Dynamic Program Analysis with Partial Execution and Summary

Thomas Huining Feng
CHESS, UC Berkeley
tfeng@eecs.berkeley.edu

Abstract

Automatic program testing by means of path exploration has been a successful technique for discovering potential bugs. However, the cost incurred by a complete test is usually high, due to the exponential number of paths in the program. Conclusion about a program's failure constraint, made on an incomplete test, yields either false negatives or false positives, or both. Methods exist that trade preciseness for efficiency, resulting in a spectrum of conservatism. Orthogonal to these, effort has been spent on avoiding the exploration of "unnecessary" paths, such as multiple paths that fall in the same equivalent class of program behavior.

In this paper, an innovative program testing method is developed, which aims to reduce the number of explored paths without introducing extra conservatism in the failure constraints. The program under test is first partitioned into segments. The last segment (in terms of program execution) is first tested. A failure constraint is generated for this test. This constraint as a summary is then used to test the next segment above, giving rise to a new constraint for a bigger part of the program. This process repeats until the program starting point is reached. We show in experiment that this approach may greatly reduce the number of paths required to explore in order to obtain a failure constraint. It also helps to extract failure constraint for a program with an unbounded number of paths, e.g., one that has an unbounded loop.

1. Introduction

Traditional program testing methods require to analyze program statements in their sequential order. Each execution path is explored from the start of the program to either the end or a failing statement. The latter indicates a bug. To draw a conclusion on the failure constraint, one records the path constraints (the constraints that characterize the paths traversed) during the test, and then takes the disjunction of those constraints for paths with failure output.

This general approach does not scale for large programs, due to the exponential number of paths that exist in them. All paths are explored even though usually only few of them actually lead to failure. Different conservatism strategies have been studied to improve efficiency, resulting in either false positives (detecting failing paths that cannot actually be taken) or false negatives (missing failing paths that could be taken) in the result. For example, Daikon [5] [2], a tool that helps to dynamically discover program invariants, has the limitation that it can only handle some kinds of invariants that are simple and pre-defined with parameters (e.g., the linear relation between variables). Other kinds of invariants cannot be detected. If Daikon were used to generate failure constraints, the result would then be affected by false negatives.

Predicate-based testing approaches have also been developed, an example of which is Korat [3]. However, Korat relies on the provided pre/post-conditions for methods, which cannot be automatically generated. Analysis of the whole program is then based on those post-conditions as test oracles. Errors made by the programmers who provide the pre/post-conditions also affect the validness of the final judgment.

In this paper, we will introduce a new approach based on partial execution of program segments. It greatly improves the analysis of two kinds of programs, which are identified later. This improvement exhibits itself in a few experimental programs as well as theoretical results.

This work is an extension to JCute [7], a concolic program testing tool for dynamic path exploration. Our testing method takes the source of a Java program as input, and generates a failure constraint that indicates the situation under which the program fails. Java annotations are used to specify failure. Starting from the initial failure annotations (which is usually "true") provided by the programmer, the tester tries to derive constraints for larger and larger parts of the program by repeating a partial execution process, each time exploring all the execution paths that lead to failure.

For simplicity of the discussion, we make an assumption that the programs under test are deterministic and free of side effect. This, however, is not a limitation. Determinism is a common assumption. If it is not satisfied, e.g. because a random number is produced that influences the program execution, a transformation can be applied before the testing, which substitutes the random number generator with a predictable number generator. Concurrency between threads is another source of nondeterminism. To deal with it, a special scheduler can be used that provides predictable threading schedules. Ruled out by the second part of our assumption,

```
public class A {
    void f(boolean op1, boolean op2,
           boolean op3, boolean failure) {
        if (op1) {
            System.out.println("Operation 1");
        }
        if (op2) {
            System.out.println("Operation 2");
        }
        if (op3) {
            System.out.println("Operation 3");
        }
        @Failure("failure") int fail;
    }
}
```

Table 1. A program with independent conditional branches

side effect may make part of a program malfunction. For example, for a program that first opens a file and then reads the content, if the read functionality is executed separately, it may always fail, causing the tester to determine "true" as the failure constraint. Note that when side effect like this happens, the failure constraint in the conclusion is always weaker than the actual failure constraint. In some cases, a weaker constraint is acceptable as a conservative guess. If it is unacceptable, however, one may temporarily continue with the test based on this constraint. Later on obtaining a failure constraint for a bigger part (in this example, the bigger part contains both the open and the read operations), he/she then repeats the first test to make the new constraint no weaker than the actual one. Ways to implement this retesting with acceptable performance are out of the scope of this paper.

2. Motivating Examples

In this section, two example programs are examined to motivate the work. The first example contains a sequence of mutually independent conditional branches. If it is tested in JCute without partition to obtain the failure constraint, the number of paths to explore is exponential in the number of branches. However, if it is partitioned into segments before the analysis, the growth of complexity can be made linear.

The second example is a program with a loop that has an unbounded number of paths. If it is partitioned before test, a constraint can be obtained in bounded time (in fact, 10 runs).

2.1 Example 1

Table 1 shows a simple program with 3 independent branch statements in a series.

The "@Failure" annotation has a string parameter that specifies the failure condition. In this case, the program fails if and only if variable `failure` is assigned true. In the test, this annotation (with the dummy variable `fail` following it, whose name does not matter) has an effect as the following statement: [1]

```
if (failure) {
    System.out.println("!!! FAILURE !!!");
    return;
} else {
    return;
}
```

#	Constraint
1	op1 && op2 && op3 && failure
2	op1 && op2 && !op3 && failure
3	op1 && !op2 && op3 && failure
4	op1 && !op2 && !op3 && failure
5	!op1 && op2 && op3 && failure
6	!op1 && op2 && !op3 && failure
7	!op1 && !op2 && op3 && failure
8	!op1 && !op2 && !op3 && failure
	failure

Table 2. Path constraints for example 1

To test function `f`, parameters `op1`, `op2`, `op3` and `failure` are arbitrarily generated by JCute, each taking value of either true or false. A `main` function is also generated, which, after producing the parameters, calls `f` on a fresh Java object in class `A`. (No object is created if the method under test is static.)

The program contains 4 conditional branches (including the one corresponding to the failure annotation), each performs the test on a different parameter. JCute produces 16 paths during the test, half of which have "!!! FAILURE !!!" outputs. Table 2 shows the failure constraints for them. The disjunction of these constraints, "failure", is the necessary and sufficient condition for the whole program to fail.

An observation from this result is that one need not explore all the 16 paths to obtain the program's failure constraint in this particular case, because it only depends on the `failure` parameter.

The approach to be presented in the next section helps to reduce the complexity of generating failure constraints for programs of this kind.

2.2 Example 2

A quite different problem occurs when the program under test has an unbounded loop. An example is can be found in Table 3.

To test this program, JCute generates assignments for parameters `i` and `j`. Because the loop variable `k` iterates from 0 to `i`, and `i`'s value is not constrained by the environment, the number of iterations is unbounded. For this test, JCute

[1] A program transformer actually replaces the annotation with the if-statement before performing a test.

```
public class B {
    void f(int i, int j) {
        for (int k = 0; k < i; k++) {
            if (i == j) {
                @Failure("true") int fail;
            }
        }
    }
}
```

Table 3. A program with an unbounded loop

keeps generating new paths until the pre-defined maximum number (10000 in the current implementation) is reached.

It is obviously time wasting to generate all these paths. Specifically, the loop does not modify `i` or `j`, so the constraint evaluates to the same true or false value regardless of the iteration. A natural question to ask is whether we can determine the constraint even for this kind of programs. The approach to be presented in the next section provides a "yes" answer by computing a fixpoint.

3. Program Segmentation and Partial Execution

In this section, we will present an innovative approach to determining failure constraint.

Given a program with branches, we first partition it into a number of segments. The number and the sizes of the segments can be controlled by the human tester. In the experiments presented here, we partition the programs at the branches in hope of better efficiency. For instance, the program in example 1 can be partitioned as in Table 4.

Out of the 4 segments, the last one is special because it contains the provided failure constraint instead of executable code. Our partial execution starts by testing segment 3 together with the constraint in segment 4. A new constraint is generated as the disjunction of the path constraints for all the failing paths collected. In this case, after simplification, the constraint remains to be "failure", because `op3` does not have an impact on it.

The program can then be rewritten and repartitioned as in Table 5. (Note that the failure annotation is hoisted one branch up.)

JCute is then used to again execute the program, producing a new constraint for segments 2 and 3. This rewrite-and-test process repeats until a constraint for the whole program is obtained.

3.1 Efficiency Assessment

The complexity of the above testing process is easy to calculate. In each test there is an if-statement followed by an annotated failure constraint, which is always "failure". Therefore, each test returns 4 paths, 2 of which have failure. 3 tests need to be performed in a series before the head of the

```
public class A {
    void f(boolean op1, boolean op2,
           boolean op3, boolean failure) {
        /* Start of Segment 1 */
        if (op1) {
            System.out.println("Operation 1");
        }
        /*  End of Segment 1  */

        /* Start of Segment 2 */
        if (op2) {
            System.out.println("Operation 2");
        }
        /*  End of Segment 2  */

        /* Start of Segment 3 */
        if (op3) {
            System.out.println("Operation 3");
        }
        /*  End of Segment 3  */

        /* Start of Segment 4 */
        @Failure("failure") int fail;
        /*  End of Segment 4  */
    }
}
```

Table 4. A partition of example 1

program is reached, so the total number of paths explored is $4 + 4 + 4 = 12$. Compared to the approach without partitioning (which requires to explore $2^4 = 16$ paths) the number of paths is 4 less. (In fact, many of those 12 paths are only sub-paths of a complete path, containing much fewer statements.)

The difference between the partitioning approach and the non-partitioning approach can be very big in practice. Take the example in Figure 1. The complete program has $8 * 8 * 2 = 128$ paths, all of which need to be explored if no partitioning is performed. However, if we partition it into two segments of the same size, then the total number of paths to explore reduces to $8*2+8*2 = 32$. (Here we assume that the intermediate constraints can be ideally simplified into a single literal, such as "failure" seen above. This is the best case scenario. Discussion on the best case and the worst case is postponed to the end of this paper.)

3.2 Loop Unrolling

Loops can be problematic. Undecidability of the number of iterations is part of the problem. Because of this, it is generally impossible to summarize the exact loop failure constraints in finite time.

The example program in Table 3, however, is a special case in which, even though the number of iterations is unde-

```
public class A {
    void f(boolean op1, boolean op2,
           boolean op3, boolean failure) {
        /* Start of Segment 1 */
        if (op1) {
            System.out.println("Operation 1");
        }
        /*  End of Segment 1  */

        /* Start of Segment 2 */
        if (op2) {
            System.out.println("Operation 2");
        }
        /*  End of Segment 2  */

        /* Start of Segment 3 */
        @Failure("failure") int fail;
        if (op3) {
            System.out.println("Operation 3");
        }
        /*  End of Segment 3  */
    }
}
```

Table 5. Rewriting example 1 and repartitioning it with one less segment

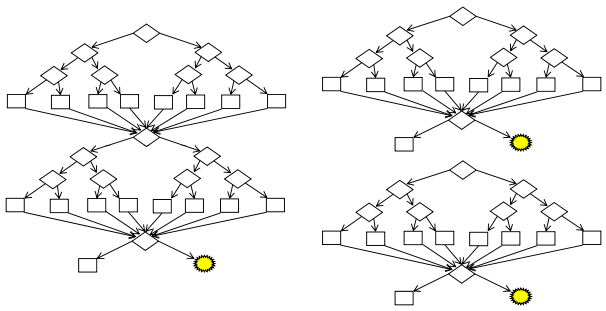

Figure 1. An imaginary program structure and a partition of it into two segments. Diamonds in the graphs represent branches; boxes represent statements; and spined balloons represent unconditional failure. The left graph shows the structure of the complete program. The right graphs are the two segments.

cidable, we can still determine the exact failure constraint. In this code, the number of paths depends on parameter `i`, which is unconstrained. In our approach, the testing tool tries to compute a fixpoint of a sequence of shrinking failure constraints. The algorithm stops once a fixpoint is reached. (Refer to [4] for the fixpoint theory.)

To help understand the algorithm, we first consider a loop to be unrolled to a decision tree with arbitrary but finite height. Figure 2 shows the unrolling of the loop in example

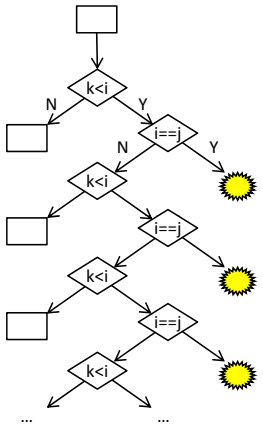

Figure 2. Unrolling of the example in Table 3, resulting in a tree of unbounded height

2. Though its height is unbounded (denoted with ...) in general, in practice each execution always corresponds to a tree with fixed height.

The above observation is essential. A naive version of our approach is as follows: Program testing starts from the end of the tree, corresponding to the last iteration (together with all other statements behind the loop, which do not exist in this example). This portion is shown as the first subtree tree in Figure 3. After testing this portion (compare to a segment of example 1), a failure constraint is obtained with the path constraints produced by JCute. This corresponds to one single iteration of the loop. In this case, 3 paths are explored, and the failure constraint returned is "k <= i-1 && i == j".

This constraint is then plugged into the second last iteration (subtree ② in Figure 3), and a second test is performed, exploring 4 paths and producing constraint "k <= i-1 && i == j" (the same as the previous one).

If no extra consideration is taken, for a loop that is supposed to execute 3 iterations, 3 tests need to be performed to obtain the loop constraint, which can then be used to prove constraints of the whole program in the 4th test.

A problem of this naive version of our approach is that it is impossible to tell beforehand how many tests to perform (if we do not assume the number of iterations). A better solution would be to consider obtaining a fixpoint as the termination condition. The tests repeat until the failure constraint generated is equivalent to the previous one. We call this constraint a *fixpoint*. It can be easily proved that once we obtain a fixpoint, we no longer need to perform tests, because more tests will just produce the same constraint. (Recall that we made the assumption of side-effect-free program.)

In this example, a fixpoint is detected at the end of the second test. Therefore, this constraint, "k <= i-1 && i == j", is used to test the whole program (with the loop initializer `k = 0`). The resulting constraint is simply "i >= 1 && i ==

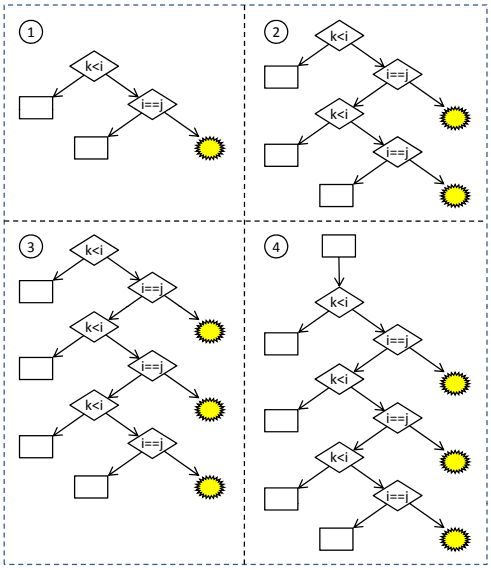

Figure 3. Subtree testing in the naive approach. ① sits at the bottom of the unrolled tree and is first tested. A failure constraint is summarized. This constraint is turned into a branch to substitute the left child of ②. ② is then tested to obtain the constraint for 2 iterations, which in turn is used to substitute the left child of ③, corresponding to 3 iterations. Assume that the loop exits in 3 iterations, then the constraint obtained from ③ can be used to determine the constraint of the whole program ④.

j". The first test explores 3 paths in subtree ①; the second explores 4 in subtree ②; subtree ③ is never tested because a fixpoint has already been reached; testing subtree ④ adds 3 to the number of paths. In total, 10 paths are explored.

A fixpoint may not be found always. For example, the loop failure constraint may keep growing with more and more conjunctive components, and never reaches a fixpoint. For such programs, we fall back to the traditional compromise, e.g., by pre-defining a maximum number of tests.

3.3 Effectiveness Evidence of Loop Unrolling

Compared to non-partitioning testing approach, in which one cannot generate a failure constraint for example 2, the fixpoint mechanism clearly is an improvement in this particular case. A question to ask is then whether this result is general. This is believed by the author to be true for at least a certain kind of programs: *the kind of programs that have loops in them, but the failure constraints for the loop "shrink" monotonically after a bounded number of iterations*.

We now define the "shrink" relation between two constraints. Constraint C_1 *shrinks to* constraint C_2 if any only if [2]

1. $C_1 \Rightarrow C_2$
2. Let V_1 be the set of variables that occur in C_1, and let V_2 be the set of those that occur in C_2. $C_1 \neq C_2 \Rightarrow V_1 \supset V_2$.

If the constraints shrink monotonically over the tests, then they must reach a fixpoint after a bounded number of steps. If the constraints C_1 and C_2 from two consecutive tests are equivalent, the they are themselves a fixpoint. If they are not equivalent, monotonicity requires that the one obtained with one less iteration to shrink to the one obtained with one more iteration. Therefore, the latter constraint has at least one less variable occurring in it. Since there are only bounded number of variables after a bounded number of iterations (even if arrays are considered), this shrinking process must terminate at some point (on or before all variables are consumed).

4. Conclusion

From the two examples provided here, we have shown that the dynamic testing of two kinds of programs for failure constraints can be improved by partitioning. For a program with a sequence of if-statements, it is usually a good idea to test one big if-statement at a time, and to use the generated constraint to further test the other if-statements above. For a program with unbounded loops, a solution to the loop failure constraint may be obtained in finite time by computing a fixpoint.

The current implementation in the form of an Eclipse [1] plugin serves as a research prototype. It does not support method calls at this time. For loops, it handles only for-statements, though while-statements and do-while-statements can be easily rewritten into for-statements.

Another limitation lies in the constraint simplifier. We have seen in the examples that the intermediate constraints need to be simplified. This is for efficiency reason. In example 1, since we were able to simplify the intermediate constraints with only one literal remaining, we successfully reduced the exponential complexity to linear complexity. However, assume that we were not able to simplify the constraints, then the situation would be changed. After the first test of the 3rd branch (with 4 paths), constraint "op3 s&& failure || !op3 && failure" would be obtained. The second test would incur 8 paths if no simplification were performed. 2 more disjunctive components would be added to the constraint. The third test would then explore 16 paths without simplification, exactly the same number as the number of paths explored without partitioning. This is the worst case scenario, in which the total number of paths explored be-

[2] Condition 2 can be slightly relaxed if we consider the finiteness of the values that variables in V_1 and V_2 can take. This relaxation is out of the scope of this paper.

comes $4 + 8 + 16 = 28$. In general, if there are n branches and the constraints are not simplified, the number of paths is $2*2^1+2*2^2+\ldots+2*2^n = 2^{n+2}-4$, even bigger than the non-partitioning approach, which requires to explore 2^{n+1} paths.

The above best case and worst case scenarios show that it is crucial to be equipped with a good constraint simplifier. Currently, the constraint simplifier is ad-hoc, so it may not be able to simplify all constraints that the tester encounters with redundant components. This limitation can be eliminated by using a constraint simplification library that implements some techniques as in [6].

References

[1] Eclipse. http://eclipse.org/.

[2] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates, 2002.

[4] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order (2nd Edition)*. Cambridge University Press, April 2002.

[5] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[6] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[7] Koushik Sen and Gul Agha. Cute and jcute : Concolic unit testing and explicit path model-checking tools. In *18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.