Heuristic Test Oracles

QUICK LOOK

- When and why to use heuristic oracles
- Choosing your heuristic
- When heuristics won't work

The balance between exhaustive comparison and no comparison at all

by Douglas Hoffman

Capture and comparison of results is one key to successful software testing. For manual tests this often consists of viewing results to determine if they are anything like what we might expect. It is more complicated with automated tests, as each automated test case provides a set of inputs to the software under test (SUT) and compares the returned results against what is expected. Expected results are generated using a mechanism called a *test oracle*.

The term *oracle* may be used to mean several things in testing—the process of generating expected results, the expected results themselves, or the answer to whether or not the actual results are what we expected. In this article, the word oracle is used to mean an alternate program or mechanism used for generating expected results.

Table 1: Five Approaches to Oracles

The heuristic approach is only one implement in a tool chest for SQA—the decision about which approach is best for you depends on your test situation. The table below gives descriptions of five approaches to oracles that have been successfully employed across the software industry to verify automated software tests.

	TRUE ORACLE	HEURISTIC ORACLE	SAMPLING ORACLE	CONSISTENT ORACLE	NO ORACLE
Definition	■ Independent generation of all expected results	■ Verifies some values, as well as consistency of remaining values	■ Selects a specific collection of inputs or results	■ Verifies current run results with a previous run (Regression Test)	■ Doesn't check correctness of results (only that some results were produced)
Advantages	■ No encountered errors go undetected	■ Faster and easier than True Oracle ■ Much less expensive to create and use	■ Can select easily computed or recognized results ■ Can manually verify with only simple oracle	■ Fastest method using an oracle ■ Verification is straightforward ■ Can generate and verify large amounts of data	■ Can run any amount of data (limited only by the time the SUT takes)
Disadvantages	■ Expensive to implement ■ Complex and often time-consuming when run	■ Can miss systematic errors (as in following *sine* wave example)	■ May miss systematic and specific errors ■ Often "trains the software to pass the test"	■ Original run may include undetected errors	■ Only spectacular failures are noticed

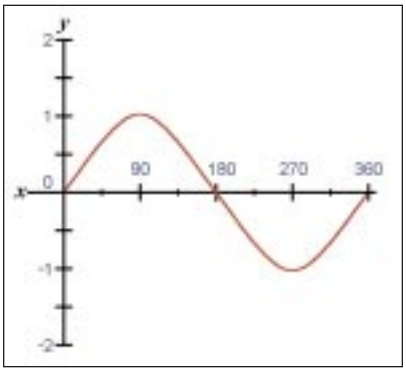

FIGURE 1 A *sine* wave

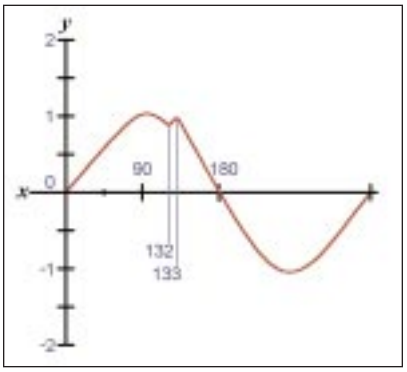

FIGURE 2 A *sine* wave with errors that could be detected by a heuristic oracle

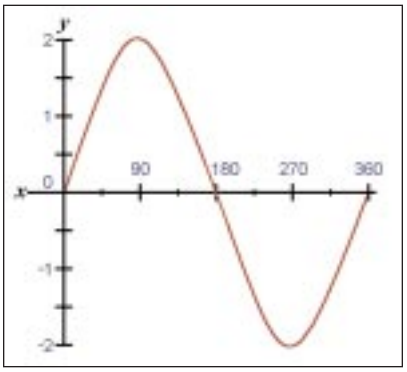

FIGURE 3 Another incorrect *sine* wave

It is often impractical to exactly reproduce or compare accurate results, but it isn't necessary for an oracle to be perfect to be useful. Several categories of oracles are described in Table 1. In this article, I'll describe some ideas associated with what I call *heuristic oracles*.

A heuristic oracle provides exact results for a few inputs and uses simpler consistency checks (heuristics) for the rest. Regardless of the complexity of the SUT, known or easily-computed result values can be chosen for the exact comparisons. The heuristic oracle can usually be built into the test case or verifier to simplify testing. This approach can have substantial advantages. Furthermore, the same heuristic oracle or simple variations are often reusable across broad classes of software.

As a simple example of the idea, consider the *sine* function (see Figure 1). An implementation of *sine* could be tested against a separately implemented routine that uses a different computational algorithm. That separate routine is a True Oracle. Such an oracle is very flexible—it can be used with as many test inputs as you have time to generate, it can accept any inputs the SUT can, and it has a high likelihood of identifying errors.

Note that it won't necessarily find all errors because it might share some with the SUT. For example, the same hardware or operating system fault might affect both (such as the "Pentium bug"), or both might use the wrong units. In such cases, both the SUT and the oracle could produce the same wrong answer. Unfortunately, this independent oracle is expensive both to create and use, often costing as much or more than the SUT to develop and using equal or greater machine resources. It also has a high likelihood of having its own errors because its complexity often rivals the SUT.

The other extreme is to have no oracle at all. I've reviewed automated tests that were proudly created and run, sending thousands or millions of test values to the SUT—and confirming nothing more than that the test does not crash the system or provide some other spectacular notice to the tester. That's not expensive, but it's also rarely useful and certainly tells us nothing about whether the answers from the SUT are correct.

A heuristic oracle provides a reasonable alternative between slow, expensive, and voluminous results generation on the one hand, and unverified SUT results on the other. The approach can be used when the SUT can be characterized as having a nice, predictable relationship between the inputs and results—one that can be exploited in testing. (See the sidebar for a further discussion of the relationships where heuristic oracles can most easily be used.) For the *sine* function, the predictable relationship used for the heuristic is that the function increases between 0 and 90 degrees, decreases from 90 to 270 degrees, and increases

What are nice, predictable relationships?

Algorithms and results come in many forms. Often there are so many variables involved that we despair at the prospect of entering arbitrary data in a test, and we fall back on sets of input and result values we already know. Heuristic oracles depend upon our seeing some relationship between the inputs and results in the SUT and exploiting that relationship through an underlying simpler one.

So, one question is what types of relationships should we look for? I define "nice relationships" as ones that work for ranges of inputs and/or outputs. The heuristic must hold for *all* values in the definable range(s) of inputs or results—without gaps. When there are exceptions in the range, the function (or comparison) gets extremely complicated, usually negating the value of using the heuristic oracle. It gets really difficult to sort through input values to decide if the relationship applies. If the *sine* relationship (sequential results getting larger or smaller) only worked for integer degree values, the heuristic wouldn't be very useful. In order to be "nice" the relationship needs to be continuous. It may only apply for one range of values, as is the case in the *sine* example, but a few simple checks on the input values easily put the results into one of the two heuristics.

"Predictable relationships" are fairly easy to identify—and the nature of the predictability is what we use to come up with the heuristic. In the *sine* example, the results predictably increase or decrease between the minima and maxima, and this is the basis of discovering the simple heuristic.

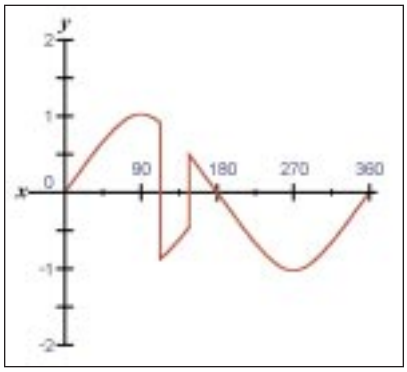

FIGURE 4 An invalid *sine* wave with inverted values

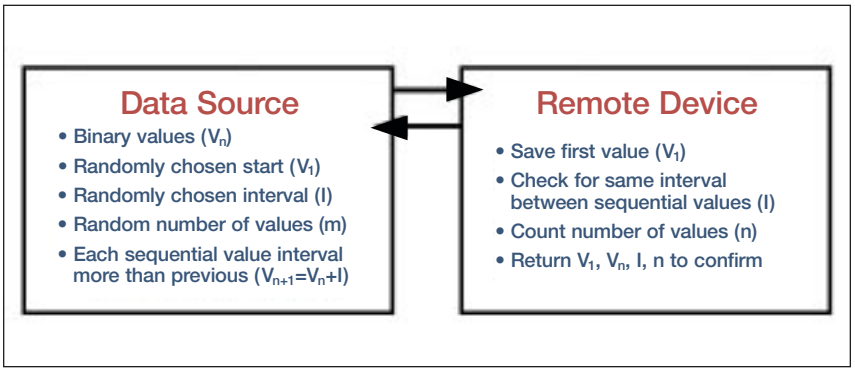

FIGURE 5 A heuristic for data communications verification

again to 360 degrees. The exact value of *sine* at 34 degrees is hard to predict, but you can easily check that it's larger than the value at 33 degrees and smaller than the value at 35 degrees. For the values of *sine* that are easy to predict **[sin(0) is 0, sin(90 degrees) is 1, sin(180 degrees) is 0, and sin(270 degrees) is –1]**, exact results can be checked. Between the four values the heuristic applies.

This heuristic oracle is very easy to implement compared to an alternate version of *sine* and runs much faster. The number of inputs to try can be chosen based on the time allocated to run the test. It will find errors such as those shown in Figures 2 and 3. In Figure 2, the predictable descent from crest to trough is violated. In Figure 3, the curve is too large in the vertical direction. Heuristic oracles are also good at finding errors such as the discontinuous (incorrect) sine wave in Figure 4. This heuristic approach also finds most problems at internal boundary conditions and special value cases.

Larger Examples

Math functions are not the only place that heuristic oracles are useful. I recall two database applications where we overcame practical limitations of automated testing using heuristic oracles. The first case was with a software company that produced commercial relational database systems. Large databases were needed, with a staggering number of transactions. Verification of results from the automated tests meant sifting through *huge* binary files containing complex interconnections, and verifying the correctness of all the data values and interconnections. That was not going to happen in our lifetimes.

But by applying heuristics to the relationships we were able to write oracles and automate verification for most of the database results. For example, auto-generated fields like Invoice Number correspond with Date/Time of creation, so when the results of searches are sorted by Date/Time they should have ascending (or descending) Invoice Numbers.

At another company the biggest challenge in testing the database was the vast size and dynamic updating of the data sets. Testing had to be done on the live data because it wasn't practical to duplicate a reasonable test set from the two hundred on-line data bases of 5 to 30GB each. The data within each record varied from a few thousand characters to several megabytes, making complete record comparison impractical. The data content in the live databases was also being modified daily, making it impossible to know in advance how many responses there would be to any reasonable query. However, we knew that old records weren't supposed to be dropped from the database, and we knew when each test was written. So we divided the results of a search into two sets: those records that predated the test (and were known), and those that came after (and the test couldn't predict). We checked that the first set was the right size (that no records had been incorrectly dropped), and that specific values appeared in each record. (Since we knew about these records, we selected information that was unique to the record and not likely to appear in new ones.) This meant the records contained specific expected values.

For the second set, we could only apply weaker heuristics to test that the search criteria had been met. In both cases, we could only check that the returned records matched (in some way) the search criteria. We couldn't check that all matching records had been returned. Even at the point the test was written, it would have been impractical to independently check every record in the databases.

Another example of heuristic oracle application occurred where the data integrity over a data communications link was tested with huge amounts of auto-generated, pseudo-random, binary data using simple algorithms for generation and checking (see Figure 5). We needed to generate a vast amount of random traffic and confirm that no data had been lost or corrupted. The sending test driver generated sequences of ascending or descending values with random start values, random lengths, and constant sequential differences chosen at random. The receiving test driver only needed to compute differences between sequential values, verify that the differences remained constant, and then send back the start value, end value, increment, and number of values to confirm any transmission.

Choosing Your Heuristic

The big trick with this approach is to come up with the heuristic to apply in the oracle. Most complex algorithms contain simpler patterns. If you step back and look at the relationships between the inputs and the results, you often can come up with really simple approximations or observations that give you the heuristic. Pictures of the functions, the data, and the SUT may help you visualize the patterns. Patterns may be based on only some of the

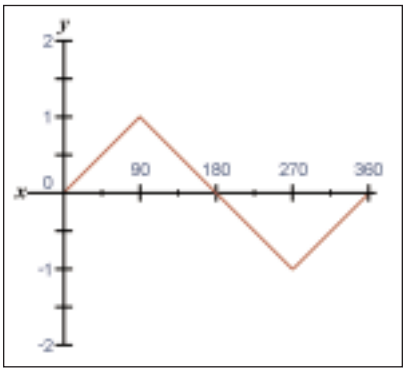

FIGURE 6 A sawtooth graph the oracle accepts

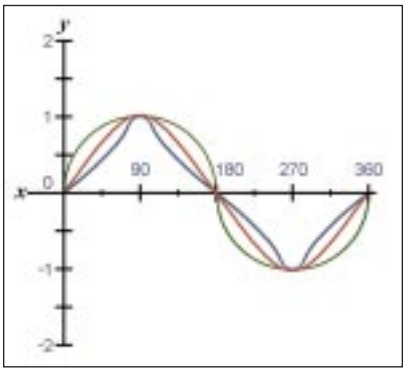

FIGURE 7 Graphs (due to rounding errors) the oracle accepts

input variables or results. There may be more than one pattern; so look for more than one, and then pick the simplest one that will do the job. The simpler the algorithm the faster, cheaper, and more reliable the heuristic oracle. If you can't see the simple approximation or rule-of-thumb, then you can't create a heuristic oracle for the function.

Some tricks may help. Break the SUT up into pieces. Each piece may be based on a range of inputs (or results). In the *sine* example, I broke the function into two ranges for the heuristics: rising results and falling results. The mega-databases were verified by splitting the results into two categories: those known at the time of writing the tests, and new results. Then two different heuristic approaches were applied to verify them.

Another trick is to look for other simple relationships between the input variables and results related to those you are testing. For example, the date/time of creation for a record may correlate with the record number, and therefore all records created on a specific date should form a block. (If there is a gap in the record numbers, the missing record must be absent from the database. It cannot have been created on any other day.)

You can often divide test results into two groups: what we know to expect, and what we can't predict. In the *sine* example, there were four values that could be predicted exactly. The rest were verified using a heuristic oracle.

When looking for patterns, consider reordering the data. For example, two equivalent sets of elements can be sorted and compared to show they do (or do not) contain the same items.

Blocks of information can often be represented by the starting value and count of elements. Variations are possible by using fixed increments between values or using simple patterns for changing increments.

When Heuristic Oracles Don't Work

If you don't have a simple pattern to work from, you don't have the necessary heuristic. GUI contents and navigations usually don't have simple patterns, and heuristic oracles won't apply. If your pattern is too complex, then you add the risk that the heuristic oracle will introduce errors (and therefore produce false reports of errors in the SUT). Overly complex patterns lead to expensive heuristic oracles that don't verify the accuracy of the SUT's results. If you're going to all that expense it might be better to use a True Oracle.

Keep in mind that the very nature of a heuristic means that it is not exact. The general rules can miss detecting actual problems. For instance, the heuristic oracle for the *sine* example would erroneously pass the sawtooth function in Figure 6. The four exact points are correct, and that function smoothly increases or decreases in the appropriate regions—but if verifying the accuracy is critical for *all* the values, then the heuristic oracle won't do the job.

However, it's not likely that a *sine* function would be incorrectly implemented as a sawtooth function. That's an important point. Heuristic oracles work when the failures they detect are failures that real programmers are likely to make. Heuristic oracles don't work if plausible mistakes would produce results that still satisfy the predictable relationship behind the oracle. For example, in the case of the *sine* function, rounding errors might lead to the kind of incorrect graphs shown in Figure 7, but these variances would still pass the heuristic oracle's test.

When planning to use a heuristic oracle, try to predict what failures it should catch and which it might miss, and ask if you're willing to accept that risk. In the *sine* example in Figure 7, other tests would be required to check for the rounding errors.

Care must be taken to choose the best method of results comparison when you are creating a test environment architecture and planning automated tests. Oracles are required for verification, but the nature of an oracle depends on several factors—most of which are under the control of the automation architect and test designer. In the range of oracles you have to choose from, heuristic oracles provide alternatives to more expensive or impractical True Oracles, while providing useful data about characteristics of expected results. When you find yourself in situations where complete information is unavailable or impractical to acquire, a heuristic oracle can offer you an important potential method of verification. STQE

Douglas Hoffman has been in the software engineering and quality assurance fields for over 25 years and is currently an independent consultant with Software Quality Methods, LLC. He is very active professionally, is a member of several professional societies, and has been a participant at dozens of software quality and metrics conferences. He has earned a Certificate in Software Quality Engineering from ASQ and has degrees in CS, EE, and an MBA. His email address is doug.hoffman@asqnet.org.