Improving Spectrum Based Fault Localization For Python Programs Using Weighted Code Elements

Qusay Idrees Sarhan and Arpad Beszedes
{sarhan, beszedes}@inf.u-szeged.hu
Department of Software Engineering, University of Szeged.

Objective

We present an approach for improving Spectrum-Based Fault Localization (SBFL) by integrating static and dynamic information about code elements. This is achieved by giving more importance to code elements that include mathematical operators and those that appear in failed tests. The intuition is that these elements are more likely to have bugs than others.

Code Example

```
def do_avg(nums):
    S1: n=len(nums)
    S2: s = 0
    S3: for i in nums:
    S4:     s = s + i
    S5: avg = s / 2
    S6: return round(avg, 1)
```

```
import avg
def test_T1():
    nums = [0, 0, 0]
    assert avg.do_avg(nums) == 0.0
def test_T2():
    nums = [2, 2, 2]
    assert avg.do_avg(nums) == 2.0
def test_T3():
    nums = [2, 2, 3]
    assert avg.do_avg(nums) == 2.3
def test_T4():
    nums = [4, -2, -2]
    assert avg.do_avg(nums) == 0.0
```

Fig. 2. Running example – code and test cases

TABLE I
RUNNING EXAMPLE – SPECTRA AND BASIC STATISTICS

	T1	T2	T3	T4	ef	ep	nf	np
S1	1	1	1	1	2	2	0	0
S2	1	1	1	1	2	2	0	0
S3	1	1	1	1	2	2	0	0
S4	1	1	1	1	2	2	0	0
S5	1	1	1	1	2	2	0	0
S6	1	1	1	1	2	2	0	0
Results	0	1	1	0				

TABLE II
RUNNING EXAMPLE – SCORES AND RANKS

	Tarantula score	Rank	Tarantula *	Rank *
S1	0.5	3.5	0.5	4.5
S2	0.5	3.5	0.5	4.5
S3	0.5	3.5	0.5	4.5
S4	0.5	3.5	1.0	1.5
S5	0.5	3.5	1.0	1.5
S6	0.5	3.5	0.5	4.5

The Tarantula formula (Equation 1) was applied to the execution information in Table I to compute the suspiciousness score of each statement as presented in Table II. However, after applying our proposed approach denoted with *, the faulty statement got a higher rank based on (Equation 2) and it will be examined before most of the other statements.

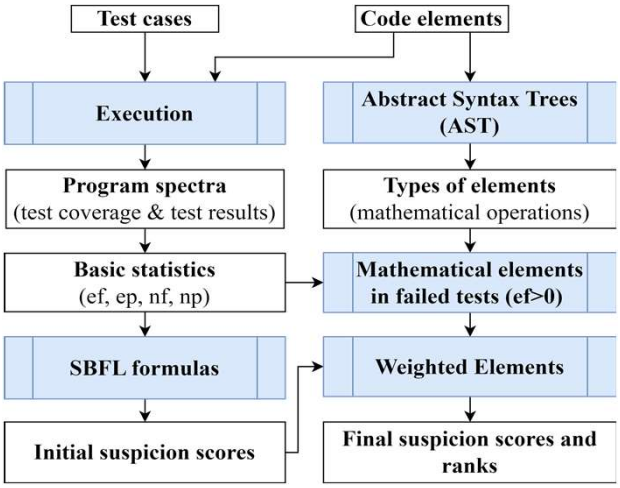
$$Tarantula = \frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}} \quad (1)$$

$$Final\_Score = Initial\_Score + Max\_Initial\_Score \quad (2)$$

Equation 2 ensures that the mathematical statements will be examined before other types by adding the highest score (Max_Initial_Score) in the ranking list to each mathematical statement's base score (Initial_Score).

How Does Our Approach Work?

After extracting the Abstract Syntax Trees (AST), we walk through them and collect only the mathematical statements that appeared in failed test cases (i.e., when ef>0). Then, we give them more importance by using (Equation 2), and finally, we rank all the code elements.



Conclusion

We improve the effectiveness of SBFL by involving static information (i.e., types of code elements) in the SBFL process for Python programs. Presently, we considered one type of statement: mathematical statements. Based on the positive results of this approach, we believe that it could be explored more in the future, such as by experimenting with other statement types and other ways for the weighting.