Designing Tests That Ensure

The

Observability of Defects

Richard Bender
Bender RBT Inc.
rbender@BenderRBT.com
518-506-8755

Note: This paper is a subset of “Comparing Requirements Based Testing Approaches”

The Challenge in Test Design

The goal of an effective set of tests is to ensure that if there are any defects in the product one or more of the tests will fail. That implies that the tester, on running the test, will be able to observe the defect in reviewing the results of the test – e.g. information on a screen, updates to a data base, messages going over the communications lines. However, when a test is run it usually causes the execution of many, many steps in the code, most of which are not observable. So the tester is deducing that all of those intermediate steps worked correctly by looking at the end results. The reality is that for a given test two or more defects might cancel each out and you get the right answer for the wrong reason. Also, something working correctly on one part of the path can hide something broken on another part of the path executed by the same test.

In testing integrated circuits, engineers understand this problem very well. They use algorithms to design tests that ensure that defects are propagated to an observable point. These are the Path Sensitizing Algorithms also known as the D Algorithms. These algorithms factor in the various ways defects can be hidden unless you have just the right combination of data across the set of tests. This is why you almost never find a functional defect in high end integrated circuits. Circuits have residual functional defect rates tens of thousands of times less than the equivalent functionality in software. The difference is not in the product; it is in the process.

In software, test design algorithms have only addressed reducing down the number of possible test combinations. For example, if you have a function with only six inputs and those inputs only have two states then you can create $2^6! = 64! = 1.27 * 10^{89}$ test suites. That is all of the possible tests in all possible orders. As you know taking the same tests and running them in a different order can produce different results so order is important. This, by the way is nine orders of magnitude greater than the number of molecules in the universe which is 10^{80th} according to Stephen Hawking. So the test design algorithms must be very smart about reducing the set of possible tests down to a number we have the time and resources to create and run.

All of the test design methods and tools address reducing down the number of test combinations. However, only the BenderRBT Test Design Tool (RBT) addresses the issue of the observability of defects. We have done this by adapting the hardware testing approach over to test software, which required significant enhancements to the algorithms. This is a huge, critical difference since, again, the goal of testing is to detect defects. Just making the big number of possible tests into a small number does not address this.

The Solution - RBT

Let’s look at what these algorithms do for us in designing tests using two examples.

Figure 1 shows a simple application rule that states that if you have A or B or C you should produce D. The test variations to test are shown in Figure 2. The “dash” just means that the variable is false. For example, the first variation is A true, B false, and C false, which should result in D true. Each type of logical operator – simple, and, or, nand, nor, xor, xnor, and not – has a well-defined set of variations to be tested. The number is always n+1 where n is the number of inputs to the relation statement. In the case of the “or” you take each variable true by itself with all the other inputs false and then take the all false case. You do not need the various two true at a time cases or three true at a time, etc. These turn out to be mathematically meaningless from a black box perspective. The test variations for each logical operator are then combined with those for other operators into test cases to test as much function in as small a number of tests as possible.

Let us assume that there are two defects in the code that implements our A or B or C gives us D rule. No matter what data you give it, it thinks A is always false and B is always true. There is no Geneva Convention for software that limits us to one defect per function.

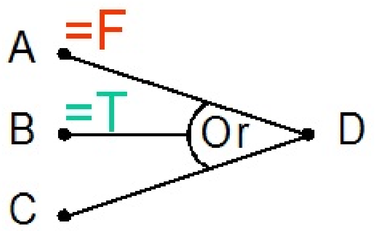
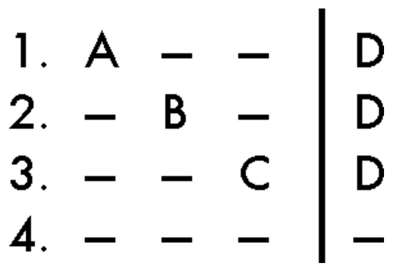


Figure 1 - Simple "OR" Function With Two Defects



1.	A	–	–	D
2.	–	B	–	D
3.	–	–	C	D
4.	–	–	–	–

Figure 2 - Required Test Cases For The "OR" Function

Figure 3 shows the results of running the tests. When we run test variation 1 the software says A is not true, it is false. However, is also says B is not false, it is true. The result is we get the right answer for the wrong reason. When we run the second test variation we enter B true, which the software always thinks is the case – we get the right answer. When we enter the third variation with just C true, the software thinks both B and C are true. Since this is an inclusive “or,” we still get the right answer. We are now reporting to management that we are three quarters done with our testing and everything is looking great. Only one more test to run and we are ready for production. However, when we enter the fourth test with all inputs false and still get D true, then we know we have a problem.

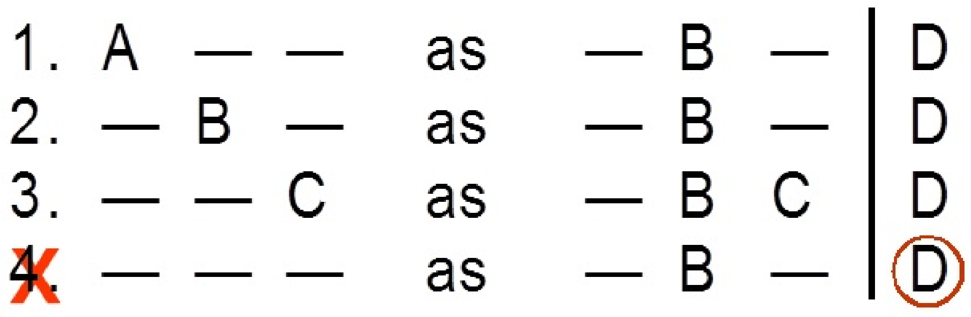


Figure 3 - Variable "B" Stuck True Defect Found By Test Case 4

There are two key things about this example so far. The first is that software, even when it is riddled with defects, will still produce correct results for many of the tests. The second thing is that if you do not pre-calculate the answer you were expecting and compare it to the answer you got you are not really testing. Sadly, the majority of what purports to be testing in our industry does not meet these criteria. People look at the test results and just see if they look “reasonable”. Part of the problem is that the specifications are not in sufficient detail to meet the most basic definition of testing.

When test variation four failed, it led to identifying the “B stuck true” defect. The code is fixed and test variation four, the only one that failed, is rerun. It now gives the correct results. This meets the common test completion criteria that every test has run correctly at least once and no severe defects are unresolved. The code is shipped into production. However, if you rerun test variation one, it now fails (see Figure 4). The “A stuck false” defect was not caused by fixing the B defect. When the B defect is fixed you can now see the A defect. When any defect is detected all of the related tests must be rerun.

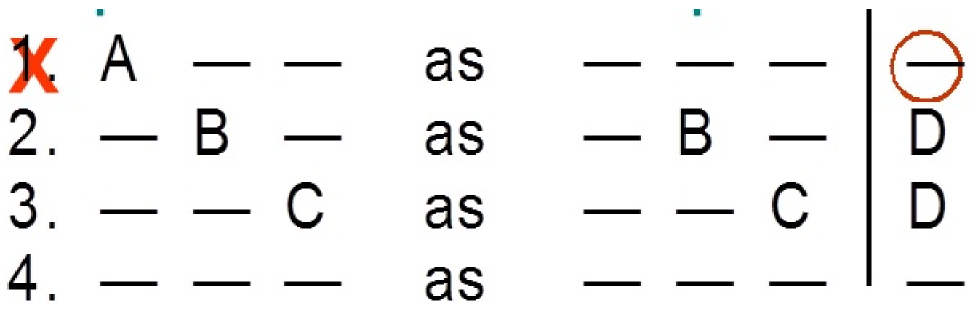


Figure 4 - Variable "A" Stuck False Defect Not Found Until Variable "B" Defect Fixed

The above example addresses the issue that two or more defects can sometimes cancel each other out giving the right answers for the wrong reasons. The problem is worse than that. The issue of observability must be taken into account. When you run a test how do you know it worked? You look at the outputs. For most systems these are updates to the databases, data on screens, data on reports, and data in communications packets. These are all externally observable.

In Figure 5 let us assume that node G is the observable output. C and F are not externally observable. We will indirectly deduce that the A, B, C function worked by looking at G. We will indirectly deduce that the D, E, F function worked by looking at G. Let us further assume there is a defect at A where the code always assumes that A is false no matter what the input is. A fairly obvious test case would be to have all of the inputs set to true. This should result in C, F, and G being set to true. When this test is entered the software says A is not true, it is false. Therefore, C is not set to the expected true value but is set to false but not observable. However, when we get to G it is still true as we expected because the D, E, F leg worked. In this case we did not see the defect at C because it was hidden by the F leg working correctly.

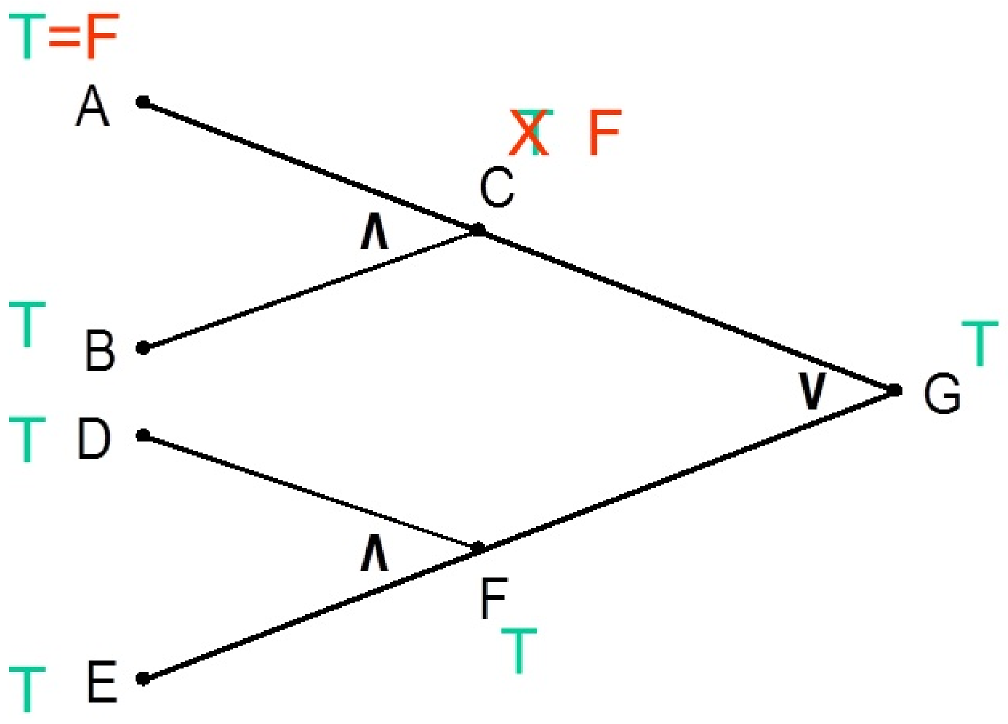


Figure 5 - Variable "A" Stuck False Defect Not Observable

Therefore, the test case design algorithms must factor in:

- The relations between the variables (e.g., and, or, not)
- The constraints between the data attributes (e.g., it is physically impossible for variables one and two to be true at the same time)
- The functional variations to test (i.e., the primitives to test for each logical relationship)

- Node observability

The design of the set of tests must be such that if one or more defects are present anywhere in that path, you are mathematically guaranteed that at least one test case will fail at an observable point. When that defect is fixed, if any additional defects are present, then one or more tests will fail at an observable point.

Designing such tests is non-trivial. In the above case we need to make sure that when we are testing the A, B, C rule that the D, E, F rule does not get in the way since neither C nor F are observable.

In Figure 6, let's say we are trying to test the A, B, C rule but C is not observable. Therefore we must ensure that whatever happened at A, B, C is observable indirectly at D. D is the result of an OR of C, D1, and D2. In designing the tests for the A, B, C rule you cannot allow D1 or D2 to be true. If either of them is, then D will be true no matter what happened at C. That would hide defects at C. So either D1 or D2 being true is incompatible in any test of the A, B, C rule.

Now let's say that D is also not observable. We must sensitize the path of the test for the A, B, C rule one more step to E. E is the result of an AND of D, E1, E2, and E3. If you let either E1 or E2 or E3 be false then E will be false, no matter what happened at D. If you cannot deduce what happened at D you cannot deduce what happened at C. Therefore, letting either E1 or E2 or E3 be false is incompatible with testing the A, B, C rule.

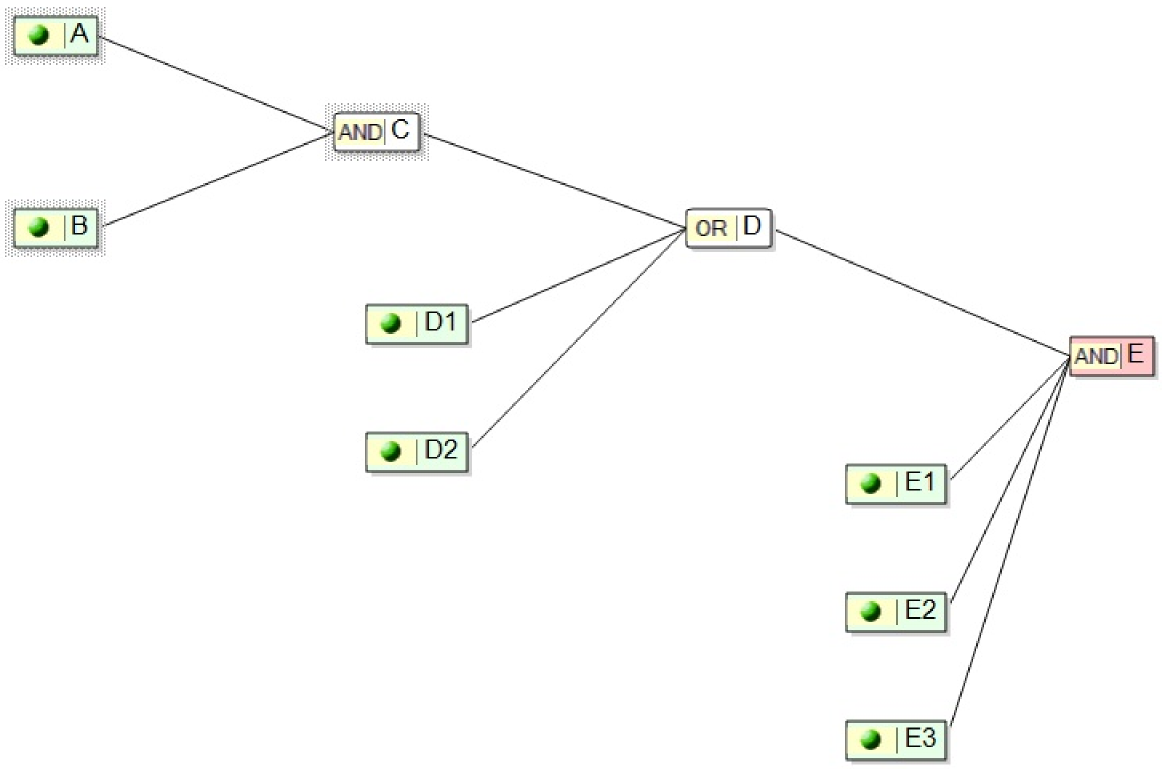


Figure 6 – Error Propagation Step 1

However, in Figure 7 we see that D2 is the result of an AND of X1 and X2. We also see that E3 is the result of an OR of Y1 and Y2 with Y2 in turn being the result of an AND of Z1 and Z2 and Z3. So the test design algorithm must factor all this in to ensure that D2 is false and E3 is true. The factors to consider in what states are compatible in ensuring that the A, B, C rule is working are growing.

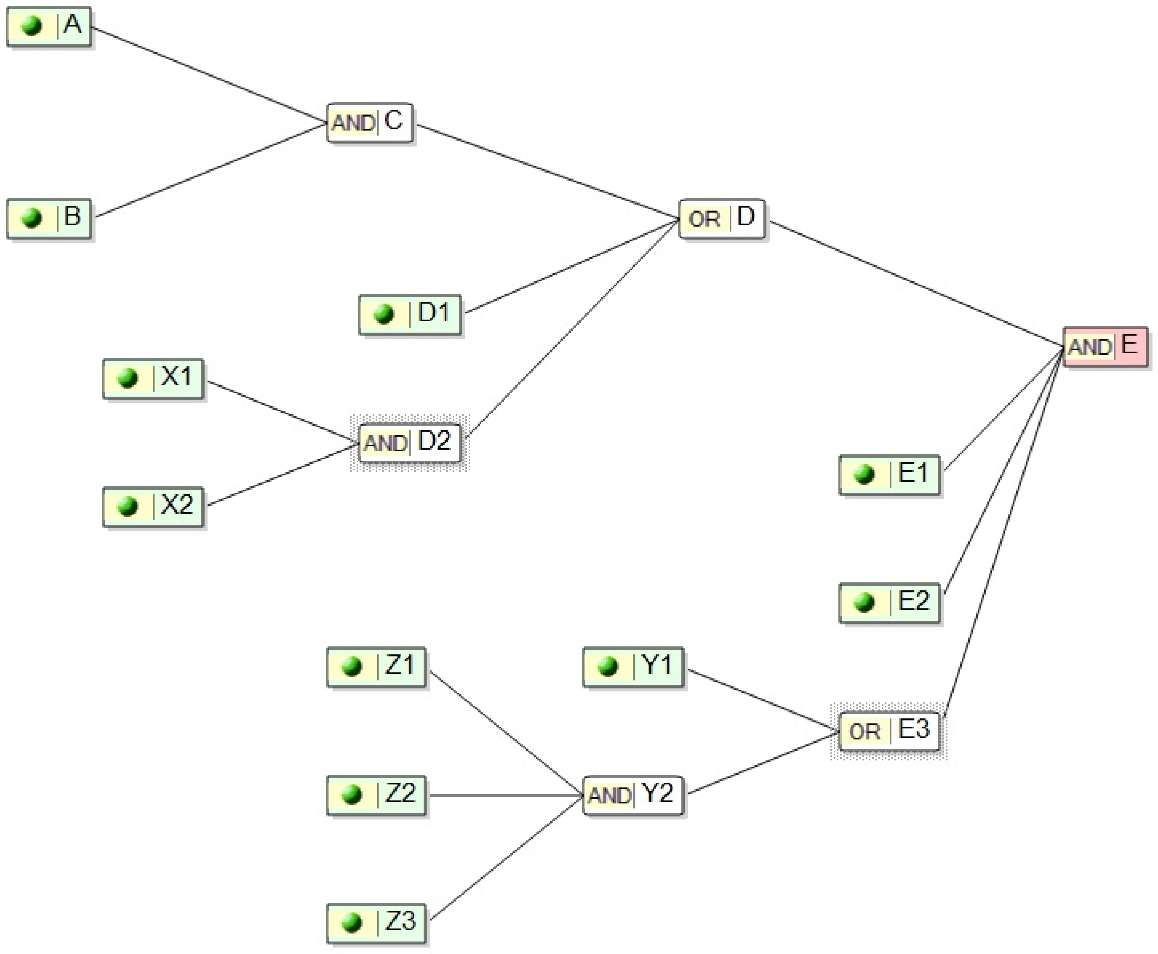


Figure 7 – Error Propagation Step 2

There is yet another set of factors to consider and these are the Constraints – i.e. the pre-conditions to this function. In Figure 8 we see that X2 and Z3 are MUTUALLY EXCLUSIVE. That means at most one of them can be true at any one time, though they could both be false. We also see that X1 True REQUIRES B True. The Constraints then further reduce the combinations of data that are compatible for ensuring the A, B, C rule works.

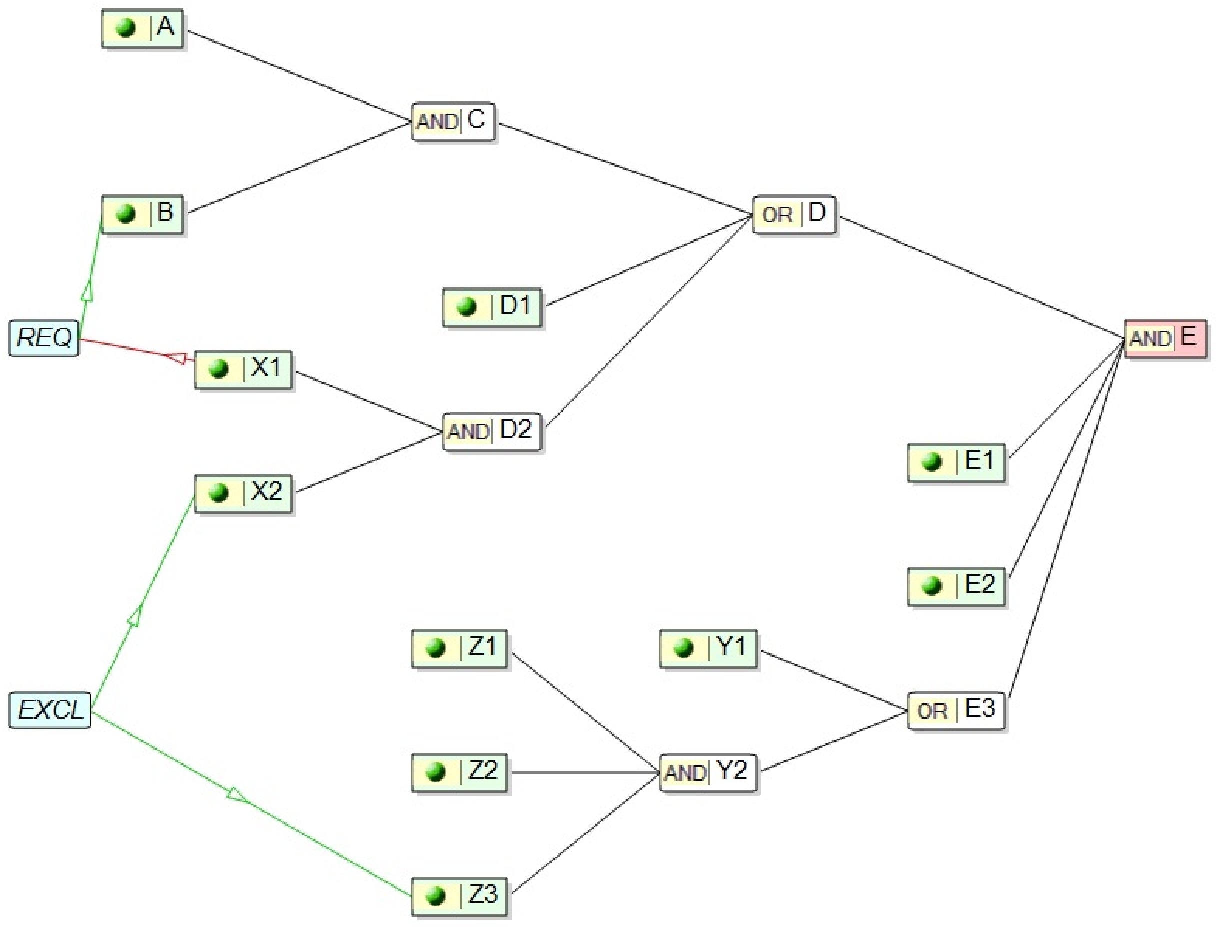


Figure 8 – Error Propagation Step 3

This process of identifying compatible states is repeated in designing tests that ensure that the rules creating D, D2, E, E3 and Y2 are tested in such a way that any defects in these rules will also be detected.

This does not mean a unique set of tests for each rule. The algorithms merge compatible states into the fewest tests possible that cover all of the rules. Each test will usually be testing multiple rules. They are highly optimized resulting in test suites that are at least half the size and provide twice the coverage of any other test design process.