How Do You Know When You Are Done Testing?

By

© Richard Bender
Bender RBT Inc.
17 Cardinale Lane
Queensbury, NY 12804
518-743-8755
rbender@BenderRBT.com

How Do You Know When You Are Done Testing?

You're responsible for a business critical, mission critical, or perhaps even safety critical system. To add to the complexity, portions of the system are outsourced to one or more vendors. How can you ensure that the quality of the delivered software is sufficient to meet the needs of its intended use?

Right up front there are four big issues which need to be understood. The first is the quality of the specifications from which they are designing tests. The second is the enormous number of possible tests. The third is that you need to ensure that you got the right answer for the right reason. The fourth is the understanding that testing is essentially like an insurance policy - you would not spend a million dollars to insure a $100K house. Similarly you would not spend a million dollars to test a system when your biggest downside risk was a $100K loss.

1. Testing, by definition, is comparing an expected result to an observed result. This is as true in testing software as it is in taking a math test in school. A test case is composed of a number of parts: the test input data, the system state(s) at the beginning of the test, the resultant outputs, and the post test system state(s). In order to determine what the test outputs and post test state(s) should be requires a deterministic specification - i.e., a specification written at a level of detail and clarity that allows the tester to predetermine the exact values of all post test data/states given any input values/states.

Very few projects have specifications at this level of detail unless the testers have been involved in testing the specifications to ensure that they are correct, complete, unambiguous, and logically consistent. (This is a topic we cover in detail in the "Writing Testable Requirements" and "Requirements Based Testing" courses.) The result of poor specifications is that testers design half tests - tests which only specify the inputs and initial state(s). When they run their tests they "validate" the outputs based on what they think is "reasonable". The result is that the test cases often find defects that the testers didn't because they did not know exactly what the test results should have been.

A well thought out test design process should force the issue on getting clarified specifications and aid in identifying such issues.

2. One of the major challenges in test case design is this simple fact: any system, except the most trivial ones, has more possible combinations and permutations of the inputs/states than there are molecules in the universe (which is 10^{80} according to Stephen Hawking in "A Brief History In Time"). The key challenge then is to select an infinitesimally small subset of tests which, if they run correctly, give you a very high

degree of assurance that all of the other combinations/permutations will also run correctly.

The software industry has done a very poor job at meeting this challenge. The result is that software still averages worse than 5 defects / 1K executable lines of code. On the other hand those producing high end integrated circuits (e.g., Intel, Motorola) achieve defect rates of less than 1 defect / million gates. Even though high end chips now exceed 40 million gates you rarely hear about a logic error on the released products. There is no fundamental difference in testing rules in software versus rules in hardware - rules are rules are rules. The difference in quality is solely due to the difference in the development and testing approaches. Hardware engineers use disciplined, rigorous approaches while software is still essentially a "craftsman" endeavor. Software is now far too complex and critical to continue in this manner.

3. Defects can cancel each other out for some cases, giving the appearance of a successful test when in fact there were defects along the path executed. In a related problem, something going right on one part of the path can actually hide other things that went wrong elsewhere. Unfortunately, the defects will show up sometime, sadly often in production. The tests must be designed in such a way that if there are any defects in the code, no matter how long the path length to an observable event (e.g. data on screen, updates to a data base, packets sent over the communications lines, report entries) the defect will propagate (i.e. be visible) to a point where it can observed.

4. Testers need to do a risk assessment of what the impact of failure would be in order to decide how much testing is enough. This risk assessment should be done at the function/feature level. Testers should do triage separating the failure exposure into groupings such as annoying vs. impairing vs. catastrophic, for example. They should then have specific quantifiable levels of testing that should be done for each level of exposure.

Determining how much testing is enough is first and foremost a business issue. For example, in the world of E-Commerce, let's consider what is at stake:

- B2B transactions topped $734 billion in 2000 and are expected to grow to $8.5 trillion by 2005 (Gartner Group) [Darwin Online - Numbers, April 5, 2001]
- In the next four years $5 trillion will be invested in e-commerce. (IDC) [Electronic Commerce World, June 2001, Forecast article entitled "Global E-Commerce Spending to Top $5 Trillion by 2005"]
- 93% of on-line users have encountered problems with B2B websites. (Motive Communications) [Darwin Online - Numbers, May 17, 2001]
- Website outages are expensive: e-bay's 22 hour outage in 1999 cost $3 to $5 million; credit card/sales authorization outages cost $2.6 million per hour; brokerage firm outages cost $6.5 million per hour. (Data Dimensions) [Testers' network, March/April 2000]

- 82% of retail transactions are not completed. 42% of customers state that web-site malfunctions were the cause. (A. T. Kearney) [Infoworld November 20, 2000, By The Numbers]
- $25 billion dollars in retail sales are lost due to the above problem. (Zona Research) [Darwin Online - Numbers, May 15, 2001]

One of the fundamental challenges for E-Commerce applications is that defects are visible by and directly impact the customer. Poor experiences with a web site can drive customers to the competition who are now only three mouse clicks away. Consider these customer loyalty statistics:

- It costs six times more to sell a new customer than an existing one.
- A 5% increase in customer loyalty can increase profits by 25% to 85%. (Quality Digest, September 2000) ["Measuring and Managing Customer Satisfaction" by Kevin Cacioppo]
- The top 50% of firms in the American Consumer Satisfaction Index have an average market value of $42.5 billion; the bottom 50% $23.2 billion (University of Michigan) [Harvard Business Review, March 2001, "High Tech the Old-Fashioned Way" by Bronwyn Fryer]

You can repeat this type of analysis for any class of application in the private and public sector. In each case the number are enormous. Just consider the following:

- 70% of all projects fail to meet their objectives - e.g., 55% to 75% of CRM projects fail to meet their objective, 70% of ERP projects fail to meet their objectives [Meta Group as reported in Infoworld, "Survival Guide" column by Bob Lewis, 29 October 2001]

Yet when you ask testers how they know they are done testing, the most common responses are:

1. We test until we are out of time and resources;
2. We test until all of the test cases we created ran successfully at least once and there are no outstanding severe defects.

I admire the honesty of the first answer. It comes from the "clean conscience" school of testing – "I did all the testing I could under the constraints management gave me, and my conscience is clear". This is especially true today where we have moved from RAD (rapid application development) to FAD (frantic application development). The obvious question that follows the second answer is how much function and code were actually tested? In the vast majority of cases, the team has no quantitative measure of their level of testing.

You need to define quantitatively and qualitatively how much testing is enough and then design tests that will ensure that criterion is met. You must do this for each type of testing: functional, performance, usability, security, etc. Given the space constraints, we will only address functional testing in this article.

DESIGNING TESTS

The basic testing steps are:

1. Define quantitative and qualitative test completion criteria
2. **Design test cases to cover the above criteria**
3. Build "executable" test cases
4. Run the test cases
5. Verify test results
6. Verify test coverage against completion criteria
7. Manage test libraries
8. Manage reported incidents/defects

This view is handy because all of the testing tools fit into one or more of these categories. No tool addresses all of them. For example, the BenderRBT test design tool addresses steps 1, 2, and 6. Playback tools address steps 3, 4, and 5.

The goal and challenge of test case design is to identify an extremely small subset of the possible combinations of data that will give you an extremely high degree of confidence that the program will always work. On the surface this might seem like an impossible task. However, engineers who test high-end integrated circuits have been doing just this for decades. In testing software we need to address the functional test completion criteria from a specification (black box) perspective and a code based (white box) perspective.

In most organizations if you gave ten different testers the same specification or code to test you would get back ten different sets of tests. This is because most tests are designed using a gut feel approach. The thoroughness of the tests depend on the tester's experience in testing, their experience in the application, their experience in the technology the application is running on, and how they were feeling the day they designed the tests. All you know for sure when all of the tests ran correctly is that the tests ran correctly. You do not know that the system is running correctly. This level of assurance only occurs when the set of tests is mathematically equal to the set of function in the specifications.

There are a number of published test completion criteria standards. The Federal Aviation Administration has 178B (1992). The code based criteria is each statement and branch executed at least once. From a requirements perspective there is no quantified completion criteria. Furthermore, 178B is now considered just a guideline. It is up to the producers of the software to decide how much testing to do. The Food and Drug Administration has 21CFR Section 820 (1997). Again requirements based testing is not quantified. Code based coverage is statement and branch. The ANSI/IEEE Std 1008-1987 is statement coverage only and aimed at unit testing.

None of these standards address the key issues that any experienced tester has run into again and again. These are: multiple defects can hide each other resulting in getting the right answer for the wrong reason; a defect might not be detected because of the long path length until an observable output is produced; some defects are only detectable if events happen in certain sequences - i.e., position dependent and sequence dependent events. The test completion criteria and the test cases designed to meet that criteria must take these issues into account in order to meet the quality required of today's systems. We will address each of these issues in this paper.

RIGOROUS BLACK BOX / SPECIFICATION BASED TEST CASE DESIGN CRITERIA

Engineers testing hardware logic use rigorous algorithms to identify the necessary test cases. Since rules are rules it should not matter, from a black box perspective, whether those rules have been implemented in hardware, firmware, or software. 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 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.

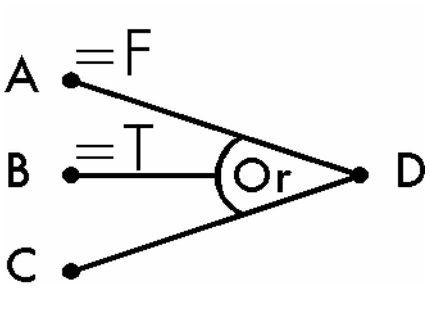
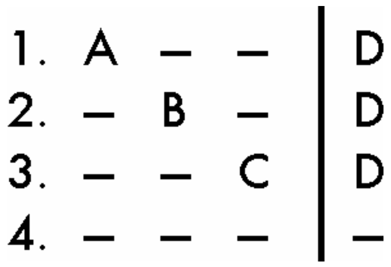


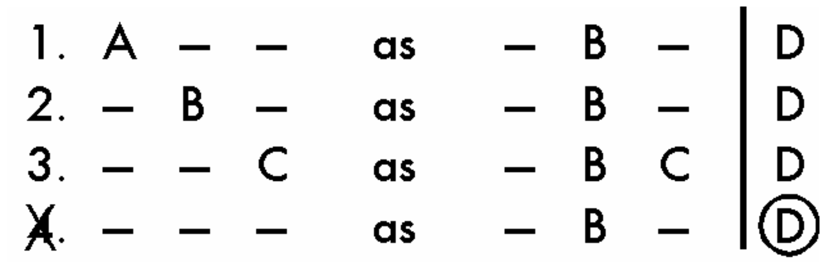
Figure 1 - Simple "OR" Function With Two Defects



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.



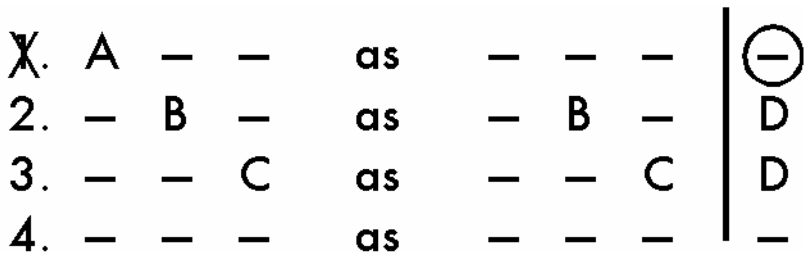
1.	A	–	–	as	–	B	–	D
2.	–	B	–	as	–	B	–	D
3.	–	–	C	as	–	B	C	D
~~4~~.	–	–	–	as	–	B	–	Ⓓ

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.



~~1~~.	A	–	–	as	–	–	–	⊖
2.	–	B	–	as	–	B	–	D
3.	–	–	C	as	–	–	C	D
4.	–	–	–	as	–	–	–	–

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. 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.

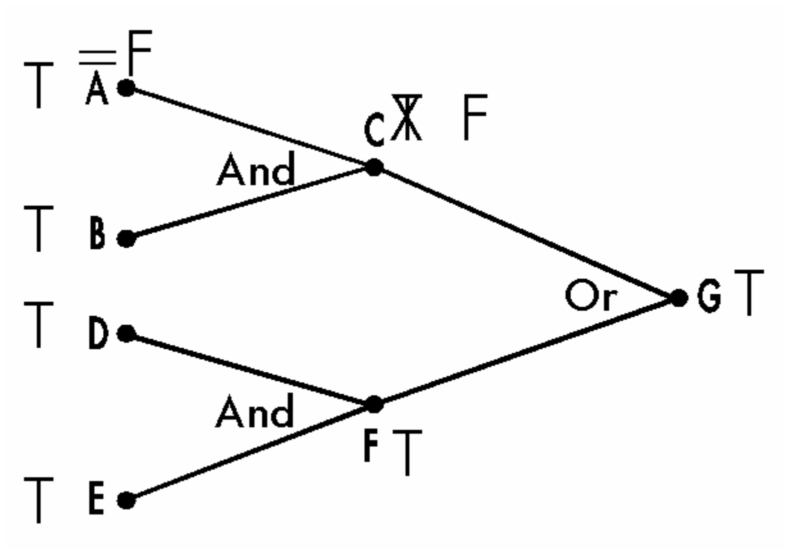


Figure 5 - Variable "A" Stuck False Defect Not Observable

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, 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.

These algorithms result in small, highly optimized test libraries relative to the complexity of the functions being tested. For example, a fairly typical screen had a theoretical maximum number of tests of 137,438,953,472. This was reduced down to 22 tests. An embedded function helping to manage the operations of a car had over 5.5 x 10^{42} possible tests which were reduced to 137. In other words, when those 137 tests ran correctly the specification based testing was done.

Clearly, this approach focuses on the decision logic. However, it also results in all of the transform rules being invoked as well. It has been successfully used to test every type of application including business applications, manufacturing applications, military applications, scientific applications, medical applications, compilers, operating systems, data base management systems, and communications software. It has been used for batch systems, real time systems and state machines. It has been applied to every technology base including embedded systems, PC's, client-server, web-based, mainframe, and super computer.

Thus, our black box test completion criteria is to **test every functional variation, fully sensitized for the observability of defects, ensuring that all tests have run successfully in a single run or set of runs with no code changes in between the test runs.**

RIGOROUS WHITE BOX / CODE BASED TEST CASE DESIGN CRITERIA

Test completion criteria for code are based on two major classifications - logic flow and data flow. In logic flow the most basic level of coverage is statement coverage - i.e., did you execute every statement at least once. This is sometimes called C0 coverage. A slightly higher level of coverage includes the C0 level but adds to it branch vector coverage. It verifies that each conditional branch has been taken true and false. This is called C1 coverage. In Figure 6, 100% C1 coverage could be obtained by executing two tests: Test 1 following a path of 1, 2(true), 3, 5(true), 6, 8(true loop), 8(false); Test 2 following a path of 1, 2(false), 4, 5(false), 7, 8(true loop), 8(false).

Even though these criteria have been around for over thirty years, little code shipped today meets this level of testing. Most code today has only 30% to 50% of the statements executed before shipping into production. This is held true for applications created for internal use as well as for vendor produced products shipped to third parties.

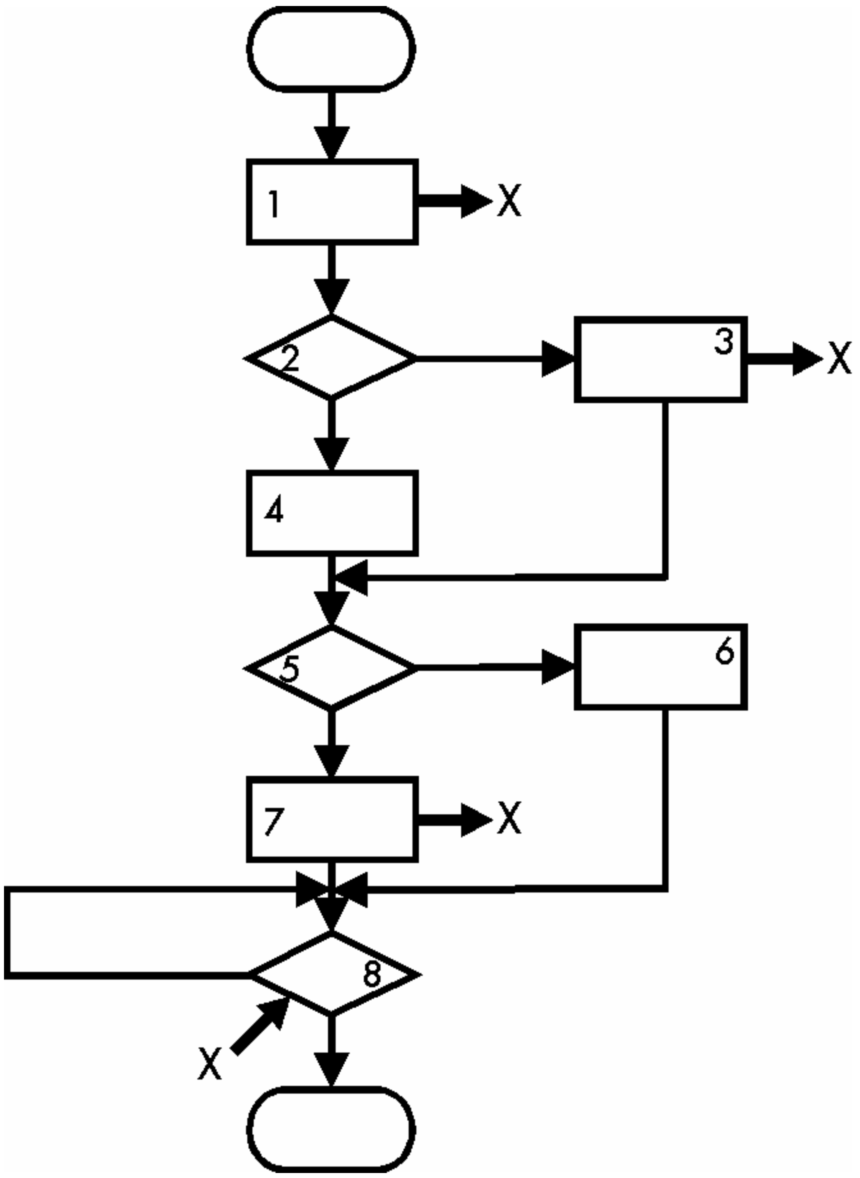


Figure 6 - Flow Chart With Data Dependencies

Data flow coverage adds a higher level of rigor to the testing process. For each variable used as input to a statement, it determines if each possible source of the data has been tested. For a data flow to exist there must be a path from the instruction which sets the data to the instruction which uses the data along which no other instruction over-writes the data. [Note: data flows are sometimes called "set-use pairs" in the literature.]

Testing to the C1 level does not guarantee that each of these data flows will be executed. In fact when you reach C1 coverage you usually still have 20% to 40% of these data flows not tested. This is a significant amount of function untested. The basic data flow coverage is called D1 and includes the C1 coverage.

Looking at Figure 6 again we see that segment "8" uses variable X to determine how many times to loop. The logic is that it loops, subtracts "1" from X, and checks to see if X is now "0". If it is not "0" it loops again; if it is "0" it terminates the loop and continues on to the next statement. X is modified by segments "1", "3", and "7". When we executed Test 1 above, X was last set at segment "3". This sets X to "10"; segment "8" therefore loops 10 times. When we executed Test 2, X was last set by segment "7". This sets X to "20"; segment "8" will loop 20 times. Remember that these two tests satisfied 100% C1 coverage. However, we never executed a path where segment "1" was the last place to set X prior to executing segment "8". We need to add Test 3 which follows the path 1, 2 (false), 4, 5(true), 6, 8. However, let us assume that segment "1" sets X to minus 1. After we loop and subtract 1, the loop control variable is now minus 2 and so on. This path causes a nearly infinite loop.

When we increased the test criteria from C1 to D1 we did find 25% more code-based defects. These also tended to be types that would have been difficult to debug. For example, many non-reproducible defects have spurious data flows at their root. Static data flow analysis actually is able to predict where many of these will occur before even running the tests.

There is another interesting insight from data flow analysis about test suite design – i.e., packaging the test cases into sets for execution. In order to test a given data flow, the test case that includes it might have to be in a certain position in the test suite. The most common requirement is for the test to be the first one executed. It is not unusual to have multiple data flows each requiring their test to be first. This requires the test suite to be broken into smaller execution packets each with the right tests first. I have even seen data flows that would be tested only if included in a test that happened to be in just the sixteenth through the nineteenth position in the test suite. It would work fine if placed anywhere from the first to the fifteenth position or in the twentieth position or later.

Yet another interesting by product is that in order to execute certain data flows, multiple transactions/events must be executed in a particular sequence. What happens is the first transaction/event modifies a variable which is then used by the second transaction/event.

D1 level testing is key to testing maintenance changes since the big issue is the ripple effect of changes producing unexpected results. The impact of any maintenance change can be calculated using data flow analysis. There are rules for calculating the impact of adding code, deleting code, and changing the logic structure. In each case the new data flows are identified, deleted data flows are identified, and the impact on the interface is calculated.

Testing the asynchronous interaction between two or more processes with shared data is also almost totally a data flow issue (e.g., multiple applications running concurrently on a shared database; communications network software; multi-processor architectures).

Thus our white box test completion criterion is:

At minimum **execute every statement and conditional branch true and false at least once (C1).**

Or for application critical, mission critical, and safety critical code **execute the C1 coverage plus every first order data flow at least once (D1).**

You do not have to decide ahead of time whether you need to go to D1. One approach is to test to the C1 level and then drill "test wells" to the D1 level. That is, take selected modules and increase the level of testing to D1. If the additional defects found are not proportional to the effort required you might decide that C1 was sufficient this time. However, if you are finding a substantial number of defects you might want to expand the scope of D1 testing.

SUMMARY

The test completion criteria we have defined here are not unique to any class of application. We have applied them to embedded software up through software running on super computers. We have applied them to business, scientific, military, and government applications. The key issue is doing sufficient testing to mitigate the risks that would be incurred if defects occurred in the application. Without quantitative test completion criteria you cannot make reasoned decisions as to whether or not the testing to date is sufficient to deploy the software. In cases where the software development and test have been outsourced, the test criteria must be part of the contract. Otherwise, how do you know the vendor has completed their testing? The key is to be practical in defining and applying these criteria.

Using the black box and white criteria on the same project does not mean you have two test efforts. You first design tests from the specifications using the black box criteria. These tests should be designed before the code is written. You then run these tests against the code and measure how much of the white box criteria are also met by these tests. You then supplement the test library to complete the white box criteria. The black box tests should execute about 90% of the C1 level coverage and about 70% of the D1 level.

Personally, as a tester, I find having quantifiable test completion status to report fundamentally changes the relationship of the tester to management. All too often testers are managed by the "pounds per square inch" school of management. That is, management keeps applying more and more pressure to get testing completed. With test status numbers, which can easily be put on a spreadsheet, we can report that we are currently 62.9% specification based tested and 55.7% code based tested. If management still wants to ship the application I ask them to please sign the spreadsheet showing the detailed status at ship time. These numbers allow management to make informed, reasoned decisions as to the risks in shipping the application into production.

Richard Bender has over thirty-five years experience in software. He initially was a developer but went straight and has been a tester for over thirty years. He is currently President of Bender RBT Inc, a software quality and testing consultancy. He can be reached at 518-743-8755; rbender@BenderRBT.com.