Abstract

Test generation is easy: we can generate piles of tests randomly, by model checker mutation analysis of formal specifications, through path coverage, etc. But just increasing the quantity of tests may not be a cost-effective way of increasing the quality of the test. The number of tests, and hence the cost of maintaining and running large test sets, can increase exponentially with the increase in real coverage.

We present a number of automated methods to pick out a smaller set of tests that still have good coverage. The first method is finding a subset of tests with equal coverage for a test criterion. To do this, we must measure the coverage for a test set, given a test criterion. Some possible test criteria are path coverage, n-switch, specification mutation, and branch coverage. We use a greedy algorithm to choose tests satisfying requirements of the criterion.

tests or duplicates may be dropped. Finally, the cross section of a requirement is the reciprocal number of tests that satisfy it. We can choose tests with the highest resolution, that is, tests satisfying requirements with the smallest cross section, since these are likely to be the most sensitive tests.

Introduction

Let us define a few terms. After [1], a test criterion is the overall strategic goal, or a judgment about what aspects one wishes to test. Some criteria are source code branch coverage, du-paths, specification boundary testing, MC/DC, random testing, use cases, and mutation adequacy [2]. A test objective is a specific, tactical goal. It comes from considering the unit under test and the test criteria. Here are some typical test objectives and their test criterion.

- Execute the program so branch #27 is not taken (branch coverage).
- Choose inputs so x is just barely less than y (boundary testing).
- Create an account, make a deposit to it, and then try to delete it (use case).

A test case is a specific test. It should satisfy one or more test objectives. (Otherwise, why consider it?) A test case consists of one or more steps. A test set is one or more test cases, usually having some common theme, objective, criterion, etc.

Test coverage is the extent to which a test set satisfies a test criterion, that is, how well the test set tests aspects of concern. It is often a percentage of the number of test objectives satisfied or some other simple, objective function.

Generating huge tests sets is easy for some test criterion. However, larger and larger test sets do not automatically lead to proportionately better test coverage, especially when there is a random element in test generation. Unless there is a very particular guiding force, random selections tend to produce broadly similar test cases over and over.

Two alternatives are to use a test generation method that produces few, if any, overlapping test cases or do a post-processing step to exclude overlapping test cases. Using a different, deterministic test generation method that produces few overlaps is desirable, but not always possible. Also, some level of random testing helps cover behavior that may be otherwise overlooked.
Winnowing

We call the post-processing step to minimize redundant test cases *winnowing* following [3]. But why reduce the size of the test set at all, that is, why not run every possible test? The goal of test engineers is to “identify errors in a minimal fashion. Due to time constraints, if the test engineers get multiple tests that fail for the same reason, they will probably get frustrated...” [4] There is always limited time to run the tests and analyze the results.

Of course, in some situations the cost of winnowing test cases outweighs the benefit of less test run, analysis, and maintenance time. In other circumstances, test engineers may prefer many test cases that each precisely indicate a specific error to fewer cases whose failure could result from many errors. For those situations and test generation methods that call for winnowing test cases, there are many different techniques. Some techniques are applicable to just certain test generation methods. We discuss two techniques: dropping prefixes or extensions and finding a complete coverage subset.

Drop Prefixes or Extensions

Test cases may be divided into two categories: failing test cases and passing test cases [5]. Proper software should not exhibit the behavior checked in a failing test case. Safety and security constraints are often naturally modeled as failing test cases. For instance, the requirement “wheel retraction is not allowed with weight on wheels” leads to a test case with weight on wheels then a “landing gear up” command. The software should fail to issue a retraction sequence. On the other hand, proper software should do what is indicated in a passing test case. As passing tests are more common, we consider them first.

Some test cases may be prefixes of longer cases. That is, both test cases start in the same state, proceed with identical inputs and expected outputs, but one has additional steps. For passing test cases, correct software should pass both the longer test case and the prefix. Since the full test case exercises more (or strictly speaking, no less) behavior than the prefix, prefixes of passing test cases may be discarded.

For failing test cases, we take the opposite tack. We drop failing test cases that are extensions of shorter failing test cases. Since correct software should fail a short test case, trying a longer test case is no more sensitive. The software should have already departed from the test case during the first steps.

Some test generation techniques, such as mutation analysis, generate many duplicate test cases. All but one of the duplicates may be dropped. Thus, a single step of sorting and syntactic comparing can reduce the number of test cases, and hence the size of the resulting test set. For some methods it may reduce the size by an order of magnitude and still have identical coverage.

Complete Coverage Subset

Since a single test case often satisfies more than one test objective, we may further reduce the size of the final test set by choosing a subset of the test cases. Although finding a minimum subset that covers the objectives is NP-complete, simple greedy heuristics often suffice.

An approach is to effectively create a matrix of all test cases indicating which test requirements each one satisfies. Test cases that are dominated by other cases may be dropped. That is, if one test case satisfies only a proper subset of requirements of another, it may be dropped.

Another method is to begin with an empty final test set and the set of candidate test cases. Repeatedly find a test requirement that is not satisfied by any case in the final set, and choose a candidate test case that satisfies it. Continuing until all requirements are satisfied, we hopefully have a smaller test set.

Highest Resolution Tests

Although the previously described heuristics can reduce the size of test sets, there is still a window of opportunity. Recall that generating test cases to satisfy test requirements is rarely our ultimate goal. We want to choose a test set, relatively miniscule compared with all possible executions, which has a good chance of revealing latent errors. Can we formalize the notion of picking a set of sensitive test cases?
Some errors are harder to find than others. Analogously, some test requirements are harder to satisfy than others. As a gross example, an error that requires many rare conditions to be manifest is harder to find than an error that immediately aborts the program. Similarly, a requirement to reach a state that takes a minimum of a dozen events to reach is harder to satisfy than a requirement to visit the start state. Whittaker and Voas [6] refer to the ease of finding a fault as the size of the fault. We use an analogy to the ease of detecting an airplane with radar or hitting an atomic particle. We call the ease of detecting an error or satisfying a requirement the cross section of the error or requirement.

A test requirement with a large cross section is generally easier to satisfy than one with a small cross section, similarly for errors. Formally, the cross section of a requirement or error is the ratio of test cases that satisfy or detect it to all tests.

$$CS(r) = \frac{\text{# satisfying tests}}{\text{# tests}}$$

In general, we want test cases that satisfy those requirements with small cross sections. Presumably these test cases are quite sensitive and can detect errors with small cross sections, too. We call the ability of a test to satisfy requirements with small cross sections, the resolution. The higher the resolution, the more small cross section requirements the case satisfies. Formally, it is the sum of squares of reciprocals of requirement cross sections it satisfies.

$$RES(t) = \sum \frac{1}{CS(r)^2}$$

An improved subset heuristic begins with an empty final test set and a set of candidate test cases, as before. We pick the unsatisfied requirement with the smallest cross section, that is, the one satisfied by the smallest number of test cases. From all the test cases that satisfy the requirement, we choose the test case with the greatest resolution and add it to the final set. We continue picking unsatisfied, small cross section requirements and high-resolution test cases until all requirements are satisfied.

Example

As a very small example, we use the automobile cruise control presented in Atlee and Buckley [7]. The specification takes one table of 12 rows and nine columns in SCR or about 90 lines in SMV [8]. Using a mutation analysis criterion with many mutation operators, we generated 807 mutant specifications that yielded 379 counterexamples. A counterexample is an execution trace of the given state machine showing how the specification is false. Thus each counterexample is a potential test case. We could have chosen a set of mutation operators with nearly the same coverage, but with an order of magnitude fewer mutants [9]. We didn't bother for this tiny example.

After winnowing duplicates, 35 unique test cases remained. Since these are all passing tests, we found and dropped 11 test cases that were prefixes of other, longer cases, leaving 24 cases. A complete subset that satisfied all the requirements (killed all the mutants) consisted of only 19 tests.

For this example, the test cases required because they were the only ones that satisfied some requirement (killed some mutant) formed a set that satisfied all requirements. So no more elaborate grading on resolution was needed.

Obviously, cross sections may be combined in different ways to compute resolutions. Depending on the test generation method and requirements, it may be better to use the geometric or arithmetic mean of the reciprocals. We could use the sum of the reciprocals, instead of the square of reciprocals, to allow requirements with high cross sections to contribute more.

References


