Fault-Localization Using Dynamic Slicing and Change Impact Analysis

[†]Elton Alves, [‡]Milos Gligoric, [‡]Vilas Jagannath, and [†]Marcelo d'Amorim
[†]Federal University of Pernambuco, Brazil
[‡]University of Illinois at Urbana-Champaign, USA
[†]{erma,damorim}@cin.ufpe.br, [‡]{gliga,vbangal2}@illinois.edu

***Abstract*—Spectrum-based Fault-Localization tools, such as Tarantula, have been shown to be effective at guiding developers towards faulty statements in a system under test. These tools report statements ranked in order of suspiciousness. Unfortunately, these tools often report statements that are unrelated to the error. This paper evaluates the impact of several approaches to ignore these statements from the rankings.**

I. Introduction

Spectrum-based fault-localization is an automated approach for assisting programmers with quick localization of errors made during development. Tarantula [1] is one reference implementation for this approach. It takes as input the test suite coverage information and outputs covered statements ranked in order of suspiciousness (according to some metric). Conceptually, statements that appear relatively more often in failing runs than passing runs are considered more suspicious. The typical estimate of *inspection cost* is given by the number of statements the user needs to inspect in ranked order before finding the error. Unfortunately, Tarantula often reports a large number of statements that have to be inspected [2]. Our goal is to reduce the inspection cost of Tarantula without significantly increasing computational cost. We propose the use of additional analysis information that enables the removal of likely non-faulty statements from Tarantula's ranking.

II. Tarantula

Tarantula is a technique that uses code coverage information, obtained from a test suite run, to help developers localize faults efficiently [1]. As the input, Tarantula accepts a test suite and a subject under test (SUT). As the output, if there is at least one failing test, it produces a ranking of the statements in the SUT based on a *measure of suspiciousness*. The higher a statement appears in the ranking, the higher its importance is to the reproduction of a bug. Tarantula instruments the program to collect statement coverage and calculates a measure of suspiciousness (according to some heuristics) for each statement based on the coverage information and the pass/fail result of each test run. The technique generalizes to other types of coverage metrics, such as branch or def-use [2]. Abreu *et al.* [3] and Santelices *et al.* [2] found empirically that the *Ochiai* measure was the most effective to determine suspiciousness. Santelices *et al.* [2] define suspiciousness of a statement s as $susp(s) = failed(s)/\sqrt{totfailed * (failed(s) + passed(s))}$, where $failed(s)$ denotes the number of failing tests that cover statement s; $passed(s)$ denotes the number of passing tests that cover s and; $totfailed$ denotes the total number of failing tests. We refer to inspection cost in this paper as the *number of statements that need inspection in the ranking before finding the fault*.

A. Example

We use a faulty implementation of a data-structure and a fault-revealing test to illustrate how the techniques work. Figure 1 shows the outline of a class implementing a binary search tree. The `BST` class uses an inner class, called `Node`, to encapsulate the contents of the tree node. The field `root` points to a root node of a tree, and the field `size` stores the number of nodes. As usual, values in the left subtree of a given node should be less than that node's value; and in the right subtree values should be greater. Figure 1 shows two versions of the `insert` method. The lines labeled with "V1" belong to the original (correct) version of the code. The modified version is obtained by removing those lines and adding lines labeled with "V2". The original version of the code permits duplicates in a tree. The buggy version is a result of an attempt to exclude duplicates. Unfortunately, this change introduces a bug. Instead of using the relational operator > at line 23 the operator >= is used. The test below reveals a violation of the "no-duplicates" rule due to the bug.

```
BST bst = new BST(); bst.insert(3);
bst.insert(3); assert(bst.size==1);
```

Table I shows statement coverage for five test cases that are executed on the buggy version of the running example. Each test case under column "Test Cases" consists of a sequence of calls to the `insert` method and an assertion that checks the size of the tree. For example, "3 3" is a shorthand for the test above. As it can be seen only this test reveals the fault. We use a black circle to indicate coverage of a statement (column "Stmt") during the execution of a test case (under the group "Test Cases"). Column "*Susp.*" shows the suspiciousness score calculated for each of the statements according to the Ochiai metric and column "$R0$" shows the rank of each statement. In this case, the faulty line 23 appears in the first rank with a suspiciousness score of 70.7%. This rank includes a total of 4 lines, which is the cost of Tarantula. Columns "$R1$", "$R2$", and "$R3$" show the rank yielded by the proposed techniques.

TABLE I
RANKING OF BST STATEMENTS FOR A SUITE WITH 4 TEST CASES.

Stmt	Test Cases				*Susp.*	Ranks			
	5	4 3 2	6 7 8	3 3		$R0$	$R1$	$R2$	$R3$
6	•	•	•	•	0.500	13	10	-	-
10	•	•	•	•	0.500	13	10	-	-
11	•	•	•	•	0.500	13	10	-	-
12	•	•	•	•	0.500	13	10	-	-
13	•	•	•	•	0.500	13	-	-	-
14		•	•	•	0.577	8	6	-	-
15		•	•	•	0.577	8	6	-	-
16		•	•	•	0.577	8	-	-	-
17		•			0.000	20	-	-	-
18		•			0.000	20	-	-	-
19		•			0.000	20	-	-	-
20		•			0.000	20	-	-	-
21		•			0.000	20	-	-	-
23			•	•	0.707	4	3	3	1
24			•	•	0.707	4	3	3	-
25			•	•	0.707	4	-	-	-
26			•	•	0.707	4	3	3	-
27			•		0.000	20	-	-	-
28			•		0.000	20	-	-	-
32		•	•	•	0.577	8	6	4	-

```
1         class BST {
2           Node root; int size;
3           static class Node {
4             int value; Node left, right;
5             Node(int value) {
6 *,cd,d      this.value = value;
7             }
8           }
9           void insert(int value) {
10 *,cd       if (root == null) {
11 *,cd         root = new Node(value);
12 *,cd,d       size++;
13              return;
14 *,cd       } Node current = root;
15 *,cd       while (true) {
16              if (value < current.value) {
17                if (current.left == null) {
18                  current.left = new Node(value);
19                  break;
20                } else {
21                  current = current.left;
22                }
23 V1        /* } else {  */
23 V2,*,cd     } else if (value >= current.value) {
24 *,cd         if (current.right == null) {
25 *              current.right = new Node(value);
26 *,cd           break;
27              } else {
28                current = current.right;
29              }
30 V1        /* } */
30 V2           } else { return; }
31            }
32 *,cd,d     size++;
33          }
34        }
```

Fig. 1. BST faulty insert method.

III. APPROACH

Our goal is to improve Tarantula's inspection cost. To achieve this goal we exploit the fact that Tarantula can report uninteresting statements at high ranks as it cannot recognize statements that although covered are irrelevant to provoke failure. We propose techniques that inform Tarantula which statements out of those covered by failing test runs are relevant to the failures, and hence should be considered for ranking. They vary in the criteria used for pruning. Note that the pruning in itself does not imply reduction of inspection cost as each ignored statement can appear above or below the rank of the faulty statement that Tarantula originally reports.

Fault-Localization Using Dynamic Slicing. First technique prunes from Tarantula's ranking all the statements that do *not* appear in the dynamic slice [4] associated with the incorrect value passed to an assertion that fails. Such statements do not influence the value that causes a test to fail. **Example:** In the running example, a total of 10 statements (out of 20 covered) have been pruned with the slice associated with the value of `bst.size`. All statements that are removed but 1 have score smaller than that of the faulty statement; statement 25 has the same score as the faulty statement (0.707). With the removal of this statement, the cost reduces to 3. Column "$R1$" in Table I shows the ranking obtained with this technique.

Fault-Localization Using Change-Impact Analysis. Second technique builds on the first to further reduce inspection cost. It looks for a set of statements that not only influence the computation of the values that leads to failure but also have been impacted by changes. The technique ignores statements from Tarantula's ranking which are not in this *impact set*. To obtain this set, we use change-impact analysis at the *statement-level*; this is implemented on top of a dynamic slicer. This technique does *not* assume that the fault is among changes. In a scenario where a benign change activates a residual fault [5] resulting in a test failure, all statements impacted by change would appear in the set. This impact set includes all statements that have been involved in the construction of values that: (a) have been dynamically influenced by changes (backwards) *and* (b) influence the value of interest (forward). **Example:** Column "$R2$" in Table I shows the ranking obtained with this technique. It did not improve over the first technique. It is important to note, however, that effectiveness of each technique depends on a number of factors not explored in this simple example, e.g., the amount of changes and how they propagate to failure.

Fault-Localization Using Slicing and Change Filtering. We also evaluated the technique that filters from Tarantula's ranking those lines that influence the computation of the value that leads to failure and also have been changed. In contrast to the previous technique, it makes the assumption, often used to evaluate testing and debugging techniques [6], that the error is among the changed statements: in principle, it may result in loss of residual (latent) errors introduced in older evolution cycles and not detected with the test suite [5]. Column "$R3$" in Table I shows the ranking obtained with this technique. **Example:** In our example, the use of this technique reduced inspection cost to 1 statement.

IV. EVALUATION

This section presents and discusses experimental results.

Subjects. We used 50 subjects in total. The subjects were obtained from 2 applications from the Software-artifact Infrastructure Repository (SIR) [6] (jtopas[1-6] and ant[1-4]) and 6 applications in the Siemens benchmark [7] (print_tokens[1-5], print_tokens2[1-7], tot_info[1-10], tcas [1-9], schedule[1-6], and schedule2[1-3]). We chose these applications since they have been used before in other studies using Tarantula.

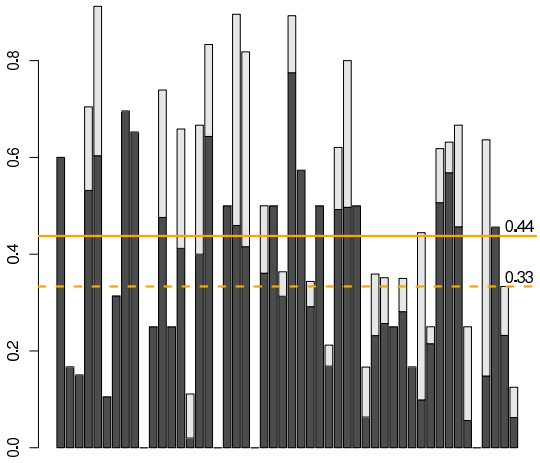

Fig. 2. Percentage reduction of inspection cost using dynamic slice(s).

A. Fault-Localization Using Dynamic Slicing

Setup: We considered two scenarios of use of Tarantula with dynamic slicing. The first scenario assumes the user will not tolerate a possibly long wait for results. To simulate this scenario we make the analysis to use the dynamic slice from only one failing test. The second scenario we considered assumes that the user is willing to tolerate a longer delay. To simulate this scenario we make the analysis to use the combination of dynamic slices from all failing tests. The results of these experiments are measured in terms of improvement in estimated inspection cost. We measure improvement with the size of the intersection between two sets. The first set includes all statements considered to compute Tarantula's original cost while the second set includes sliced statements. When considering multiple failing tests, the second set is the intersection across all slice sets.

Results: Figure 2 summarizes the results of the experiments. Each bar in the figure corresponds to one subject. The height of the lower/darker bar indicates the average percentage reduction in inspection cost obtained by using the dynamic slice from a single failing test. It is calculated as $(y - \bar{x})/y$, where y corresponds to Tarantula's original inspection cost and $\bar{x}$ is the arithmetic mean of the improved inspection costs considering the dynamic slices from various choices of failing tests. The dashed horizontal line shows the average over all these bars. The height of the higher/lighter bar indicates the additional improvement in inspection cost that can be obtained by combining the dynamic slices of all the failing tests to improve Tarantula. The continuous horizontal line shows the average over all these bars.

Inspection Cost Reduction: Results indicate that using the dynamic slicing reduced inspection cost for 46 of the 50 subjects (92%). The only 4 subjects for which inspection cost did not reduce were those for which Tarantula's associated cost was already very low (<4LOC). Inspection reduced 33%, on average, when considering one failing test. Using slices from all failing tests reduced the inspection cost further, on average by 44%.

Absolute reduction: We observed that the absolute reduction in inspection cost is more significant when Tarantula's original cost is high. For example, considering slices from individual failing tests, the inspection cost for print_tokens(4) reduced from 211 to 124 statements, on average. With the combination of all slices of failing tests, inspection cost reduced to 72 statements.

Larger Apps and Slices: We observed that for larger applications the size of dynamic slices was often higher. However, this did not preclude reductions in inspection cost. For large subjects, test execution often covers a large number of statements and Tarantula's inspection cost also tends to be larger. For example, even though jtopas(5) produced relatively large slices (>300 statements on average), reduction in inspection cost was significant. Considering slices from individual failing tests, Tarantula's original inspection cost of 57LOC was reduced by 60.3%, on average. Considering the combination of slices from all failing tests, the inspection cost was reduced by 91.2%, the largest reduction across all subjects (highest bar in Figure 2).

Variance in Reduction: For a given subject we observed that the size of dynamic slices obtained from different failing tests can vary significantly even though the failures are caused by the same fault. This occurs due to the variation in the dynamic dependencies observed across various test executions. The combination of dynamic slices from all failing tests takes advantage of this to further reduce inspection cost. In the worst case, the improved inspection cost is equivalent to the minimum inspection cost in the distribution.

Additional Computational Cost: For all subjects considered except ant, execution of one test in slicing mode terminated very quickly (<2s) and consumed low amount of memory (<50MB). For ant, the execution of one test in slicing mode took, on average, 9s (median=6s) and consumed 57MB (median=56MB) compared to 2s (median=2s) and 29MB (median=31MB) in the standard, non-slicing mode.

B. Fault-Localization Using Change-Impact Analysis

Setup: We evaluated the effect of using change-impact analysis across two different evolution periods. First, we considered longer evolution periods during which subjects were changed significantly. In this scenario, we used the period between two releases of SIR subjects. Second, we considered shorter evolution periods during which subjects were changed very little. For these experiments we used the Siemens subjects. Since the Siemens benchmark does not contain multiple releases, we simulated evolution by asking an external developer to refactor the subject code. For both scenarios of evolution we considered the use of one or all failing tests to compute slices and measured the impact on inspection cost and computational cost. We compare results of the change-impact analysis combination with dynamic slicing combination. Change-impact analysis requires information of change across two code version. To obtain the diff we used the DiffJ open-source tool (see `http://www.incava.org`). DiffJ compares the abstract

syntax trees of compilation units as opposed to plain text which improves the accuracy of the comparison.

Results for longer evolution periods: In this scenario, we considered the period between two releases of a SIR subject.

Inspection Cost Reduction: Compared to dynamic slicing, change-impact analysis reduced the size the slices for 6 out of 10 cases, with an average reduction of 58%. Unfortunately, the use of change-impact analysis further reduced inspection cost (compared to dynamic slicing) only for ant(1), and the reduction was very small (2 statements). For the other 5 subjects, dynamic slicing combination already performed well.

Additional Computational Cost Reduction: For jtopas, no improvement on computational cost was noticed: execution in slicing mode already finishes fast and consumes low memory. For ant subjects, where computational cost is higher, the average reduction is significant: from 57MB in full slicing mode to 35MB in change-impact mode and from 9s in full slicing mode to 4s in change-impact mode.

Results for shorter evolution periods: In this experiment, we used the Siemens subjects with simulated evolutions. The simulated evolutions were performed by an external developer by refactoring 5, 10, and 15% of the original code. We validated that the changes were indeed refactorings by using the subject tests. In these experiments we considered only subjects for which the original inspection cost was more than 10 statements. Table II compares the improved inspection costs for these subjects using dynamic slicing vs. change-impact analysis. Column "T+DS" shows the improved inspection cost achieved by using dynamic slicing and the other columns show the improved inspection cost achieved by using change-impact analysis with 5 and 10% changed code. We omit the setup with 15% changed code since there was no improvement compared to 10%. Numbers outside parentheses (respectively, inside) correspond to the average of inspection costs when using the slice from one failing test (respectively, all tests).

Inspection Cost Reduction (one test): Considering the slice from a single failing test, the absolute reduction for print_tokens2 and schedule was very small. We marked the rows for these subjects in grey color. For print_tokens, tot_info, tcas, and schedule2 reduction was more significant. For print_tokens we observed that, as we move from 5 to 10% of changed code, the results of using change-impact analysis improve. This occurs due to a reduction in total number of statements in the program due to refactoring.

Inspection Cost Reduction (multiple tests): Considering the combination of slices from all failing tests, the reduction was significant for all cases with the exception of print_tokens2 and schedule. The best results appeared in print_tokens and schedule2. For example, on 5% of changes, average cost for the schedule2 case reduced from 24.67 (for Tarantula + Slicing) to 6.67 (for Tarantula + Change-impact).

Additional Computational Cost: As for computational cost, we did not observe significant improvement compared to the cost of using dynamic slicing since the cost was already small for Siemens subjects.

TABLE II
IMPROVING TARANTULA WITH DYNAMIC SLICING (T+DS) VS. IMPROVING TARANTULA WITH CHANGE-IMPACT ANALYSIS (T+CI).

name	T+DS	T+CI (5%)	T+CI (10%)
print_tokens	117.32 (63.50)	103.99 (42.00)	101.96 (42.00)
print_tokens2	16.19 (3.50)	15.79 (3.50)	15.79 (3.50)
tot_info	24.61 (21.50)	10.20 (8.40)	14.34 (12.40)
tcas	30.30 (27.50)	10.59 (8.50)	12.97 (10.50)
schedule	26.19 (23.50)	21.19 (18.50)	23.69 (21.00)
schedule2	34.64 (24.67)	23.48 (6.67)	28.08 (17.00)

C. Fault-Localization Using Slicing and Change Filtering

We also evaluated the use of change filtering on top of dynamic slicing. We compared the effect of using this lossy approach with the conservative alternative of using change-impact analysis. The benefit of change filtering for SIR subjects was marginal. There was a small improvement only for ant(1) when inspection cost reduced from 33 to 8 statements. For Siemens subjects, which contain a smaller number of changes, the reduction was significant. Considering both scenarios of 5 and 10% of changed code and the use of a slice from one failing test, inspection cost *was below 7 statements* for all subjects. As the costs were already very low the impact of combining slices from all failing tests was not as significant as in the other scenarios.

V. CONCLUSIONS

This paper shows that dynamic slicing can be effective to improve performance of spectrum-based fault-localization. When change information is available, improvement can be further magnified. We show the effect of a conservative adaptation of the slicer that takes code changes into account and an unsafe adaptation that just filters the changed statements from the reported slice sets. Combination of dynamic slicing with other fault-localization techniques have been proposed in the past [8], [9]. To the best of our knowledge this is the first combination that uses Tarantula and considers change information as another dimension of improvement.

Acknowledgments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES: `www.ines.org.br`).

REFERENCES

[1] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*, 2002.
[2] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *ICSE*, 2009.
[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *MUTATION*, 2007.
[4] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, 1995.
[5] P. Nagahawatte and H. Do, "The effectiveness of regression testing techniques in reducing the occurrence of residual defects," in *ICST*, 2010.
[6] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques," *Springer ESE*, vol. 10, 2005.
[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow and control flow-based test adequacy criteria," in *ICSE*, 1994.
[8] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," in *PLDI*, 2006, pp. 169–180.
[9] F. Wotawa, "Fault localization based on dynamic slicing and hitting-set computation," in *QSIC*, 2010, pp. 161–170.