.py	17	1.0	1.0	🗎
example.py	15	0.48	2.5	🗎
example.py	16	0.48	2.5	🗎
example.py	18	0.0	4.5	🗎
example.py	19	0.0	4.5	🗎

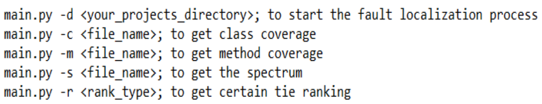
Fig. 7. CharmFL rankling list output

B. Framework's Architecture

This is the part of the tool where we gather and process the coverage and test result data. The framework can be used as a stand-alone tool or integrated in other IDEs too as a plug-in.

In order to collect the program's spectra, code coverage measurement is needed. To obtain the code coverage, the target program needs to be instrumented. For this purpose, our tool uses the popular coverage measuring tool for Python, called "coverage.py" [22]. This tool can measure on either statement or branch coverage levels, however in its current format it is not able to measure method or class coverage levels. Our framework transforms the statement level to method and class levels as shown in Figure 7. This is achieved by putting all the statements of each function under the corresponding function's name and then putting all the functions of each class under the corresponding class's name. Thus, each function will has its own set of statements and each class its own set of functions including the statements. Afterward, the classes are sorted based on their suspiciousness scores, then the functions, and finally the statements. For example, the statement at line 37 will not be examined before the statement at line 8 because the latter is belong to a function of higher rank in the ranking list. This hierarchical coverage feature gives additional useful information about the suspicious scores on all layers to the user. They can exclude whole methods or even classes based from the list.

Additionally, in order to make the coverage matrix, we used the ".coveragerc" file where the user can configure the measurement. After collecting the coverage report, we run tests using "pytest" [23] to fetch the results. Having those collected, we make coverage and test results matrices according to Jones et al. [8] from the raw data. Afterward, the tool calculates the suspiciousness score for each program element based on the equations as described in Section II. The framework provides class, method, and statement coverage levels; test results; coverage matrix and the hierarchical "ranking list". Figure 8 lists the usages of the framework.



```
main.py -d <your_projects_directory>; to start the fault localization process
main.py -c <file_name>; to get class coverage
main.py -m <file_name>; to get method coverage
main.py -s <file_name>; to get the spectrum
main.py -r <rank_type>; to get certain tie ranking
```

Fig. 8. Framework usage

VI. Potential Use Cases

When programmers face bugs in their Python programs, they have a couple of directions to go with in the debugging life-cycle. In order to find the bug, they can either run the test suite to figure out which test is failing and go from there, or they can inject break points in the code to investigate each value while pausing the program execution.

In this section, we will show three ways of how to use our tool in various phases of the debugging, as follows:

1) Running the test suite, then start the CharmFL tool.
2) Running the test suite, inject break points and then start the CharmFL tool.
3) Running the CharmFL tool, then inject break points.

For each scenario, we start from the point when the existence of the bug was first detected, i.e. someone reported the bug while using the software. For demonstration purposes, we will use a simple example project; but also any other Python project can be used. The example project has four methods, and four test cases that cover 90% of the program. We injected an artificial bug, in the 11th line, so two of the four test cases would fail. Our tool is successful if the 11th statement is in the top-10 of the list of suspicious elements, and the debugging is successful if the test cases pass after the bug is fixed.

First, we demonstrate the usage of our tool after running the test suite. Having done that, we get from pytest's report that there are two failing test cases. We open the test files, meanwhile we start the CharmFL tool. Reading the pytest's test results report, we can see that there are two failing test cases. Additionally, we can see that in the test cases the method "addToCart" is called two times and the "removeFromCart" is called once. Hence, we start with the examination of the "addToCart" method. When we investigate the method we can see that there are three statements. At this point, we start the CharmFL tool and use it to decide which statement to investigate first. We click on the first element in the method with the highest score in the list of suspicious elements and try to fix the bug in the statement. We run the test cases again and see that all test cases pass. In this scenario, our tool helped deciding which statement should be investigated first, hence saving time on debugging.

Second, we use our tool with a bit more advanced debugging technique; break-point oriented debugging. The PyCharm IDE has a built-in debugger, which is the best option to use alongside our tool. First, we run the tests and investigate the failing ones similarly like in the previous scenario. Again, we have two failing test cases that cover the "addToCart" and "removeFromCart" methods. Then, we insert break points to those lines that are covered by the failing test cases. Next, start the debugging session to investigate what values do the variables take and what is not going according to the plan. Meanwhile, we start the CharmFL tool to get the most suspicious elements. We can see in Figure 5 that the 11th statement in the example.py file is dark red, which is very suspicious. We fix the bug in the statement, then verify the fix by running the tests again. We can conclude from this scenario, that checking against the suspiciousness list can help the programmer a lot with the debugging.

The final scenario, we start the CharmFL tool before doing any debugging. In this scenario, we start the tool then look at the list of suspicious elements.

Developers tend to investigate only the first ten (also referred to as top-10) elements in the ranking list because after that, they start to lose interest in using the tool [24], [25]. Therefore, a fault localization algorithm is successful if it can fit as many faulty elements in the top-10 list as possible.

Using this technique, we look at and click on the element with the highest score in the suspiciousness list table shown in Figure 7. The tool then redirects us to the statement we want to investigate. The background color refers to the suspiciousness level, i.e. how likely the statement is to contain a fault, the darker the color the higher the suspicion is. We can see in Figure 5 that the statement has a dark red background, meaning it is the most likely to contain a bug. When investigating the element we can use break-point-based debugging. Without running the test cases, we can place a break-point to the statement we clicked on. We fix the statement and run the tests for verification. This scenario takes a few more extra steps. However, this is a helpful guide when the test cases are well defined and maintained. In this case, our tool can reduce the excessive time and energy that would have been spent on debugging.

VII. Conclusions

This paper describes "CharmFL"[1], an Open-source fault localization tool for Python programs. The tool is developed with many interesting features that can help developers debugging their programs by providing a hierarchical list of ranked program elements based on their suspiciousness scores. The applicability of our tool has been evaluated via different use cases. The tool has been found to be useful for locating faults in different types of programs and it is easy to use. For the future work, we would like to implement interactiveness to enable the user to give his/her feedback on the suspicious elements to help re-rank them, thus improving the fault localization process. Also, we would like to add other features such as displaying the tool's output using different visualization techniques. Finally, assessing the tool with real users and in real-world scenarios would be a valuable next step too.

Acknowledgements

The research was supported by the Ministry of Innovation and Technology, NRDI Office, Hungary within the framework of the Artificial Intelligence National Laboratory Program, and by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology. Qusay Idrees Sarhan was supported by the Stipendium Hungaricum scholarship programme.

References

[1] C. Gouveia, J. Campos, and R. Abreu, "Using html5 visualizations in software fault localization," in *First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–10.

[2] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges," pp. 1–46, jul 2016. [Online]. Available: http://arxiv.org/abs/1607.04347

[3] 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, aug 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7390282/

[4] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: A Literature Review," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 5, pp. 1–8, sep 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2659118.2659125

[5] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges," pp. 1–46, jul 2016. [Online]. Available: http://arxiv.org/abs/1607.04347

[6] J. Kim and E. Lee, "Empirical evaluation of existing algorithms of spectrum based fault localization," in *The International Conference on Information Networking (ICOIN)*. IEEE, feb 2014, pp. 346–351. [Online]. Available: http://ieeexplore.ieee.org/document/6799702/

[7] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 332–347, 2021.

[8] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, 2002, pp. 467–477.

[9] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2006, pp. 39–46.

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

[11] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1, 2007, pp. 449–456.

[12] O. C. Chesley, X. Ren, B. G. Ryder, and F. Tip, "Crisp - A fault localization tool for Java programs," *Proceedings - International Conference on Software Engineering*, pp. 775–778, 2007.

[13] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," *Proceedings - International Conference on Software Engineering*, pp. 301–310, 2008.

[14] 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, 2009.

[15] T. Janssen, R. Abreu, and A. J. Van Gemund, "Zoltar: A spectrum-based fault localization tool," *SINTER'09 - Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution at Runtime*, pp. 23–29, 2009.

[16] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: An eclipse plug-in for testing and debugging," *2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings*, pp. 378–381, 2012.

[17] N. Wang, Z. Zheng, Z. Zhang, and C. Chen, "FLAVS: A fault localization add-in for visual studio," *Proceedings - 1st International Workshop on Complex Faults and Failures in Large Software Systems, COUFLESS 2015*, pp. 1–6, 2015.

[18] C. Chen and N. Wang, "UnitFL: A fault localization tool integrated with unit test," *Proceedings of 2016 5th International Conference on Computer Science and Network Technology, ICCSNT 2016*, pp. 136–142, 2017.

[19] H. L. Ribeiro, H. A. De Souza, R. P. A. De Araujo, M. L. Chaim, and F. Kon, "Jaguar: A Spectrum-Based Fault Localization Tool for Real-World Software," *Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation, ICST 2018*, pp. 404–409, 2018.

[20] "Fault-localization tool," https://pypi.org/project/fault-localization/, accessed: 01-06-2021.

[21] "Pinpoint tool," https://pypi.org/project/pytest-pinpoint/, accessed: 01-06-2021.

[22] "Coverage.py tool," https://coverage.readthedocs.io/en/coverage-5.5/, accessed: 01-06-2021.

[23] "Pytest tool," https://docs.pytest.org/en/6.2.x/, accessed: 01-06-2021.

[24] 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*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 165–176. [Online]. Available: https://doi.org/10.1145/2931037.2931051

[25] X. Xia, L. Bao, D. Lo, and S. Li, "Automated debugging 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)*, 2016, pp. 267–278.

[1] https://sed-szeged.github.io/SpectrumBasedFaultLocalization/