Regression Aware Debugging for Mobile Applications

Rohan Bavishi Awanish Pandey Subhajit Roy

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, India.
{rbavishi, awpandey, subhajit}@iitk.ac.in

Abstract

Regression-aware fault localization attempts to rank suspicious statements in a manner such that potential regression inducing suggestions are ranked low. The algorithm extracts the proof of correctness of all the correct executions in the form of Craig Interpolants over the successful execution traces. It, then, labels a program location suspicious if it can find a possible value for the assignment that can allow the hitherto failing execution to produce the expected output. However, any such value that does not satisfy the proof constraints of the passing tests are penalized in terms of their ranking. In this article, we sketch the regression-aware fault localization algorithm and motivate its potential application in debugging mobile applications.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Debugging, fault localization

1. Introduction

This is an era of smart devices—from smart-watches to cellphones and tablets, mobile applications have become an integral part of our lives. The amount of software developed in this domain necessitates support for effective testing and debugging.

One potent debugging technique is *fault localization*: given a failing test, a fault localization algorithm suggests potential program statements that can be "repaired" to correct the failing execution; we refer to such statements as *suspicious* statements. However, many such suggestion may induce regression—though there exist possible alterations to these statements that can "repair" the failing execution, but any such modification could lead to the failure of a hitherto passing test-case. *Regression-aware fault localization* [1] attempts to rank the suspicious statements such that potential regression inducing suggestions are ranked low.

We illustrate our algorithm using the code snippet shown in Figure 1. The program is an abstraction of a graphical user-interface (GUI) based event-driven program described in a C-like language. The program encodes the essence of a sprite-based game: the `handle_user_input()` method detects the keypress events and moves the player's sprite by translating its bounding box. Every time the player moves, the handler also detects if it collides with the stationary objects around it (for simplicity, we show only a single object) using the `detect_collision()` routine; the boolean variable `collision_detected` trips to true any time a collision happens. The `main()` procedure installs the keypress event handler using `Handle_User_Input()` such that the `handle_user_input()` routine is invoked at every keypress. The `Check_Desired_Collision_State()` routine provides the correctness specification by asserting if the state of the `collision_detected` variable is indeed correct.

The tests for such a routine would include a sequence of keypress events and the expected outcome is the state of the `collision_detected` flag. Such tests can be produced by human testers or automatically generated from tools like the UI/Application Exerciser Monkey[3]. Table 1 shows a set of tests (column 2), their expected outcome (column 3 labelled 'Collision Occurs'), the actual outcome from the program captured as the state of the variable `collision_detected` (column 4 labelled 'Collision Detected') and the test outcome (column 5) for the given program.

The program listed in Figure 1 is faulty and the bug lies in lines 21 and 22: it fails when the player's bounding box exactly overlaps with that of the object (the bounding boxes of the player and the object have the same dimensions). The correct expression is provided as a comment on line 19. This is precisely the reason why the fifth test fails: the player moves right once, moves up twice, and then moves right again; this aligns the player exactly with the object, but the collision is still not detected (due to the bug described above). In the four passing tests, the player is either moving away from the object or its bounding box intersects partially with that of the object.

```
1  #define OBJ_X1 20
2  #define OBJ_X2 25
3  #define OBJ_Y1 4
4  #define OBJ_Y2 9
5  // Horizontal & Vertical Distance
6  // covered by key-presses
7  #define HOR_INC 10
8  #define VER_INC 2
9
10 // 2-D Bounding Box of the Player
11 int p_x1, p_x2, p_y1, p_y2;
12 bool collision_detected;
13
14 bool x_intersect_check() {
15      return (p_x1 <= OBJ_X1 && OBJ_X1 < p_x2) ||
16             (p_x1 < OBJ_X2 && OBJ_X2 <= p_x2);
17 }
18
19 bool y_intersect_check() {
20  BUG FIX : (p_y1 <= OBJ_Y1 && OBJ_Y1 < p_y2) ||
21            (p_y1 < OBJ_Y2 && OBJ_Y2 <= p_y2);
22 ##  return (p_y1 < OBJ_Y1 && OBJ_Y1 < p_y2) ||
23            (p_y1 < OBJ_Y2 && OBJ_Y2 < p_y2); ##
24 }
25
26 void detect_collision() {
27     collision_detected = (collision_detected ||
28                           ## (x_intersect_check() &&
29                               y_intersect_check()) ##
30                          );
31 }
32
33 void handle_user_input(int keypress) {
34     if (keypress == UP)
35      ## { p_y1 += VER_INC; p_y2 += VER_INC; } ##
36     else if (keypress == RIGHT)
37         { p_x1 += HOR_INC; p_x2 += HOR_INC; }
38     else if (keypress == DOWN)
39          { p_y1 -= VER_INC; p_y2 -= VER_INC; }
40     else if (keypress == LEFT)
41          { p_x1 -= HOR_INC; p_x2 -= HOR_INC; }
42
43     // Check if box of player intersects with that
44     // of the object because of previous step
45     detect_collision();
46 }
47
48 int main() {
49     // Initialization of variables
50     p_x1 = p_y1 = 0; p_x2 = p_y2 = 5;
51     collision_detected = false;
52
54     Handle_User_Inputs();
55     // Check if the given sequence of user inputs
56     // causes a collision at any step
57     Check_Desired_Collision_State();
58 }
```

Figure 1: Code excerpt from a sprite-based game handling 2-dimensional collision detection among rectangular boxes

Our regression-aware fault localizer starts off by running the tests and recording a trace for each successful execution (corresponding to the passing tests 1 to 4) as a sequence of dynamic statements executed. It, then, extracts Craig Interpolants[2] from the proofs of correctness of the trace formula of all these passing tests (tests 1 to 4). Next,

Test ID	User Inputs	Collision Occurs	Collision Detected	Test Outcome
1	[⇐, ⇐, ⇐, ⇐, ⇐]	False	False	Passed
2	[⇒, ⇑, ⇒, ⇒, ⇒]	True	True	Passed
3	[⇒, ⇒, ⇑, ⇑, ⇒]	True	True	Passed
4	[⇑, ⇑, ⇑, ⇑, ⇑]	False	False	Passed
5	[⇒, ⇑, ⇑, ⇒, ⇒]	True	False	Failed

Table 1: Test-suite and respective outcomes (for Figure 1)

for the failing test (test 5), it uses the CBMC bounded model-checker [4] to search for program locations where it is possible to replace an expression by some value that can rectify the failing test. It also checks whether this value conforms to the constraints imposed by the interpolants obtained from the correctness proofs of the passing tests; the more proof terms it violates, the more likely it is that a change to this location will cause regression.

The overall ranking of a candidate location is composed of two metrics: the number of program locations that need to be modified, and the number of proof terms it violates. Based on this, the localizer outputs a ranked set of locations with regression-inducing locations ranked lower.

The statements highlighted in gray (in Figure 1) are the locations suggested when information from the passing tests is not used. The statements enclosed within ## are the statements that are ranked as the most suspicious locations by our regression-aware localizer.

Our localizer is able to rank the correct bug location among one of its three most suspicious locations, while giving a lower score to the other superfluous candidates. For example, it ranks the program location at line 15 low as it is able to correctly reason that the return statement for the `x_intersect_check()` function (at line 15) should not be modified. This location is indeed regression-inducing as a change that could rectify the failing test may break Tests 1 and 3. Thus, regression-awareness improves the quality of localization by reducing the number of locations that a developer would need to inspect while debugging the program.

References

[1] Rohan Bavishi, Awanish Pandey, and Subhajit Roy. To Be Precise: Regression Aware Debugging. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, 2016.

[2] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, 1957.

[3] UI/Application Exerciser Monkey. `https://developer.android.com/studio/test/monkey.html`. Accessed on 20th September, 2016.

[4] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.