Technique Report: A Tree-based Approach to Identifying and Explaining Regression Bug

Yun Lin

1 Introduction

In this report, we introduce an approach to identifying regression bug by matching program traces. Such an approach is originally developed for evaluating our feedback-based debugging work [4] in which we aims to categorize the steps on a buggy execution trace into either correct step, wrong-variable-value step, or wrong-path step.

Original Scenario Given a method, m, and one of its passed test cases, t, we mutate a line of code in m to generate m' to kill t. Therefore, running the test case t on m and m' generates two traces $trace_m$ and $trace_{m'}$. Clearly, $trace_{m'}$ is a "wrong" trace. In the work of [4], we need to further know (1) which steps in $trace_{m'}$ is wrong and (2) how they are wrong. With our proposed trace matching approach, we can match the steps in $trace_m$ and $trace_{m'}$ so that we know each step in $trace_{m'}$ is either correct, has wrong variable value, or should be never executed (wrong-path). With the simulated feedbacks (i.e., correct, wrong-variable-value, or wrong-path) on steps of $trace_{m'}$, our debugging work [4] can iteratively recommend the suspicious steps on $trace_{m'}$ based on control and data flow. If we can finally recommend the step running into the mutated line in m', our debugging work is proved to be effective in such a case.

Extension When we regard a mutation as the introduction (or injection) of a bug into original program, the above scenario can be interpreted into the scenario of locating a regression bug. Compared to existing fault localization work to report the bug in terms of source code line [1, 2, 5, 6, 7], this approach support:

- **Execution Replay**: With the trace, the developer is able to replay the execution and check when the bug happens during the execution. Suppose the bug lies in certain line inside a loop, and the defect only happens after several loop iterations, the developer can understand how the bug happens in grain of *step* instead of *line of code*.
- **Causality Explanation**: We can generate a causality chain to explain how the defect is propagated to the revealed fault.

In the following, we introduce the approach details 2 and the follow-up research questions to be addressed for its application on identifying regression bug **??**.

2 Approach

In this section, we start with modelling an execution trace. Then, we introduce the solution. This section ends with an open research question.

2.1 Trace Model

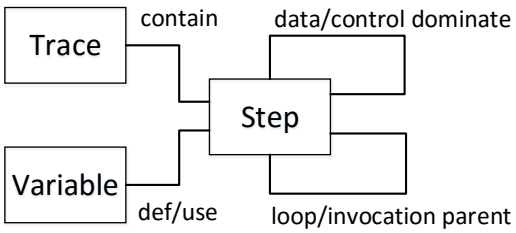

Figure 1: Trace Model

Figure 1 shows the meta model we use to describe a trace. In Figure 1, a **trace** contains a number of **step**s. Each step corresponds to an executed source code line, which can *define* (i.e., write) or *use* (i.e., read) some **variable**s. Given a variable *var*, if *var* is defined by step s_1 while used by step s_2, then we say that step s_1 *data dominate*s s_2 on *var*. In addition, the variable *var* is called as the **attributed variable** of the data dominance relation. On the other hand, given a conditional statement *con_stat*, if *con_stat* is executed in step s_1 while the evaluation value of *con_stat* (i.e., true or false) decides the execution of step s_2, then we say that step s_1 *control dominate*s s_2. Given two steps s_1 and s_2, if s_1 control or data dominates $s2$, then we say that s_1 is control or data **dominator** of s_2, and s_2 is the control or data **daminatee** of s_1. In addition, we say that s_1 is the **contextual parent** of s_2 if either of following conditions happens:

- **Loop Parent**: s_1 starts a loop iteration l, and s_2 is executed in l but not in any nested loop iteration or method invocation in l.
- **Invocation Parent**: s_1 starts a method invocation m and s_2 is executed in m but not in any nested method invocation or loop iteration in m.

The contextual parent-child relation can organize our trace into a **step tree**, in which the root is the entry method and the leaves are the steps invoking no method or starting no loop iteration. Given a step, we define its layer on step tree as its **abstract level**. By default, the abstract level of the entry method is 0.

2.2 Solution

Given a trace for a buggy program, its steps must fall into either of the following three cases: (1) correct step, (2) steps with variable of wrong value, and (2) steps which should not be executed. If we are aware of the above information, we can debug on the trace in a backward manner, i.e., we trace back to and report the very first incorrect step by data and control dominance relation. We propose trace match approach to this end.

In our approach, we match step tree represented by contextual parent-child relation instead of a step sequence in trace. The benefits of tree-matching lie in that:

- **Keep Boundary** The contextual parent-child relation confines syntactic boundary so that our matching results will never across the iterations or method invocations.
- **Improve Accuracy** The number of direct children of a contextual parent is much smaller than that of a trace. Matching in smaller range greatly improves the accuracy of applying existing matching algorithm such as LCS [3].

In general, we adopt the matching procedure in a top-down manner as showed in Algorithm 1. Algorithm 1 takes as input the roots of an original tree and a buggy tree and return a set of pairs representing matching results. A pair $pair =< o, b >$ is represented by a node in original tree $pair.o$ and a node in buggy tree $pair.b$. Either $pair.o$ or $pair.b$ can be ϵ, which means that a step cannot be matched.

Algorithm 1 tree_match

Require: root node of original tree $root_o$, root node of buggy tree $root_b$
Ensure: a set of match pair $list_{pair}$

```
1:  list_pair ← ∅;
2:  children_o ← retrieve direct children list of root_o;
3:  children_b ← retrieve direct children list of root_b;
4:  pairs ← LCS(children_o, children_b);
5:  for each pair pair ∈ pairs do
6:      list_pair ← list_pair ⋃ {pair};
7:      if ¬(ϵ ∈ pair) then
8:          sublist_pair ← tree_match(pair.o, pair.b);
9:          list_pair ← list_pair ⋃ sublist_pair;
10:     end if
11: end for
12: return list_pair;
```

Given two roots, in order to take the step execution order into consideration, we match their direct children by a dynamic programming algorithm such as LCS [3] (line 4). The trace step similarity in LCS is based on some heuristics such as source code line and visited variable names. If two nodes (steps) are matched, we further proceed to match their children with the same tree-matching procedure (line 7–9). Such a matching process is conducted until all the nodes on trees have been traversed.

Pair Category In the perspective of the buggy trace, we categorize the pairs into three categories:

- **Pure Match**: A step in buggy trace can be matched to a step in original trace, and all their reachable variables along with their values are exactly the same.
- **Impure Match**: A step in buggy trace can be matched to a step in original trace, but the values of some of their reachable variables are different.
- **Missing Match**: A step in buggy trace cannot be matched to any step in original trace.

Clearly, we can derive the correct step, wrong-variable-value step, and wrong-path step from the above match category correspondingly. Noteworthy, for impure match, we may further match the reachable variables in original step and buggy step. Since variables can be represented by a graph, we leverage a similar algorithm like Algorithm 1 to match variables so that we can identify which variable is of wrong value.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, 2009.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780 – 1792, 2009.

[3] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343.

[4] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong. Feedback-based debugging. In *Proceedings of the 39th International Conference on Software Engineering*, 2017. Accepted.

[5] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 432–449, 1997.

[6] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of 31st International Conference on Software Engineering*, pages 56–66, 2009.

[7] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, pages 45–55, 2009.