Iterative User-Driven Fault Localization

Authors: Xiangyu Li, Marcelo d’Amorim, Alessandro Orso

Presented by Xianhao Jin

Outline

1. Problem Statement
2. Background Knowledge
3. Approach
4. Evaluation
5. Related work
6. Conclusion
7. Discussion

Problem Statement

1. Debugging contributes greatly to software development costs.
2. Existing statistical fault localization techniques are unrealistic.
    a. examine a long list
    b. recognize faulty lines by simply looking at them
3. There is a disconnect between research and practice.
4. A tool which helps debugging in a natural way is in demand.

Background Knowledge - Ochiai

$$Ochiai(element) = \frac{e_f}{\sqrt{(e_f + n_f) \cdot (e_f + e_p)}}$$

e_f	Number of failed tests that execute the program element.
e_p	Number of passed tests that execute the program element.
n_f	Number of failed tests that do not execute the program element.
n_p	Number of passed tests that do not execute the program element.

Problem Statement - Example

```
1  public class BoundedStack {
2   Integer[] elems; int numElems;
3
4   BoundedStack(int max) {
5     elems = new Integer[max]; }
6
7   void push(Integer k) {
8    /* check size */
9    elems[numElems++] = k; }
10
11  void pop() { --numElems; }
12
13  Integer peek() {
14   if (size() == 0)
15     return null;
16   else return elems[size()-1];}
17
18  void clear() { numElems = 0; }
19
20  int size() { return numElems; }
21  ...
22 }
23
```

SFL Limitation:

Rank Line 18 as the most suspicious, Line 15 as the least suspicious.

```
24 // tests
25 @Test t1() {
26   BoundedStack bs =
27     new BoundedStack(3);
28   bs.push(5); bs.push(6);
29   bs.pop();
30   assertEquals(5, bs.peek());}
31
32 @Test t2() {
33   BoundedStack bs =
34     new BoundedStack(3);
35   bs.push(7); bs.push(8);
36   bs.clear();
37   bs.pop();
38   assertEquals(null, bs.peek());
```

Approach - Four Steps

1. Test Execution.
2. Fault localization.
3. Query Generation.
4. Feedback Incorporation.

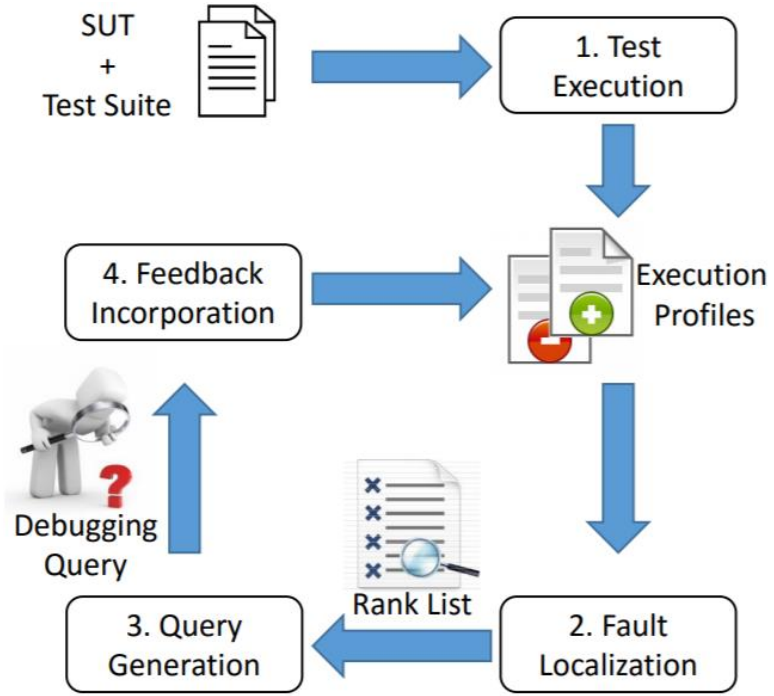


Fig. 3: Workflow overview

Approach - Test Execution

Swift executes the test suite for the SUT and collects an execution tree for each test.

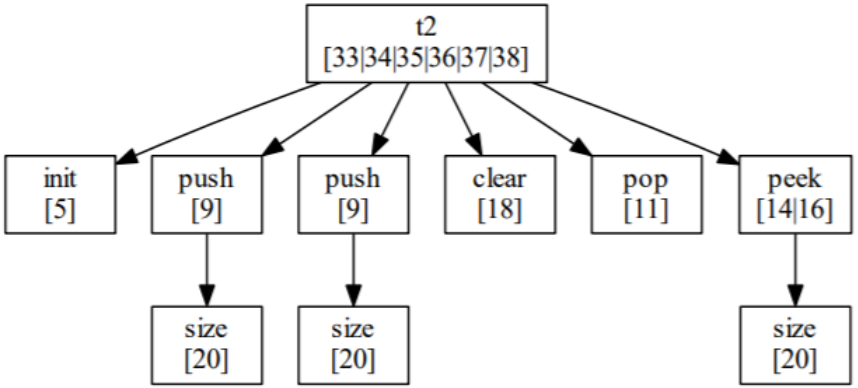


Fig. 4: Execution tree corresponding to test `t2` (see Figure 2).

Approach - Fault Localization

Swift leverages existing fault localization techniques to compute the suspiciousness of program entities based on the collected runtime information.

Table 1: Example of coverage and suspiciousness information for the `BoundedStack` example.

	✓ t_1	✗ t_2	✓ vt	$susp.$
5	1	1	0	0.7
9	1	1	1	0.6
11	1	1	0	0.7
14	1	1	0	0.7
15	0	0	0	0.0
16	1	1	0	0.7
18	0	1	0	1.0
20	1	1	1	0.6

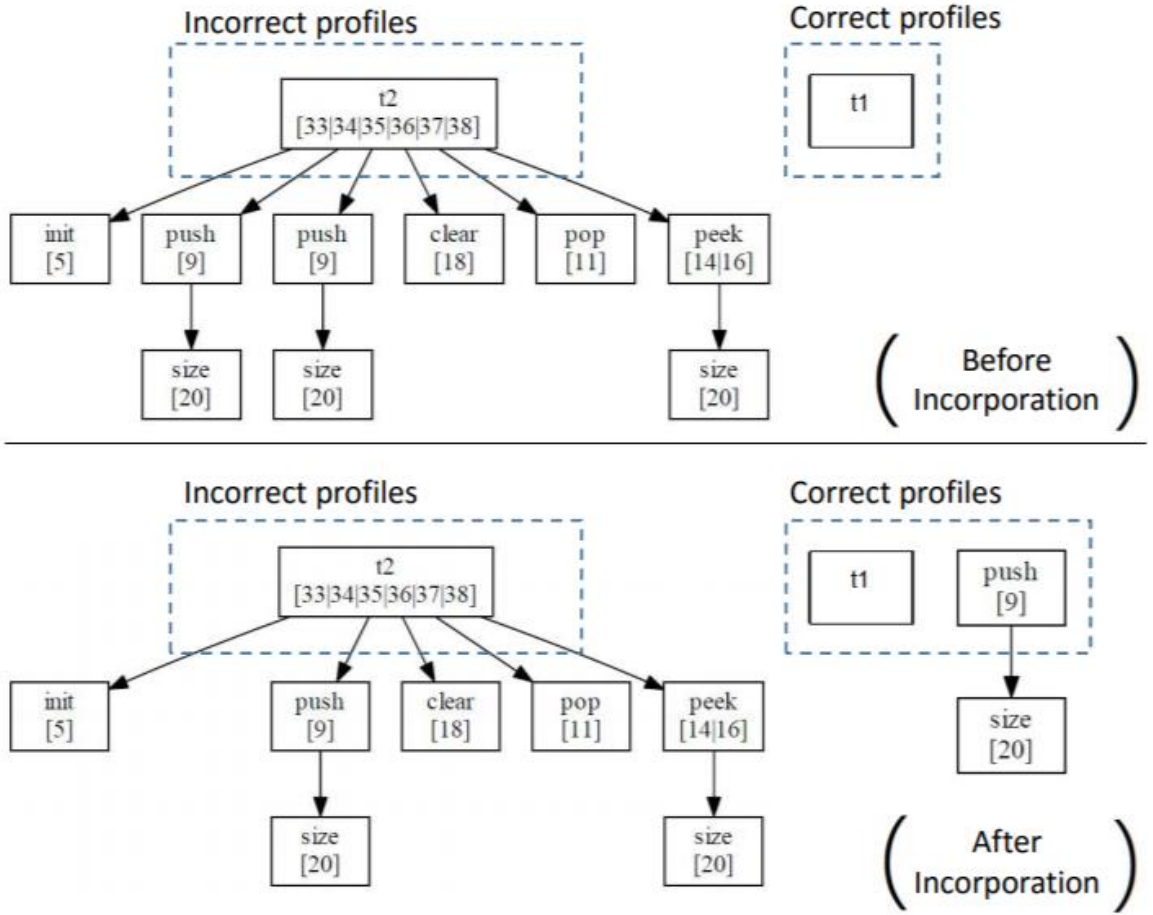
Approach - Query Generation

Swift asks developers for feedback through debugging queries, which basically consist of the input and output of a method invocation. Developers are expected to assess the correctness of the computation for that invocation.

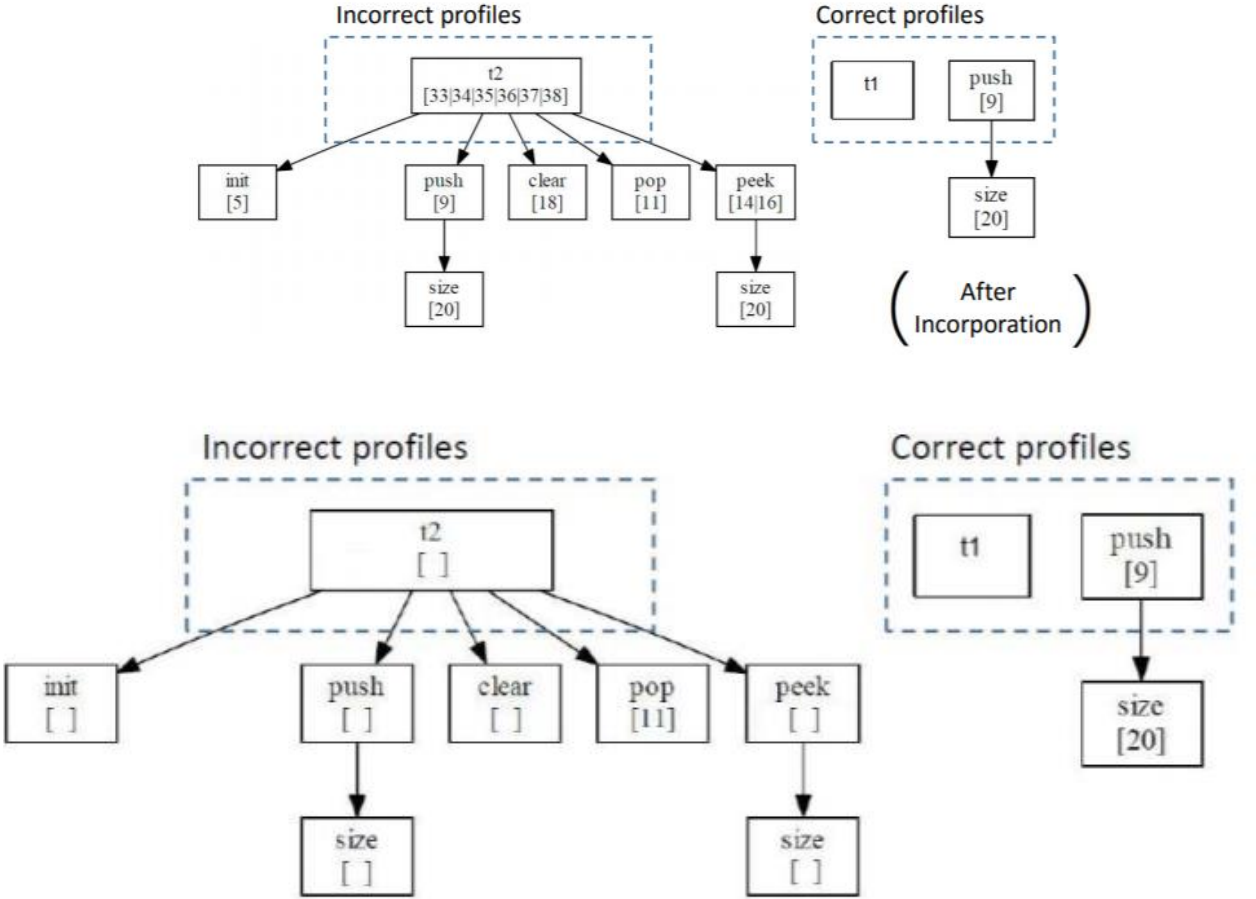
Approach - Feedback Incorporation

```
1  class Feedback {
2   Invocation invocation;
3   Invocation fromProfile;
4   bool isCorrect;
5  }
6
7  List<Invocation> correctProfiles =
8  List<Invocation> incorrectProfiles = ...
9
10 incorporateUserFeedback(Feedback feedback) {
11  if (feedback.isCorrect) {
12   feedback.invocation.removeFromParent();
13   correctProfiles.add(feedback.invocation);
14   for (Statement s
15     : feedback.invocation.getCoverage()) {
16    if (!feedback.fromProfile.covers(s)) {
17     for (Invocation incorrectProfile
18       : incorrectProfiles) {
19      incorrectProfile.removeCoverage(s); }}}
20  } else {
21   for (Invocation incorrectProfile
22     : incorrectProfiles) {
23    for (Statement s
24      : incorrectProfile.getCoverage()) {
25     if (!feedback.invocation.covers(s)) {
26      incorrectProfile.removeCoverage(s); }}}}}
```

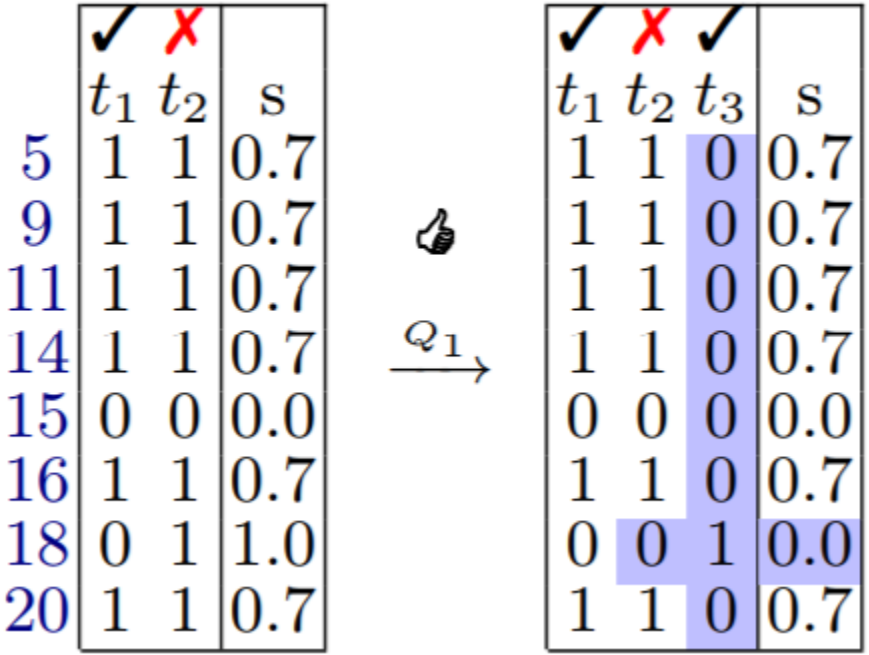
Approach - Feedback Incorporation



Approach - Feedback Incorporation



Approach - Complete Debugging Session

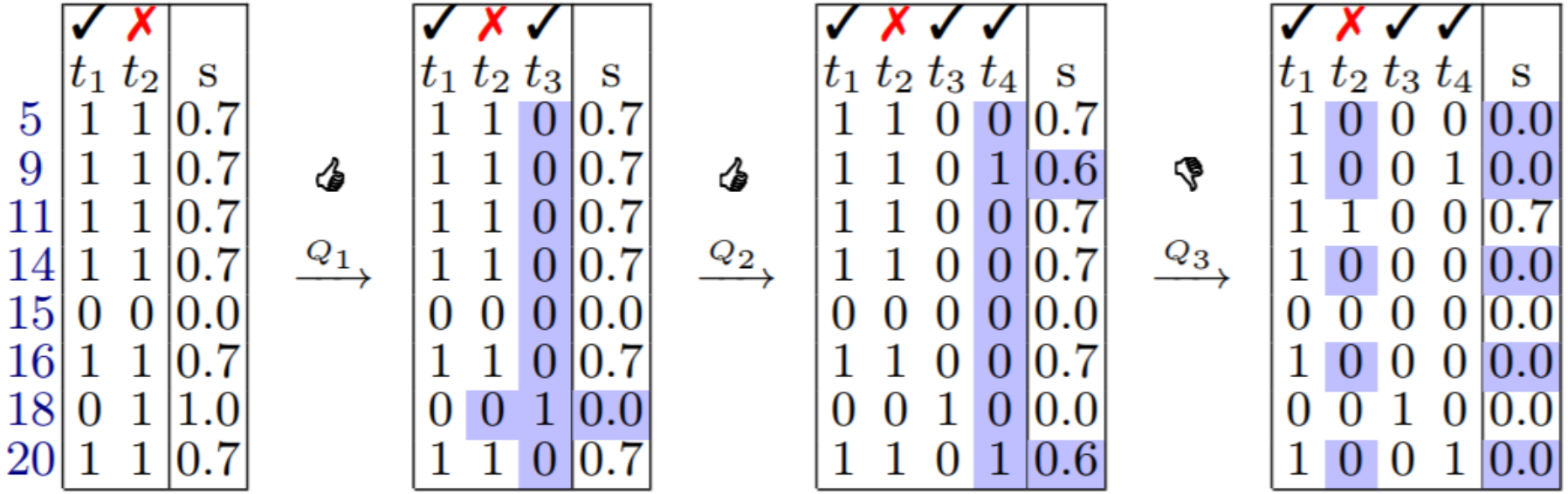


Q_1: `BoundedStack.clear()`#0 in t_2	
Input:	Output:

this: {
elems: {7, 8, null}
numElems: 2
}
	this: {
	elems: {7, 8, null}
	numElems: 0
	}

Fig. 7: Debugging query Q_1.

Approach - Complete Debugging Session



Approach - Complete Debugging Session

✓	✗	✓	✓	
t_1	t_2	t_3	t_4	s
1	1	0	0	0.7
1	1	0	1	0.6
1	1	0	0	0.7
1	1	0	0	0.7
0	0	0	0	0.0
1	1	0	0	0.7
0	0	1	0	0.0
1	1	0	1	0.6

👎

Q_3 →

✓	✗	✓	✓	
t_1	t_2	t_3	t_4	s
1	0	0	0	0.0
1	0	0	1	0.0
1	1	0	0	0.7
1	0	0	0	0.0
0	0	0	0	0.0
1	0	0	0	0.0
0	0	1	0	0.0
1	0	0	1	0.0

Q_3: `BoundedStack.pop()` #0 in t_2	
Input:	Output:

this: {
elems: {7, 8, null}
numElems: 0
}
	this: {
	elems: {7, 8, null}
	numElems: -1
	}

Fig. 9: Debugging query Q_3.

Evaluation

1. Data Set : 26 faults from 5 open-source applications
2. Research Questions:
    a. Can Swift locate the fault with a small number of debugging queries?
    b. How does user feedback affect fault ranking?

Evaluation – Experimental Setup

1. Apply SWIFT and record the number of queries.
2. Record the query, its answer, and the updated ranking.
3. Use an automated oracle to answer queries.

Evaluation - Result Summary

Subject	*Repo.*	*Fault ID*	*P-F*	#Cls.	#Meths.	kLOC
jtopas	[2]	FAULT_2	123-3	25	251	7
		FAULT_6	125-1	25	251	7
commons-math	[2]	C_AK_1	1162-1	236	1723	43
		EDI_AK_1	1162-1	236	1723	43
		F_AK_1	1162-1	236	1723	43
		M_AK_1	1162-1	236	1723	43
		VS_AK_1	1162-1	236	1723	43
		CDI_AK_1	2048-2	477	3899	83
		MU_AK_1	2048-2	477	3899	83
		MU_AK_4	2049-1	477	3899	83
		URSU_AK_1	2048-2	477	3899	83
xml-security	[2]	CN2_AK_2	89-2	198	1278	40
		C2E_AK_1	92-2	198	1275	41
jsoup	[1]	1_3__4_b3	225-1	75	611	8
		1_4_2_b2	295-1	89	698	9
		1_5_2_b2	236-4	86	682	9
		1_5_2_b5	243-1	86	682	9
		1_6_1_b1	290-2	198	979	13
		1_6_3_b3	323-1	206	1032	14
commons-lang	[16]	b6	2125-3	169	2281	57
		b9	2057-8	170	2224	54
		b10	2055-8	170	2224	54
		b16	1913-1	160	2142	53
		b24	1698-1	143	2022	50
		b26	1677-1	139	2000	50
		b39	1566-1	123	1835	45

Evaluation - Result

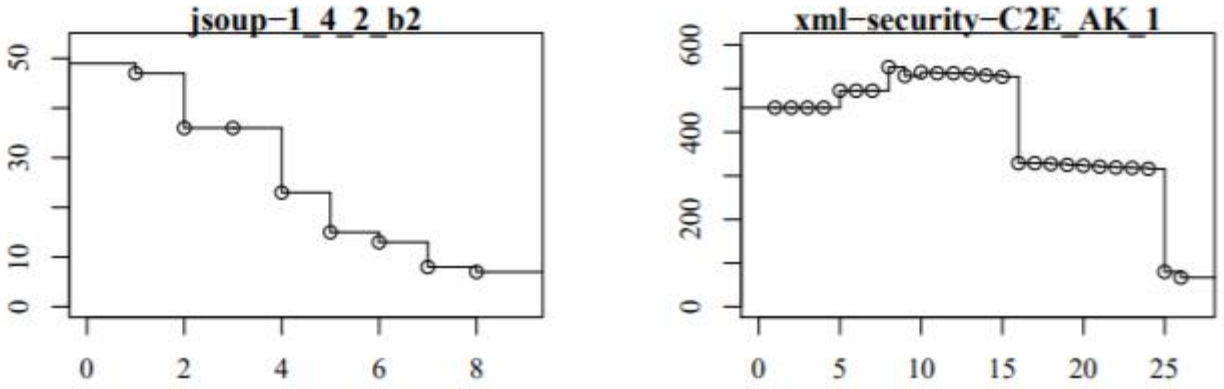


Fig. 10: Progress of stmt.-level suspiciousness.

Evaluation – Effect of Incorporation

1. Assess how effective the updates to the ranking list are for guiding the search of the faulty method invocation.
2. Measure the number of queries that would be generated if Swift did not update the fault localization results using the answers to debugging queries.
3. Start from the beginning of the ranking list, present all method invocations, then go to the next statement.

Related Work

		Differences:
2004	Designing the whyline: A debugging interface for asking questions about program behavior.	Tool asks questions instead and focus on suspicous parts.
2009	Vida: Visual interactive debugging.	Generate user queries at the level of abstractions of methods which is more likely to be understood.
2012	Interactive fault localization leveraging simple user feedback.	Ask questions about concrete input-output pairs and does not rely on developers' ability to assess the correctness.
2015	A user-guided approach to program analysis.	Dynamic rather than static analysis that supports debugging rather than bug finding.

Conclusion

Swift, a technique that aims to mitigate the existing disconnect between research and practice in the area of software debugging, and in particular in fault localization.

Swift operates in an iterative and user-driven fashion.

This process allows Swift to improve the localization results and guide the developer increasingly closer to the fault at hand.

Conclusion - Contributions

1. A novel technique that overcomes some of the limitations of existing SFL approaches by leveraging user feedback in a natural way.
2. An implementation of our approach for Java programs that is publicly available.
3. An empirical 2 evaluation that provides initial evidence of the potential usefulness of our approach and identifies several directions for future work.

Discussion

1. The idea of interaction with the developers?
2. Select randomly when ties, or other choices?
3. How about accuracy not 100%?
4. Try different fault localization techniques?
5. How much does the technique rely on each step?
6. User study?

Thank you!