Reverse Path Exploration for Failure Detection

Thomas Huining Feng (tfeng@eecs.berkeley.edu)

Abstract

Automatic program testing by means of path exploration has been a successful technique for discovering potential failures. However, the cost of a complete test is usually high, while an incomplete test yields false negatives. Algorithms exist trade preciseness for efficiency, resulting in a spectrum of conservatism. Orthogonal to these, effort has been made to avoid exploring "unnecessary" paths, such as multiple paths that fall in the same equivalent class of program behavior.

In this project, a new analysis method is studied that aims to reduce the number of paths without introducing extra conservatism. Unlike many existing program analysis methods, this one tries to explore execution paths in the reverse direction by first focusing on the failure statements, and then the statements (or the conditional tests) above them. This helps to promptly detect failing paths in a lengthy program. There are more applications for this approach, including testing for concurrent programs and incrementally building up tests from small summaries.

1 Motivation

Traditional program testing methods analyze statements of a programs in their sequential order. Each execution path is explored from the start of the program to either the end or a failing statement, which indicates a potential bug that can be reached at run-time. This general approach does not scale for large programs, due to the exponential number of paths that exist in the programs. Many algorithms end up exploring every path even though only few of them can actually lead to a failure. Different conservative strategies have been studied to improve time efficiency, resulting in either extra false positive (detecting failing paths that cannot actually be taken), or extra false negative (missing failing paths that could actually be taken).

An alternative to this is reverse program analysis, which analyzes the programs in their reverse order. This method is based on the observation that failure statements and normal return statements are almost always at the end of a program. If we can first locate those points of interest and focus only on the paths that can lead to failure, then we may be able to avoid exploring paths that are proved safe.

2 Example

Take program `testme.c` as an introductory example to the reverse analysis. (Function `fail` takes a condition and, if it evaluates to `true`, halts the program with a failure.)

```
/* testme.c */
#include <stdio.h>
#include <cute.h>
int dbl(int x) {
    return 2 * x;
}
void f(){
    int x; int y;
    int z = dbl(x);
    if (z == y) {                // IF#1
        if (x != y+10) {         // IF#2
            if (x == 0) {        // IF#3
                printf("fine\n");
            } else {             // ELSE#3
                printf("also fine\n");
            }
        } else {                 // ELSE#2
            y = 0;
            fail(y == 0);
        }
    }
}
```

In this program, function `f` is under test. Unlike the concolic test generation approach which starts executing `f` from the beginning with a random input, we start with the `fail` statement, and generate the first program segment under test (`testme_seg1.c`):

```
/* testme_seg1.c */
#include <stdio.h>
#include <cute.h>
void f() {
    int y;
    y = 0;
    fail(y == 0);
}
```

When we perform concolic analysis on this segment, we detect that the failing condition is "`true`". We then extend this segment with the nearest conditional test and all the unconditional statements above that test. The result is in `testme_seg2.c`.

```
/* testme_seg2.c */
#include <stdio.h>
#include <cute.h>
void f() {
    int x; int y;
    if (x != y+10) {
        if (x == 0) {
            printf("fine\n");
        } else {
            printf("also fine\n");
        }
    } else {
        fail(true);
    }
}
```

We again perform concolic analysis on this segment, considering `x` and `y` as concolic variables. Assume that we arbitrarily let `x=0` and `y=0`. The program terminates at the "fine" branch. We then analyze the path taken in a top-down fashion up to "`(x != y+10)`", because negating this branch takes us to the `fail` statement. In the summary of this analysis, the failing condition is "`(x == y+10) && true`".

`testme_seg3.c` further extends the above segment with one more conditional branch and the statements above it:

```
/* testme_seg3.c */
#include <stdio.h>
#include <cute.h>
void f() {
    int x; int y;
    int z = 2 * x;
    if (z == y) {
        fail((x == y+10) && true);
    }
}
```

Again we consider `x` and `y` as concolic variables and perform concolic testing. This test either takes the true branch of "`(z == y)`" or the false branch. In both cases, we analyze the path up to the branch condition, and summarize the failing condition as "`(2*x == y) && (x == y+10) && true`".

We now reach the head of program `testme.c`. The failing condition is satisfiable (with assignment `x=-10` and `y=-20`), so we know that the original program may fail under some inputs.

3 Assessment

This method aims to avoid exploring unnecessary paths (those that guarantee not to fail) by gradually expanding the backward search from the failing point. Multiple failing points can be handled in a reverse breadth-first search manner.

In `testme.c`, we avoided examining the path `IF#1->IF#2->ELSE#3`. If we used concolic testing, starting with input `x=0` and `y=0`, we would have first explored the path `IF#1->IF#2->IF#3`, and then backtracked to `IF#1->IF#2->ELSE#3` using a constraint solver. This is a redundant test since we already know (with a simple static analyzer) that this path never fails. In practice, in the `IF#2` block there could be much more code that does not fail. Our method avoids testing it.

The performance of our method can be further improved by partial execution. For example, in `testme_seg2.c`, there was no need to fully execute the program. It could be stopped once we took the "then" branch of condition "`(x != y+10)`". With this improvement, one can see that in this example, the amount of work required by this method is roughly equal to examining one full path. [1]

The summaries of program segments (i.e., the conditions in the `fail` function calls) can be reused in multiple analysis. This can be extremely helpful for the testing of concurrent programs, where different interleavings of the threads need to be analyzed. When we examine a sequential interleaving, if the failing condition is satisfied at some point, then we can immediately return the result, and there is no need to analyze further down the execution.

Another interesting application is separate testing. When a component (e.g., some related functions) is developed, it can be checked without considering the overall system. The failing condition is extracted and recorded as an annotation in the component's "interface." The programmer then proceeds to develop other parts of the system. At composition time, a test is performed. It uses the annotations to check whether a component's failing condition is satisfied, and notifies the programmer if so. Changes can be made and test can be re-done. The annotations help to avoid testing the non-failing components over and over again.

[1] In `testme_seg1.c`, we tested the `fail` statement; in `testme_seg2.c`, we tested `ELSE#2`, and the execution stopped right at the branch; in `testme_seg3.c` we tested `IF#1`. The total amount of work is roughly equal to examining path `IF#1->ELSE#2->fail` in full.