Metamorphic Testing

By **Michael Stahl** - November 24, 2021
Share URL
Bookmark

Summary:
Rapid change and adoption of new ideas are attributes we readily assign to engineering and high-tech. What was a novelty and special last year is old news today. And yet… have you ever heard of Metamorphic Testing?

Rapid change and adoption of new ideas are attributes we readily assign to engineering and high-tech. What was a novelty and special last year is old news today. And yet… have you ever heard of Metamorphic Testing? If you have not, no wonder. It is a brand new idea. The **first paper** on this innovative test technique was published just recently… in 1998.

It's quite amazing that this technique is not better known, nor does it appear in most of the training material or textbooks for basic or advanced software testing. With the increasing ubiquity of AI-based applications, this technique, which solves some of the inherent problems in testing AI, will probably gain more popularity.

Why Aren't the Existing Test Techniques Enough?

Many of the systems we test are deterministic, which allows us to write test cases with unambiguous expected results. For example: Given a code that implements some function y=f(x), we can calculate an expected result for f(x) using a simulator or different implementation of the function and compare the two results. In fact, the basic test techniques (equivalence class partitioning; boundary value testing; decision tables etc.) are all based on the assumption that we have a "test oracle" which provides us with an expected result to compare against.

However, there are many systems where the situation is not that simple. In such systems the expected result can be calculated, but only at a specific time or with a specific set of data. When the data in the system changes, the expected results change as well and the effort to calculate the expected result needs to be repeated. Other systems may call for investing significant compute resources or logistic efforts to implement a test oracle. Programs that use large databases are a good example for such systems. Think of a flight-search application (remember those things – flights?...). As long as we run our tests on a test database, populated with well-known test data, all is good. We can pre-calculate the expected results for any query we may run. But experience shows that when such applications are released to the real world, new bugs are found due to combinations of data that the developers did not think about or thought were not possible. The solution is to run tests "in production" or at least on a copy of the production database contents. The problem then is that it's impossible to define the expected results for any test query. The data in the real-world database changes continuously, which means we test with a different set of data each time we run a test cycle.

Another problem is the cost of creating test cases. Even if we had a test environment with flight data that is completely under our control, it is not effective to run all the test cycles on exactly the same dataset, since we need to verify that the search algorithm works when the database content changes. Assume we wrote a test case that searches for a flight to New York for two persons, in economy class. We know that in the current test database there are two flights that fit the parameters we provided (date, departure location, etc.). If the test case has a deterministic expected result (e.g. “two available flights”), it means that before executing the test we must make sure the database contains only two flights that match our search criteria, otherwise the test would fail. For thorough testing we will have many such tests, and before running each test we would have to update the database with the correct information so the expected results would match the database contents. If this sounds inefficient, it’s because it is. It also means that we will miss combinations or situations we can’t think of. In more complex systems, where there are non-linear or time-dependent relations between data, inputs, and outputs, the problem is even more difficult.

For some programs, defining an expected result is by-definition a challenge. Here is an example: Let’s say we research the properties of some material under extreme environmental conditions. We developed an algorithm that describes the material’s behavior and we want to test if the algorithm is correct. All that is left is to go to the lab, run some experiments on the given material and compare the experimental results with the results from the algorithm. The algorithm is implemented in code and we, as testers, need to test this code. But when we see a mismatch between the code results and the lab results, how can we tell if the problem is in the algorithm or in the code implementing it Additionally, if the experiments are costly, we would want to have a way to test the algorithm with a minimal number of results from real-life lab tests. How can we test the algorithm without having “expected results”?

Metamorphic Testing provides a partial solution to the problems I raised. This technique allows generating many test cases automatically and does not require pre-determination of the expected results.

The Technique

There are many situations where it is possible to define a *relation* between the result of one test to results of following tests, without consideration for the correctness of the results of the first test. Let’s get back to the flight-search example and see how even when the contents of the flights database are unknown we can still say something intelligent about the results of tests, once we have the results of one initial test.

First, we will copy the production database to a test database – to avoid the risk that data changes while tests are executing. We will then run the first test: searching for a flight to NY for two people. We will get some result (let’s say: three available flights). Since we don’t know the content of the database, we can’t tell if this result is correct. But now that we know this result, we can say something clear about the expected results of further tests, which are permutations of the first test:

The Change	**Expected Result**
Search flight to NY for three passengers; all other parameters are the same	The same number of flights as the first test, or less
Search flight to NY for one passenger; all other parameters are the same	The same number of flights as the first test, or more

Search flight to cities within 100 miles radius of NY for two passengers; all other parameters are the same	The same number of flights as the first test, or more

The first test can be run not only on different databases, but also with different parameters, such as dates, cost, with or without stops, etc. Even though we have no idea if the test results are correct, we can still run the next three tests and be very clear about their expected results. We removed the tight linkage between test cases and the database contents. Using this approach, we can define many **metamorphic relations** between the results of one test case and the expected results of many other tests that are derivatives of the first test.

Metamorphic testing allows the creation of a large number of tests on complex systems without the need to invest much time and effort in calculating expected results or in setting up the test system to ensure a specific result. Here is another example, which appears in many of the papers about the technique:

Test an implementation of the trigonometric function sin(x).

For each test of sin(x) we can also run the following tests:

- sin(-x) : The result should be the same as for the first test, with a sign inversion
- sin(x+π) : The result should be the same as for the first test, with a sign inversion
- sin(x+2π) : The result should be the same as for the first test

More relations can be defined this way.

Note: When using this technique to test a basic property of a function or a system, as in the case of the sine function, some texts call it **property-based testing**. The idea is similar to metamorphic tests.

Implementation

The challenging part in implementing metamorphic testing – which is also the interesting part – is the definition of useful metamorphic relations. This is something that has no systematic or algorithmic solution. You must sit down and rack your brain – and it's not a trivial challenge. In the original 1998 paper the authors give some examples for metamorphic relations in a binary search and some of these relations are not something you'd naturally think of.

Here is an example from my own experience where we developed tests that now, in hindsight, I understand were metamorphic tests.

We tested an object-location algorithm. The input to the algorithm was a video clip that contained the object. The algorithm was supposed to identify that the clip contained the object and to report its location in the frame. The camera panned from right to left, so the location of the object changed 30 times each second – at the frame rate of the video. To create "expected results" we tagged every 10th frame in the video with the location of the object. This was tedious and labor-intensive work. With this approach, we tested only every 10th frame and were "blind" to possible inaccuracies in the calculations of the object location on unmarked frames. But then we understood we can test much more without having to tag each frame. Since the camera pans right-to-left and the object is on the left-hand side of the frame initially, then at each frame, the object should be a bit to the right of its location in the previous frame. Using this relation, we could now test the algorithm's output on every frame with very little extra work. Not only is this effective, research also shows that metamorphic testing can find bugs that manage to slip through oracle-based testing.

Coverage Metrics

It is customary to suggest a coverage metric when defining a test technique. This gives a measure for how well the tests we created cover all the possible tests that the technique defines. For example, a 100% coverage for equivalence class testing means that each equivalent class was represented in at least one test case. However, defining a coverage metric for metamorphic testing is not so easy. To start with, there is no clear upper boundary to the number of metamorphic relations that can be defined. In the flight search example I gave three relations – but it is clear I could define many others. Moreover, for each relation, we can run a practically infinite number of tests, by changing the input to the first test, and thus changing the expected results of the related metamorphic tests. Indeed, currently there are no defined coverage metrics for this technique and it's unlikely there will be. It is logical, however, to expect that for each identified metamorphic relation there will be at least one test.

Further Reading

If you'd like to learn more about metamorphic testing, I highly recommend reading **Hillel Wayne's Metamorphic Testing** which is a good summary of the topic. At the end of that paper there is a list of resources. The one I found most informative is **Metamorphic Testing: A Review of Challenges and Opportunities**.