Localizer: A Visual Debugging Assistant for Python Programs

Khan, Shehroz ; Sudheerbabu, Gaadha; Truscan, Dragos; Ahmad, Tanwir

Published in:
DEBT 2024 - Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques, Co-located with

DOI:
10.1145/3678720.3685321

Published: 13/09/2024

Document Version
Final published version

Localizer: A Visual Debugging Assistant for Python Programs

Shehroz Khan
shehroz.khan@abo.fi
Åbo Akademi University
Finland

Gaadha Sudheerbabu
gaadha.sudheerbabu@abo.fi
Åbo Akademi University
Finland

Dragos Truscan
dragos.truscan@abo.fi
Åbo Akademi University
Finland

Tanwir Ahmad
tanwir.ahmad@abo.fi
Åbo Akademi University
Finland

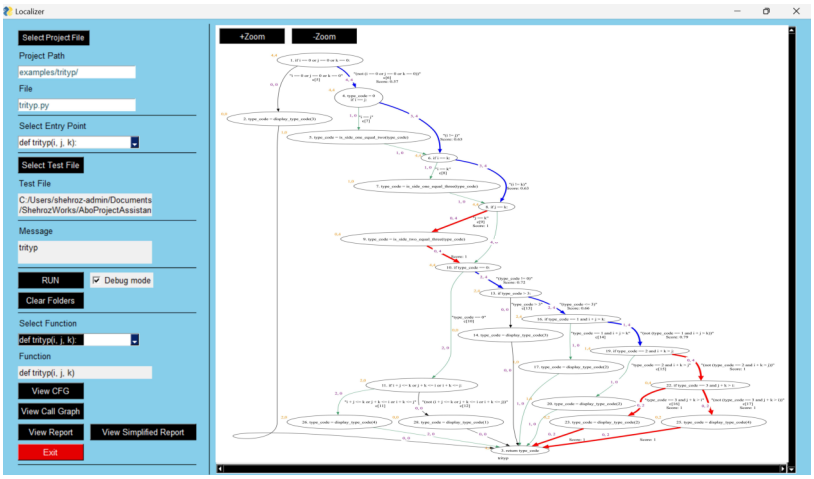

Figure 1: GUI of the Localizer tool

Abstract

We present the Localizer tool, which is targeted at assisting developers and testers in debugging their Python code. The tool combines program slicing and program spectra analysis to analyze the difference between the execution paths of the passing tests and of the failing tests, respectively. In addition, it suggests suspicious parts of the code where the fault can be potentially located. To assist the user in inspecting the code, the tool graphically presents the structure of the code as control flow graphs annotated with program spectra information. Preliminary studies show that the tool facilitates the identification of faults by reducing the complexity of the code analysis process.

CCS Concepts

• **Software and its engineering → Software testing and debugging**;

Keywords

Spectrum-based fault localization, program debugging, dynamic analysis, program spectra, program slicing

DEBT '24, September 19, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1110-7/24/09
https://doi.org/10.1145/3678720.3685321

ACM Reference Format:
Shehroz Khan, Gaadha Sudheerbabu, Dragos Truscan, and Tanwir Ahmad. 2024. Localizer: A Visual Debugging Assistant for Python Programs. In *Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques (DEBT '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3678720.3685321

1 Overview of Localizer Tool

The graphical user interface of the Localizer tool is presented in Figure 1. The tool combines two techniques, dynamic program slicing and spectrum-based fault localization [1]. It takes as input a Python program (program under debugging - PUD) and a file containing two sets of tests which have previously been run and to which an either pass or fail verdict has been assigned. If the PUD has multiple methods, the user will be asked to select the one for which the tests are defined.

On pressing RUN, the Localizer automatically instruments the program code with counter variables for each basic block corresponding to a branching condition in the program, and then it executes both the passing tests and the failing tests. The values of the counter variables are analyzed, and the program spectra are created.

As output, the Localizer produces a call graph (CG) of the PUD, showing the sub-methods that are being called by the top method and control-flow graph (CFG). Both types of graphs are annotated with spectra information collected during the execution of the tests.

To exemplify, we use the classic example of a triangle classification program, *trityp* (see code excerpt in Listing 1), which receives as input parameters the dimension of the sides of a triangle and classifies it a scalene, isosceles, equilateral or invalid triangle. The

program contains a fault, and a set of 4 passing and 4 failing tests are available (see Table 1).

Listing 1: Top method of the `trityp` program

```
1  def trityp(i, j, k):
2      if i == 0 or j == 0 or k == 0:
3          type_code = display_type_code(3)
4      else:
5          type_code = 0
6          if i == j:
7              type_code = is_side_one_equal_two(type_code)
8          if i == k:
9              type_code = is_side_one_equal_three(type_code)
10         if j == k:
11             type_code=is_side_two_equal_three(type_code)
12         if type_code == 0:
13             if (i + j <= k) or (j + k <= i) or (i + k <= j):
14                 type_code = display_type_code(4)
15             else:
16                 type_code = display_type_code(1)
17         elif type_code > 3:
18             type_code = display_type_code(3)
19         elif (type_code == 1) and (i + j > k):
20             type_code = display_type_code(2)
21         elif (type_code == 2) and (i + k > j):
22             type_code = display_type_code(2)
23         elif (type_code == 3) and (j + k > i):
24             type_code = display_type_code(2)
25         else:
26             type_code = display_type_code(4)
27     return type_code
```

Table 1: Passed and Failed Tests for the 'trityp' Program

Passed Tests	Failed Tests
(10, 10, 12), (3, 8, 3), (2, 5, 2), (4, 6, 4)	(12, 10, 10), (8, 3, 3), (5, 2, 2), (6, 4, 4)

The resulting annotated CFG of this program is shown in Figure 1. Each node, representing a basic block in the program, is annotated with a pair *p,f* depicting the number of passing tests and, respectively, failing tests that have visited that basic block during the execution of the program. The edges are annotated with the number of passing and failing tests that visited each edge *p,f* and are coloured for easier identification in black - visited by no tests, green - visited only by passing tests, red - visited only by failing tests, and blue - visited by both passing and failing tests. In addition, the edges that are associated with the instrumentation counters display the index of that counter (e.g., *c[7]* for edge [4,5]). Based on this information, spectra analysis is performed, and a *suspiciousness score*, with a value between 0 and 1, is computed for each edge with a counter using the metrics supported by the tools, such as Ochiai, Jaccard, and Tarantula [2, 3]. The suspiciousness score will give a hint on the probability of a fault to be present in the statements associated with a given edge. To highlight this on the CFG, the width of the edge is proportional to the value of the suspiciousness score.

The call graph of the top method is also annotated with suspiciousness information. For instance, Figure 2 shows the call graph annotated with maximum scores for each method in *trityp* Python program and the methods with the highest suspiciousness score are highlighted in red.

Upon inspection of the call graph, one can notice that the fault is potentially located in the top method and, more specifically, in the method being called by it `is_side_two_equal_three`. Further inspection of the CFG of the top method (see Figure 1) shows a first suspicious set of elements adjacent to node 9, which also pinpoints that the fault may originated from the statement `type_code = is_side_two_equal_three(type_code)`, corresponding to line 11 in Listing 1.

Figure 2: Call Graph for trityp.py program

In the same CFG, additional suspicious elements are highlighted (in this case in red) and upon further analysis, it can be observed that they are related to the same variable as previously, namely `type_code`. Upon inspection of the method, `is_side_two_equal_three` fault is identified. Due to space reasons, we omit the code of this method.

As a further help for the user, Localizer includes in the report a simplified version of the CFGs for all methods in which only the parts executed by the tests are executed in order to reduce the complexity of manual analysis and allow the user to focus on relevant parts of the code. Lastly, to facilitate the analysis of variables used in the basic blocks with the highest suspiciousness score, the report provides a list of statements in different methods involved that can be filtered by the suspiciousness score (see Figure 3).

Threshold: 0.57 Update the Table

Module Name	Max Score	Susp. Score	Block ID	Statements
is_side_two_equal_three *(View Code)*	1	1	1	if type_code >= 0:
trityp *(View Code)*		0.63	4	type_code = 0
				if i == j:
			6	if i == k:
		1	8	if j == k:
			9	type_code = is_side_two_equal_three(type_code)
		0.72	10	if type_code == 0:
		0.66	13	if type_code > 3:
		0.79	16	if type_code == 1 and i + j > k:
		1	19	if type_code == 2 and i + k > j:
			22	if type_code == 3 and j + k > i:
			23	type_code = display_type_code(2)
			25	type_code = display_type_code(4)

Figure 3: Table showing suspicious statements across all methods

2 Conclusions

The tool has been evaluated on several industrial-grade and academic examples, and preliminary results show that it brings benefits in reducing the time needed by the user to identify the location of the faults [2]. Further work will improve the tool with automated data-flow analysis techniques that will allow the user to isolate better the source of the failure in the presence of multiple faults.

References

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A Practical Evaluation of Spectrum-Based Fault Localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[2] Gaadha Sudheerbabu, Tanwir Ahmad, Dragos Truscan, and Jüri Vain. 2023. Metamorphic Testing for Verification and Fault Localization in Industrial Control Systems. In *CyberSecurity in a DevOps Environment: From Requirements to Monitoring*. Springer, 127–159.

[3] Yong Wang, Zhiqiu Huang, Bingwu Fang, and Yong Li. 2018. Spectrum-Based Fault Localization via Enlarging Non-Fault Region to Improve Fault Absolute Ranking. *IEEE Access* 6 (2018), 8925–8933.

Received 2024-06-21; accepted 2024-07-22