Partial Specifications for Program Repair

Linsey Kitt
Dept. of Computer Science
Iowa State University
Ames, IA, USA
ljkitt@iastate.edu

Myra B. Cohen
Dept. of Computer Science
Iowa State University
Ames, IA, USA
mcohen@iastate.edu

***Abstract*—In this paper we argue for using many partial test suites instead of one full test suite during program repair. This may provide a pool of simpler, yet correct patches, addressing both the overfitting and poor repair quality problem. To support this idea, we present some insight obtained running APR partial test suites on the well studied triangle program.**

***Index Terms*—program repair, overfitting, test suites**

I. Introduction

Automated program repair (APR) has become a common approach to assist developers by finding potential program patches [1]. No longer an academic exercise, APR first localizes, and via a series of automated edits, transforms a failing program to one that passes a given test suite, or is, in essence, deemed correct. While many advances have been made in APR, key challenges remain including (1) having a sufficient set of specifications [2] and (2) ensuring a quality test suite based on those specifications [3], [4]. Incomplete or poorly designed test suites may cause low quality patches. For instance, code which is not covered by test suites may be erroneously deleted or changed, leading to overfitting [5]. Others have also noted some test suites lead to overly complex patches. To date there has been a lot of research trying to build better test suites, and as such, some have argued manual tests are better than automated tests [3], [5].

In this paper we argue for a new paradigm. Rather than focusing on individual test suite quality or specialized selection techniques (such as lexicase selection), we propose using a lightweight, weaker, but distributed approach: run many parallel APR runs, each with different partial test suites, informed by input specifications. This will lead to many overfit patches, but it also has potential to provide a *pool* of correct patches that still satisfy the full specifications and are simpler in nature. We believe this approach will work well on programs with test suites which have been designed using functional decomposition.

Many existing APR tools use some form of partial test suites, e.g. randomly selected subsets for different program generations, or on different chromosomes for selection. Some also post-process tests to reduce complexity of patches at the end [6]. We instead partition tests based on program specification and use a single set of test cases for all generations during an individual repair.

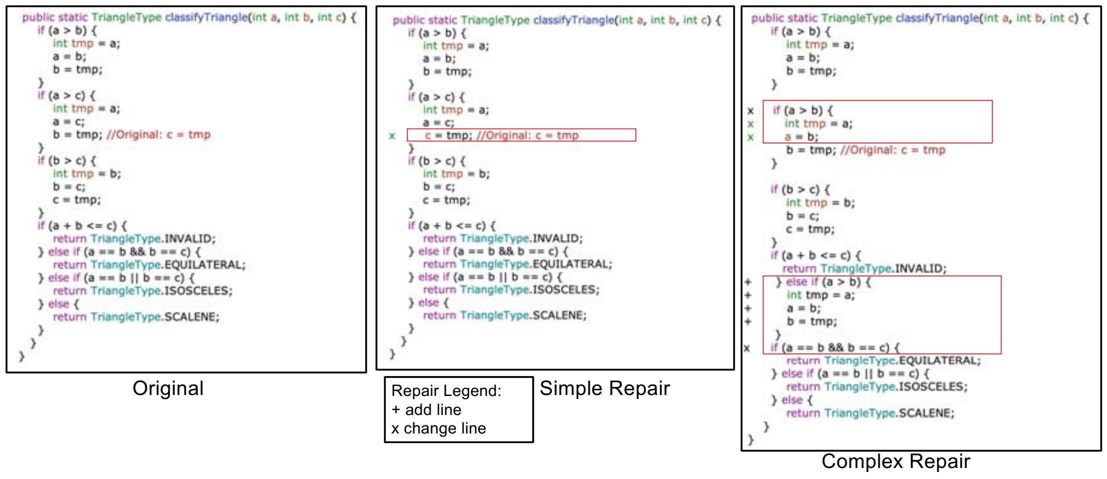
Figure 1 shows an example fault in a triangle program (left). This is a well studied program for APR. Recent work by Langdon et al. [7] explored the search space of this program and demonstrated it has an uneven distribution of faults. This suggests tests may fall into partitions that can be leveraged for our distributed approach.

The fault shown is a single mutation of the program; however, when we examine automated repairs, we see a range of fixes from simple to complex. For our purposes, simple means the lines of the program remain as small as (or smaller than) the original program. We also assert these must be correct repairs: ones that would pass a high quality test suite. If we look at the repair in the middle, it simply reverts the mutation. This correct version of triangle has the same size as the original program, i.e. it is simple. This repair was found repeatedly, but only using partial test suites. If we look at the program on the right, the original `if` statement was modified and an additional `if` statement was added. This repair is correct, but it is complex; we found only complex repairs when using the full test suite. Our observation is *fewer test cases lead to simpler repairs.* These repairs may also be incorrect; however, if we run different *partial test suites* in parallel we may also obtain correct repairs.

II. Demonstration

We performed a small study to explore the impact of partial test suites on the success and complexity of repairs. We use the example triangle program and test cases provided with the PyGGI 2.0 repair framework [8]. This program has a single mutation in the same `if` statement (if (a > c)) as the mutant in Figure 1, changing tmp=a to tmp=b. The test suite has 28 test cases, grouped by program specifications. There are 13 invalid, 3 equilateral, 6 isosceles and 6 scalene triangle tests.

We first ran PyGGI using tabu search with an unlimited number of iterations and repeated this 15 times. We then systematically removed test cases for each requirement. For instance, we removed the equilateral test checks. Then we removed the isosceles and then removed both of these. We re-ran the repair with the same settings, 15 times for each experiment. All experiments are run on a laptop



```
public static TriangleType classifyTriangle(int a, int b, int c) {
    if (a > b) {
        int tmp = a;
        a = b;
        b = tmp;
    }
    if (a > c) {
        int tmp = a;
        a = c;
        b = tmp; //Original: c = tmp
    }
    if (b > c) {
        int tmp = b;
        b = c;
        c = tmp;
    }
    if (a + b <= c) {
        return TriangleType.INVALID;
    } else if (a == b && b == c) {
        return TriangleType.EQUILATERAL;
    } else if (a == b || b == c) {
        return TriangleType.ISOSCELES;
    } else {
        return TriangleType.SCALENE;
    }
  }
}
```

```
  public static TriangleType classifyTriangle(int a, int b, int c) {
      if (a > b) {
          int tmp = a;
          a = b;
          b = tmp;
      }
      if (a > c) {
          int tmp = a;
          a = c;
x         c = tmp; //Original: c = tmp
      }
      if (b > c) {
          int tmp = b;
          b = c;
          c = tmp;
      }
      if (a + b <= c) {
          return TriangleType.INVALID;
      } else if (a == b && b == c) {
          return TriangleType.EQUILATERAL;
      } else if (a == b || b == c) {
          return TriangleType.ISOSCELES;
      } else {
          return TriangleType.SCALENE;
      }
    }
  }
```

```
  public static TriangleType classifyTriangle(int a, int b, int c) {
      if (a > b) {
          int tmp = a;
          a = b;
          b = tmp;
      }

x     if (a > b) {
x         int tmp = a;
x         a = b;
          b = tmp; //Original: c = tmp
      }

      if (b > c) {
          int tmp = b;
          b = c;
          c = tmp;
      }
      if (a + b <= c) {
         return TriangleType.INVALID;
+      } else if (a > b) {
+         int tmp = a;
+         a = b;
+         b = tmp;
       }
x     if (a == b && b == c) {
          return TriangleType.EQUILATERAL;
      } else if (a == b || b == c) {
          return TriangleType.ISOSCELES;
      } else {
          return TriangleType.SCALENE;
      }
    }
  }
```

Fig. 1. Example fault in a triangle program (left). We show a simple correct repair in the middle and a more complex repair on the right.

TABLE I
Count of successful and simple repairs by test suite. All indicates all tests are used. The other columns indicate which tests are removed. IS=isosceles, IN=invalid E=equilateral, S=scalene.

	All	**IS**	**IS,S**	**IS,S,E**	**IS,E**	**IS,E,IN**	**IS,IN**	**S**	**IN**	**IN,E**	**S,E**	**E**
Success	15	1	1	3	0	1	1	15	15	9	15	15
Simple	3	1	0	2	0	1	1	8	10	6	7	3
Avg. Iter	765.5	132.3	175.7	83.9	117.1	202.5	209.7	1214.8	1351.4	902.1	1845.7	810.2
Median Iter	746.0	72.0	137.0	37.0	86.0	134.0	80.0	1042.0	1521.0	992.0	1589.0	570.0

running Windows 10 inside of an Ubuntu 18.04 virtual machine with 2GB of memory.

We recorded the iteration where the repair is found and performed two checks. First, we tested all repairs against the full test suite. If they passed, we manually examined them for correctness. We then classified them into two categories. Those which contained the same (or fewer) lines of code as the smallest patch we found were marked simple. All others were considered complex. We saw common patterns repeat which simplified this process.

Table I shows the results of this study. The columns represent the test suites with *All* using all 28 test cases. The other columns show which tests are removed. For instance, in the second column (IS) isosceles tests are removed, and in the third column (IS,S) both isosceles and scalene tests are removed.

The first row shows the number (out of 15) trials with a correct repair. For example, the All column finds 15 and IS only 1. The next row lists the number of simple repairs. The All column finds 3 and the IS, 1. The next two rows show the average and median number of iterations which were required to find a repair (irrespective of its correctness). Most of the partial test suites find at least one repair (all except IS,E). The smaller test suites such as S, IN, and E find a large number of correct patches, and more of these are simple. This suggests an interplay between relaxed requirements during repair and the algorithm's ability to find simple patches.

III. Conclusions

In this paper we argue for a new perspective on APR test suite quality, based on partial specifications. We see an opportunity to repair in a distributed fashion, and use a pool of (potentially overfit) repairs to choose those which satisfy the full test suite. A full evaluation is future work.

Acknowledgment

This work was funded in part by National Science Foundation Grant CCF-1901543.

References

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE TSE*, vol. 45, no. 1, pp. 34–67, 2019.
[2] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *CACM*, vol. 62, no. 12, p. 56–65, Nov. 2019.
[3] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of automated program repair on real-world defects," *IEEE TSE*, 2020.
[4] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *ESEC/FSE*, 2017, p. 831–841.
[5] M. Lim, G. Guizzo, and J. Petke, "Impact of test suite coverage on overfitting in genetic improvement of software," in *SSBSE*, ser. LNCS, vol. 12420, 2020, pp. 188–203.
[6] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE TSE*, vol. 38, no. 1, pp. 54–72, 2012.
[7] W. B. Langdon, N. Veerapen, and G. Ochoa, "Visualising the search landscape of the triangle program," in *EuroGP*, ser. LNCS, vol. 10196, 2017, pp. 96–113.
[8] G. An, A. Blot, J. Petke, and S. Yoo, "PyGGI 2.0: Language independent genetic improvement framework," in *ESEC/FSE*, 2019, pp. 1100–1104.