Comparing Requirements Based Testing Approaches

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

Comparing Requirements Based Testing Approaches is

© Copyright 2013, 2015, 2018 by:

Table of Contents

1. Introduction

There are numerous tools available for designing test cases from the requirements. They all use just one of two approaches. One is combinatorics techniques. These take the various variables and variable states and combine them into tests via various algorithms. The most commonly used approach is Pair-Wise Testing. The second approach is model based testing. These take a representation of the requirements such a flow chart or finite state diagram and design tests based on paths through the model. Within model based testing there is a refined approach which sensitizes the path to ensure that defects are propagated to an observable point. This approach uses Cause-Effect Graphing which is based on the same algorithms used to test high end integrated circuits.

In this paper we will first define what Requirements Based Testing is and the comparison criteria used to determine which approach is the most effective and most efficient. Each of the approaches are then measured against the criteria.

There are three fundamental issues that each technique must address. The first issue is the quality, or lack thereof, in the requirements specifications. As I write this we have just passed the 38th anniversary of Bender RBT Inc. In all those years, working with hundreds of clients on many hundreds of projects, we have only seen two testable specifications on starting work with a new client. So any process that assumes you have good specifications is fatally flawed to start with. Adding to the challenge is the adoption of agile techniques which are initiated with higher level specifications called User Stories. So the approach must also address this.

The second issue is that you cannot exhaustively test even a simple function. 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.

The third issue is the observability of defects. There are cases where two or more defects can cancel each other out resulting in the right answer for the wrong reason. There are also cases where something working correctly on one part of the path hides something broken in another part of the path. So the test algorithms need to take this into account and propagate defects to an observable point.

2. Executive Summary

Of the three basic alternatives to test design – path coverage through models, pair-wise testing, and cause-effect graphing – the graphing approach is overwhelmingly the most effective and the most efficient. It is the only approach that eliminates most of the defects in the requirements prior to the start of coding. It is the only approach to testing that factors in the observability of defects in designing tests – i.e. it ensures that any defect will show up at an observable point when the tests are run.

Tests designed via graphing give you, at minimum, twice the coverage with half the number of tests as compared to other approaches. This results in huge savings in the effort to build the executable tests, run the tests, verify the test results, and maintain the tests from release to release.

Graphing is also the only viable approach to providing strong support for agile projects using Acceptance Test Driven Development.

These tables summarize the findings in the sections that follow.

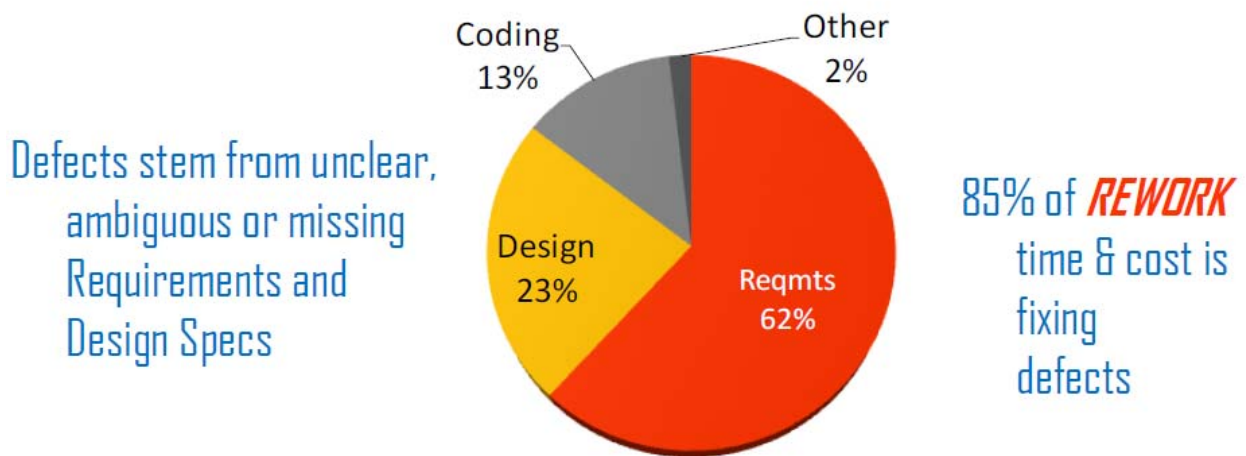| Validate Requirements	Cause-Effect Graphing	Path Coverage	Pair-Wise
Flexible Requirements Format	X		X
Ambiguity Eliminated	X		
Implicit Requirements Clarified	X		
Sequencing Clarified	X	X	
Concurrency Clarified	X		
Logical Relationships Clarified	X	x	
Intra-Functional Logical Consistency Verified	X	x	
Inter-Functional Logical Consistency Verified	X		

Test Design	Cause-Effect Graphing	Path Coverage	Pair-Wise
Expected Results Included	X	x	
Boundary Constraints Factored In	X		x
Strong Negative Testing	X		x
Observability of Defects	X		
Reduce Number of Tests	X	x	x
Test Coverage	100%	<50%	<50%
Can Support Agile Projects	X		

3. Defining Requirements Based Testing

Requirements-based testing (RBT) consists of two basic tasks: first, validating that the requirements are correct, complete, unambiguous, and logically consistent; and second, designing a necessary and sufficient (from a black box perspective) set of test cases from those requirements to ensure that the design and code fully meet those requirements.

Numerous studies have shown that the majority of defects have their root cause in poor requirements. They also show that the vast majority of scrap and rework costs can be traced back to poor requirements.

Where defects come from:

Coding 13%

Other 2%

Defects stem from unclear, ambiguous or missing Requirements and Design Specs

Design 23%

Reqmts 62%

85% of REWORK time & cost is fixing defects

*Capers Jones, Applied Software Measurement 2nd Ed.

Numerous studies also show that the earlier you find the defect the cheaper it is.

Phase In Which Found	Cost Ratio
Requirements	1
Design	3-6
Coding	10
Development Testing	15-40
Acceptance Testing	30-70
Operation	40-1000

The net is that there is no way to significantly improve software quality and productivity without first and foremost address the quality of the requirements.

The Bender RBT Inc. process for Requirements Based Testing consists of twelve steps:

1)   Validate requirements against objectives
2)   Apply scenarios against requirements
3)   Perform initial ambiguity review
4)   Perform domain expert reviews
5)   Create cause-effect graph
6)   Logical consistency check by BenderRBT
7)   Review of test cases by specification writers
8)   Review of test cases by users
9)   Review of test cases by developers
10) Walk test cases through design
11) Walk test cases through code
12) Execute test cases against code

(More detail on this can be found in the paper "Bender-Requirements Based Testing Process Overview" on our website.)

Some of these steps, such as steps 1 through 4, can be applied independently of the test design approach. So they will not be factored into the comparison criteria. Steps 5 and 6 are tightly coupled to Cause-Effect Graphing. However, the other test design approaches need to have something equivalent to address the kinds of issues uncovered in the requirements by these steps. The effectiveness of steps 7 through 12 are greatly affected by the completeness of the test descriptions and the robustness of the coverage. So again this will be addressed in the comparisons.

4. Comparison Criteria

In order to design test cases you need to be able to do the following things:

- Identify all of the variables
- Resolve aliases within/across processes
- Identify the possible states of the variables
    - Both positive and negative states
- Know which variables are mandatory versus optional
- Identify all of the preconditions
    - Based on the physical structure of the data
    - Based on the post conditions of prior functions
- Understand the precedence relationships
- Understand concurrency
- Know which variables are observable
- Identify implicit information and get it clarified
- Identify the transforms
- Identify the expected results

From this we derive our criteria for comparing the various RBT approaches:

1. Allow for flexible requirements formats.
2. Eliminate ambiguities.
3. Identify and clarify implicit requirements.
4. Clarify the sequence of steps.
5. Clarify the inherent concurrency of the steps.
6. Clarify the logical relationships.
7. Verify intra-functional logical consistency.
8. Verify inter-functional logical consistency.
9. Determine the expected results for each test.
10. Factor in the boundary constraints in designing tests.
11. Do strong negative testing.
12. Ensure the observability of defects.
13. Reduce the impossibly large number of theoretically possible tests down to a reasonable number.
14. Ensure high test coverage.
15. Be able to support Agile projects.

Allow Flexible Requirements Formats

Testers cannot, nor should they, dictate the formats in which requirements are written. Requirements have a great diversity in the target audiences who have a vested interest in the contents – users, product managers, program managers, analysts, designers, coders, testers,

technical writers, etc. So the test design process must be able to handle whatever format the requirements are written in.

Eliminate Ambiguities

Requirements are written primarily in natural languages – e.g. English, German, Chinese. All natural languages are inherently ambiguous. So these ambiguities must be identified and eliminated as part of the RBT process.

Identify and Clarify Implicit Requirements

The hardest thing to test for is the missing requirement. If you look carefully at the list of "enhancements" from release to release you will find that many of these are actually missed requirements from prior releases. In other words these are not enhancements, they are defects. They were within the scope of functions implemented previously and these cases/factors were missed. The RBT process should help find these.

Clarify the Sequence of Steps

When you verify the results from running a test you need to ensure that you got the right outputs and you got them in the correct order. That implies you know the desired sequences of events.

Clarify the Inherent Concurrency of the Steps

In testing you want to maximize coverage while minimizing the number of tests. If multiple steps can happen concurrently, then these can often be put into the same tests. For example, in editing a set of data is it a full screen edit or a field by field edit? This issue is related to the above sequencing issue.

Clarify the Logical Relationships

The test design approach should factor whether the decision logic is driven by an OR versus an AND, etc. In code based testing there is a coverage criteria called MCDC (modified condition/decision coverage). In this approach you have specific test rules for each logical operator. For example, in testing an OR you take each object true by itself and the all false case. In testing an AND you take each object false by itself and the all true case. This criteria is based on the path sensitizing algorithms in hardware logic testing. We will cover this in the Cause-Effect Graphing section.

Verify Intra-Functional Logical Consistency

Some requirements are illogical within a single function. For example, if you have a path which is taken only when A and B are both true and later in that path have a decision which is taken only when one or both of A, B are false, this is broken. You can never execute that portion of the path. These issues need to be identified.

Verify Inter-Functional Logical Consistency

You need to verify that the rules within a process are logically consistent with the post conditions of the upstream functions and/or the physical structure of the data. This is a very nasty issue and cannot be reliably done during machine based testing. These issues need to be identified as part of the test design process. For example, let us assume that the post condition of an upstream function ensures that one and only one of A, B, and C is true (e.g. it is the result of selecting an option on a pull down menu where there is a default already set). If there is a rule in a downstream function that is driven by A, B, and C all being false that rule can never be executed.

Determine the Expected Results for Each Test

Testing, by definition, is comparing the expected results against the observed results. That implies that you do not have a test fully defined unless it includes the expected results. Tools that do not include the expected results require the tester to manually determine that for each test. That is a lot of work in practice. What really happens is the testers do not do this step. They just run the tests and see if the outputs look "reasonable". The result is that the test cases find defects that tester missed since systems are too complicated to rely on a reasonableness check in verifying the test results.

Factor in the Boundary Constraints in Designing Tests

There are five basic Constraints: Exclusive (at most one is true), One (one and only one is true), Inclusive (at least one is true), Requires (this object's state requires that other objects are in specific states), and Masks (if this object is in the specific state then the other objects it Masks either do not exist or they will be ignored in processing). If there is a Constraint then it must be impossible for the object states to violate them due to either the physical structure of the data or the post conditions of upstream functions. The test design process must take into account these Constraints while still doing strong negative testing.

Factoring in Constraints is tightly coupled with the ability to identify inter and intra functional logical consistency problems.

Do Strong Negative Testing

Most requirements focus on the happy path or go right path. Often missing are the rules for exception handling and alternative paths. When these are not addressed properly in the requirements the results are major problems in data integrity, event/transaction integrity, usability, and even security exposures. So the test design approach must include strong negative testing.

Ensure the Observability of Defects

The primary purpose of testing is to identify any and all of the defects. How would you know you have a defect? When the observed result does not match the expected results you know something is wrong. However, in most systems you have a very limited set of things you can observe – data on the screen, packets sent over the communications lines, data on reports, and updates to data bases. In some systems you also can observe the movement of mechanical objects. However, most of the processing in software is happening internally and only indirectly observable. Think about how long your path lengths usually are before you reach a step that produces something you can see externally. The result is that defects, even in paths you executed, might not show up externally. So the test design approach must ensure that any defects will show up an observable step.

Reduce the Impossibly Large Number of Tests

You cannot exhaustively test all possible combinations and permutations of the data. So the test design algorithms must be very smart in reducing this large number down to what can be built and run in the time and resource constraints any project has.

There is a related issue here regarding the effort required for the various tasks in testing. Our experience is that if it takes 1X amount of effort to design a set of tests it will take 3X to 5X that effort to implement the executable tests. That includes creating the automated scripts and provisioning the test data bases. Then you have the effort to run the tests, review the test results, and maintain the tests. So minimizing the number of tests has major implications on the total testing effort. It should be noted that each of the alternative approaches to test design take about the same amount of effort. You just spend your time differently.

Ensure High Test Coverage

Most organizations have a test completion criteria that states that when all of the tests they have designed and built have run correctly at least once and there are no outstanding severity 1 or severity 2 defects, then they are ready to deploy the system. However, when you ask what their functional and code coverage numbers/percentages are they have no answer. So the functional coverage on the RBT side should be based on a rigorous test completion criteria. Since you cannot test 100% of any system with just black box tests, the white box coverage should also be

very high from the RBT tests. That means there should only be a 10% or 20% supplement to the set of tests to complete the code coverage.

Be Able to Support Agile Projects

At least half of the projects we work on these days are agile. As such you do not have detailed requirements from which to derive tests. You have higher level User Stories. The RBT process must be able to ensure that at the end of a release or even a sprint that you can prove the set of specifications, the set of code, and the set of tests are provably all in sync.

5. Path Coverage Through Models

Path coverage methods/tools design a set of tests such that they execute each step in the process at least once and take each conditional branch each way at least once. They do not try to create a test for each unique path from the beginning to the end of the process since the number of tests quickly gets astronomical. It is a "track in the sand" approach.
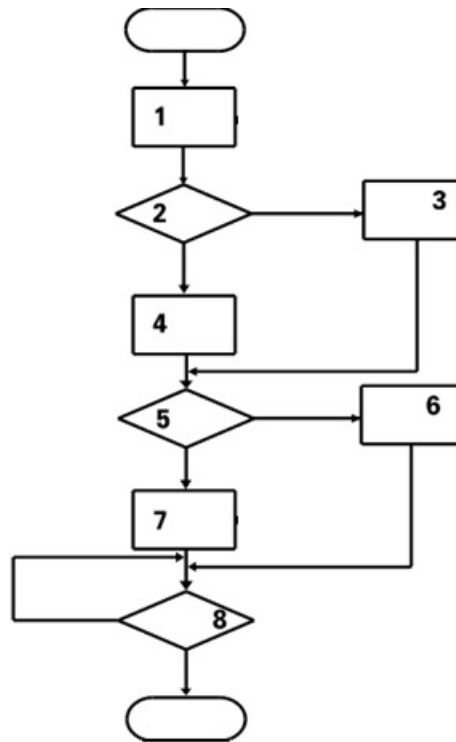
For example, in the above flow chart you might get one path that tests the sequence 1, 2 false, 4, 5 false, 7, and 8 true. The second test would follow the sequence 1, 2 true, 3, 5 true, 6, and 8 false. These two tests cover every step and each branch vector at least once. Similar approaches are used for other requirements formats.

Allow for flexible requirements formats

These tools require that requirements be written in the desired format. The alternative is to rewrite the specification in the format the tool needs.

Eliminate ambiguities

These tools assume the requirements are clear.

Identify and clarify implicit requirements

These tools assume that the requirements are complete. They only design tests for what is in the requirement.

Clarify the sequence of steps

The requirements format generally makes the sequence of steps clear.

Clarify the inherent concurrency of the steps

These tools do not take into account or even recognize which steps could happen concurrently. They follow one path at a time. So you would get more tests than you would if you allowed multiple concurrent paths to be packaged into the same test.

Clarify the logical relationships

Most of these tools do not differentiate between an AND, OR, NAND, NOR, XOR, or XNOR. So they do not generate MCDC like tests. They just take each branch vector true or false once. I have seen one tool though that did factor in the logical operators.

Verify intra-functional logical consistency

Since most of these tools do not factor in the logical relationships or the Constraints they cannot address this issue. I did see one tool that did recognize intra-functional logical inconsistencies but only in regards to the logical operators. It did not factor in Constraints which is the root cause of the vast majority of the problems in this area.

Verify inter-functional logical consistency

None of these tools address inter-functional logical consistency. This is a huge gap in validating the requirements.

Determine the expected results for each test

Most of these tools do not include the expect results as part of their tests. Therefore the tester has to manually complete the test definition. I did see one tool that did include the expected results but this is uncommon in this category.

Factor in the boundary constraints in designing tests

None of these tools factor in the boundary constraints. This is a critical issue since they create tests that are physically impossible to build or do not test the intended path when executed. Often the tester will toss some of these tests out. However, portions of the test may be legitimate and covering something not covered in any other test. So at minimum the tester has a lot of manual rework to do in finalizing the test design. Worse case is that coverage is significantly lower than assumed.

Do strong negative testing

These tools tend to be weak in negative testing since requirements are weak in this area. The tests follow only the paths that exist in the model. So if the exception paths are not documented you do not get negative tests.

Ensure the observability of defects

None of these tools factor in which steps are observable or not into test design. They do not propagate defects to observable points. This is a huge weakness since that is the whole purpose of testing.

Reduce the impossibly large number of theoretically possible tests down to a reasonable number

All of these tools reduce the number of tests to a reasonably sized set.

Ensure high test coverage

These tools define functional coverage based on the "track in the sand" approach described above. That is a fairly weak criteria. They also do not tend to get high code coverage, especially when MCDC coverage is required.

Be able to support Agile projects

They cannot support agile projects since they require that detailed specifications in their format have been written. In most agile projects the detailed specifications are written late after coding and most testing is done. This is especially problematic when using the Acceptance Test Driven Development form of agile.

In agile the test cases are often used as a surrogate for the detailed requirements. That means they have to be available in a timely manner for review by the users / domain experts. It also means that the tests have to include the expected results in order to be useful for this task. These tools fail on both issues.

6. Pair Wise and Other Combinatorics Techniques

Pair wise testing is a test design technique that came out hardware configuration testing. It was originally developed by Genichi Taguchi in the 1950's. The goal is to be able to find certain classes of defects caused by how parts behave together. There are three levels of fault detection:

Single Mode Faults - Single mode faults occur only due to one parameter.

Double Mode Fault - Double mode fault is caused by the two specific parameters values interacting together. Such an interaction is a harmful interaction between interacting parameters.

Multimode Faults - If more than two interacting components produce the consistent erroneous output, then it is a multimode fault. Orthogonal array detects the multimode faults.

In pair wise testing you define a set of variables. For each variable you define a set of states. You then combine each variable's states with the other variables' states two at a time (thus the term pair wise). For example let's say:

Variable A has 3 states
Variable B has 2 states
Variable C has 4 states

The number of pairs would be:

A X B =   6 pairs
A X C = 12 pairs
B X C =   8 pairs

The total number of pairs = 26.
The minimum of tests = 12 (the largest number of pairs for a join – i.e. the A X C set of pairs).

There are two approaches to pair wise test design – optimized pairs and orthogonal pairs. In optimized pairs the goal is to minimize the number of tests required to cover all pairs at least once. This focuses on the Double Fault Mode. No one has created an algorithm that generates the provably minimum set but there are a number of solid algorithms available.

In orthogonal pairs the goal is to detect Multimode Faults so the test set is larger. For hardware configuration testing it was found that the orthogonal pairs approach was much better at finding faults. In software the main concern is minimizing the number of tests so optimized pairs is most often used.

Some tools will also generate tests based on the states of three variables at a time or four or more. As you go up above pairs based on two variables the number of tests goes up

exponentially. It quickly exceeds what is a viable number of tests for no significant gain in finding defects.

[Note: In the BenderRBT Test Design tool we have test design engines for Cause-Effect Graphing and for Pair Wise testing. For Pair Wise testing there are two alternative engines for optimized pairs (one is faster but produces a few more tests for large problems; the other is a little slower but produces fewer tests for large problems). There is also an Orthogonal Pairs engine. We have used the Orthogonal Pairs engine to design configuration tests. This has proved very effective as it has in hardware configuration testing.]

Allow for flexible requirements formats

Pair wise tools do not need the requirements in any special format. The tester goes through the specifications and identifies each variable and the possible states for each variable. From that information the pairs and tests are generated.

Eliminate ambiguities

Pair wise testing is only focused on the variables/states so little is usually done to identify and eliminate ambiguity in the overall specification.

Identify and clarify implicit requirements

Pair wise does nothing to ferret out missing/implicit requirements.

Clarify the sequence of steps

Pair wise does not factor in the sequence of processing. So the tester needs to adjust any tests generated to ensure the proper sequencing. This can be a lot of work.

Clarify the inherent concurrency of the steps

Pair wise does not factor in concurrency in how pairs are packaged into tests.

Clarify the logical relationships

The logical relations play no part in designing the tests.

Verify intra-functional logical consistency

Since the logical relationships are not factored in there is no way that pair wise testing can identify intra-functional logical inconsistencies.

Verify inter-functional logical consistency

Since the logical relationships and the boundary constraints are not factored in together there is no way that pair wise testing can identify inter-functional logical inconsistencies.

Determine the expected results for each test

A major weakness in pair wise testing is that the tools only identify the input combinations to test. They do not ever include the expected results. The tester must manually figure this out for each test. This is a huge amount of work and a major weakness in pair wise testing.

Factor in the boundary constraints in designing tests

The tools rarely include the ability to define constraints to the pair wise test engines. You need to define them variable to variable, variable to variable/state, and variable/state to variable/state. The only tool that can do this is the BenderRBT Test Design tool in the Quick Design component which support various pair wise test design engines.

Do strong negative testing

Pair wise testing is usually weak in this area. In order to do this in pair wise testing you need to include invalid states in your list for each variable. However, in practice if one of the invalid variable/states is included in a test the logic will often terminate early. Therefore the processing never reaches the point where the valid variable/states merged into a test with the invalid one would be used. So you actually do not test what you thought you were testing. This can be addressed by using the Mask constraint – e.g. if Variable A is at State X (invalid), it will mask any of the States for Variables B, C, and D. Then the tool would not merge the states of those three variables into any test where A is set to X. Only the BenderRBT tool addresses this issue. Again, this is a major weakness of most pair wise tools.

Ensure the observability of defects

The pair wise algorithms do not factor in the outputs at all, let alone which ones are observable. So this is a huge weakness in pair wise testing.

19

Reduce the impossibly large number of theoretically possible tests down to a reasonable number

The number of tests is greatly reduced from the upper limit of all combinations of variables/states. However, as the number of variables goes up and especially as the number of states for each variable goes up the number of tests goes up. Pair wise will generally produce more tests than any of the techniques for the same problem. However, this increased number of tests does not inherently increase the coverage, especially as compared to Cause-Effect Graphing.

Ensure high test coverage

Coverage is defined based on the double mode fault premise. This is actually fairly weak in software. It also generally leads to the lowest code coverage as well. That means that the supplemental white box testing effort is larger for this approach than for the others.

Be able to support Agile projects

Pair wise testing is very weak for agile projects. In order for the tests to be useful for user reviews as a surrogate for the requirements they need to include the expected results. This information is not available until the tester figures this out which is often not until the code is done. You also get a lot of effectively redundant tests making the review process more time and labor intensive.

7. Cause-Effect Graphing, Including the Full RBT Process

Cause-Effect Graphing is another technique that was adapted from hardware testing. However, unlike pair-wise testing which was based on configuration testing, Cause-Effect Graphing is based on how high end integrated circuits (IC's) are function tested. It was originally developed by William Elmendorf of IBM in 1970. He noticed that the software had a high residual defect rate after all of the testing was done. He also noticed that IC's rarely had any functional defects. Today, and this has held fairly steady for many years, the software residual defect rate is about 5 defects per 1,000 lines of executable code. In hardware it is less than 1 defect per billion gates. The laptop on which I am writing this paper contains an Intel i7 chip set with 40 billion gates. I have had this machine for two and a half years. To my knowledge there has never been a functional defect reported on it. If it was tested like most organizations test software it would have been delivered with 200 million functional defects. I don't think they would have sold many of them.

The question he asked was could we adapt the hardware logic testing rules over to test software. The basic algorithms used in hardware are the path sensitizing algorithms for hardware logic testing, also known as the D-algorithms. This looked promising since rules are rules are rules. Why should a rule work when it is implemented in hardware and not work as well in software? The difference is not in the product, it is in the process used to create and test the product. Those creating IC's write requirements. They also code, using a Logic Design Language which compiles to a circuit design. They then test using the D-algorithms.

If you look at all functional requirements they are composed of just two types of rules – decision rules and transform rules. If you model just the decisions you will unavoidably invoke all of the transforms. So Cause-Effect Graphing is the very first model based testing approach.

Let's say you have a set of rules:

1. If A or B, then C.
2. If D or E, then F.
3. If C and F, then G.

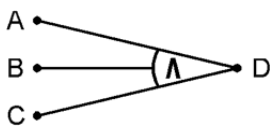The tester translates this into a Cause-Effect Graph:
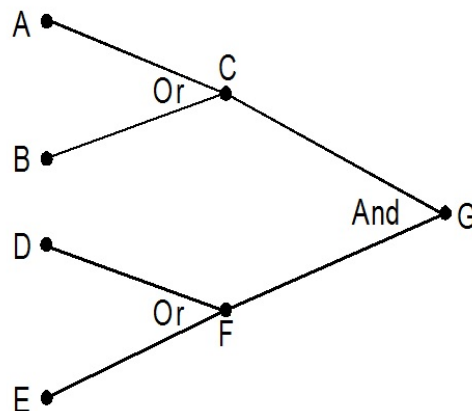
Once it is in this format then it looks like a circuit and the D-algorithms can be applied. Actually they only serve as a base for the algorithms used to design tests. For software we had to make significant extensions to the algorithms to cover software specific issues, but the underlying principals are the same.
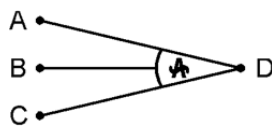
The focus of the Cause-Effect Graphing based test design is the observability of defects. There are well defined rules for which test variations are required for type of logical operator.

Functional Variations

AND (Λ)　　　　　　　**NAND (⋀̷)**

			Λ	⋀̷
_	B	C	_	D
A	_	C	_	D
A	B	_	_	D
A	B	C	D	_

OR (V)　　　　　　　**NOR (Ⅴ̷)**

			V	Ⅴ̷
A	_	_	D	_
_	B	_	D	_
_	_	C	D	_
_	_	_	_	D

XOR (X)　　　　　　　**XNOR (✻)**

			X	✻
A	_	_	D	_
_	B	_	D	_
_	_	C	D	_
A	B	C	_	D

22
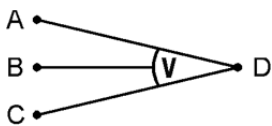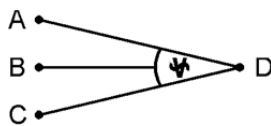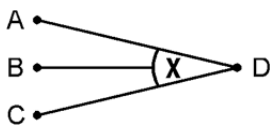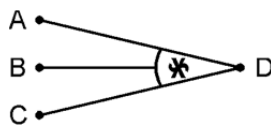
Note that the rules for defining functional variations are the basis for the MCDC level code based test completion criteria. So tests designed via graphing will automatically cover most of this criteria in the code.

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. As noted above, 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.
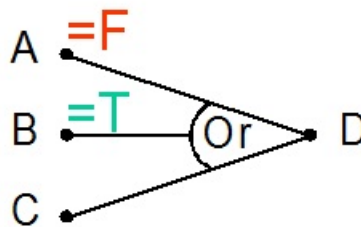

Figure 1 - Simple "OR" Function With Two Defects

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.
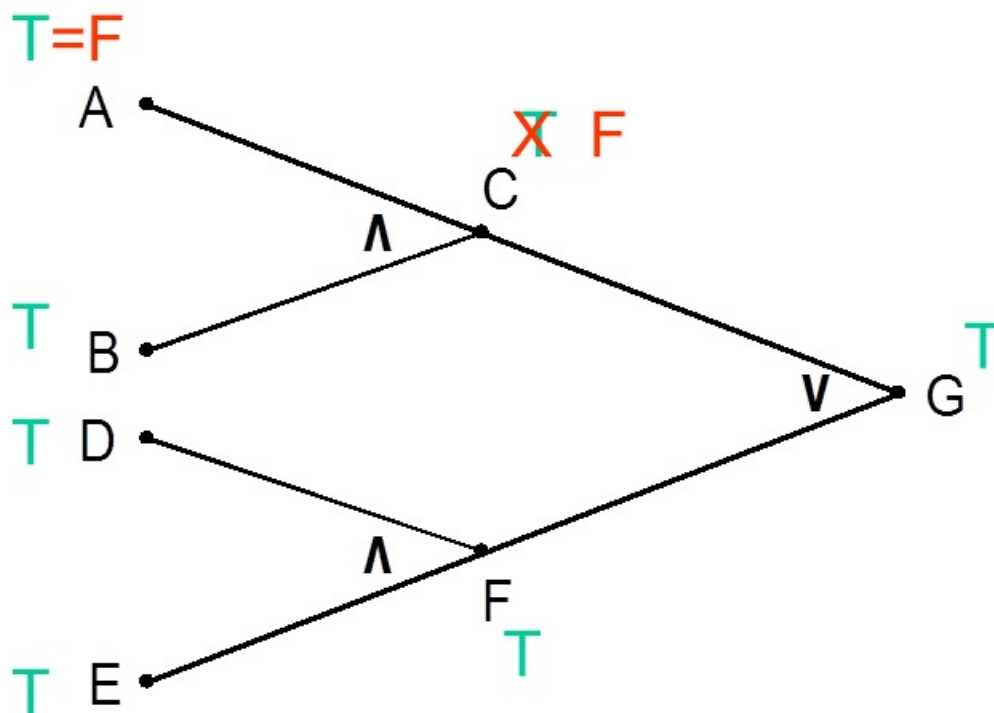

Figure 5 - Variable "A" Stuck False Defect Not Observable

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.
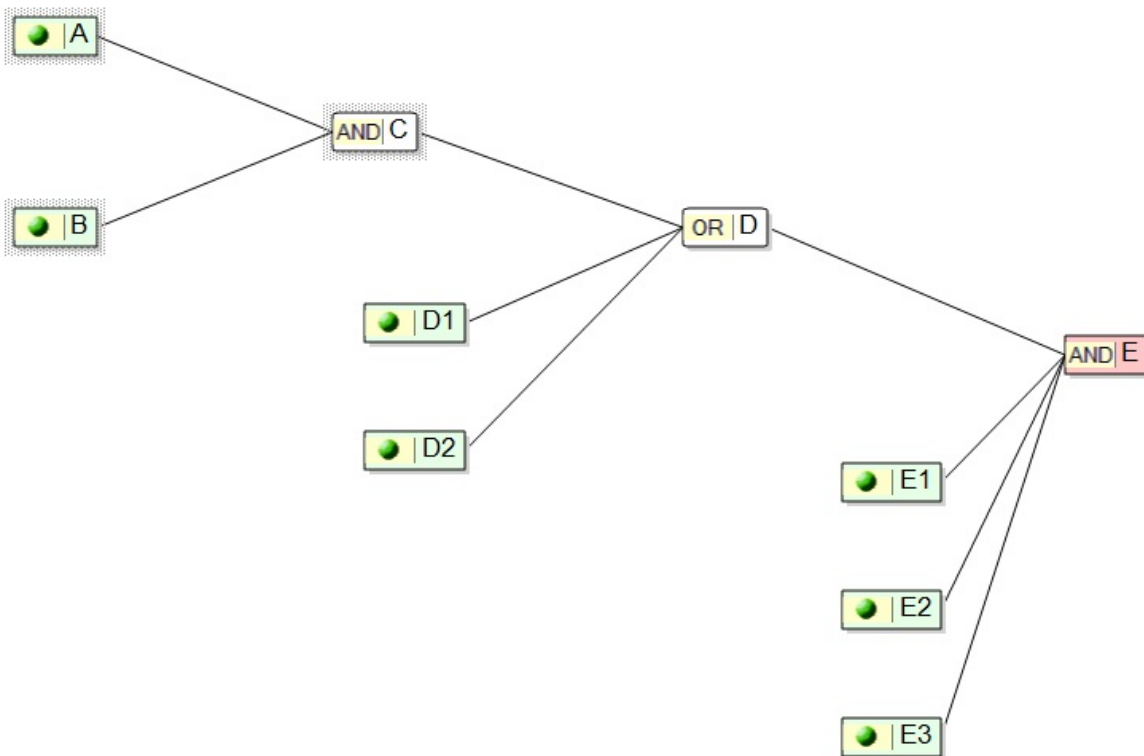

Figure 6 – Error Propagation Step 1

26

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.
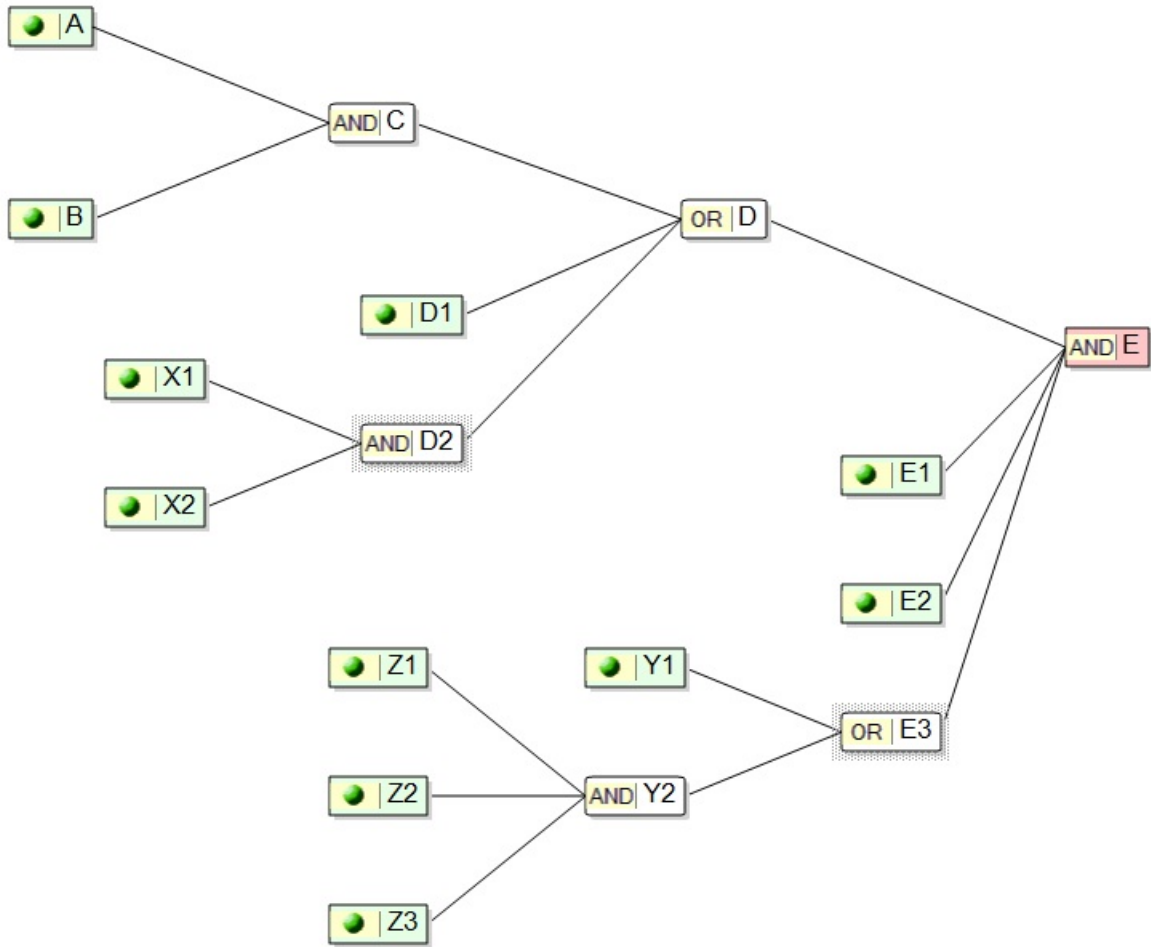

Figure 7 – Error Propagation Step 2

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.
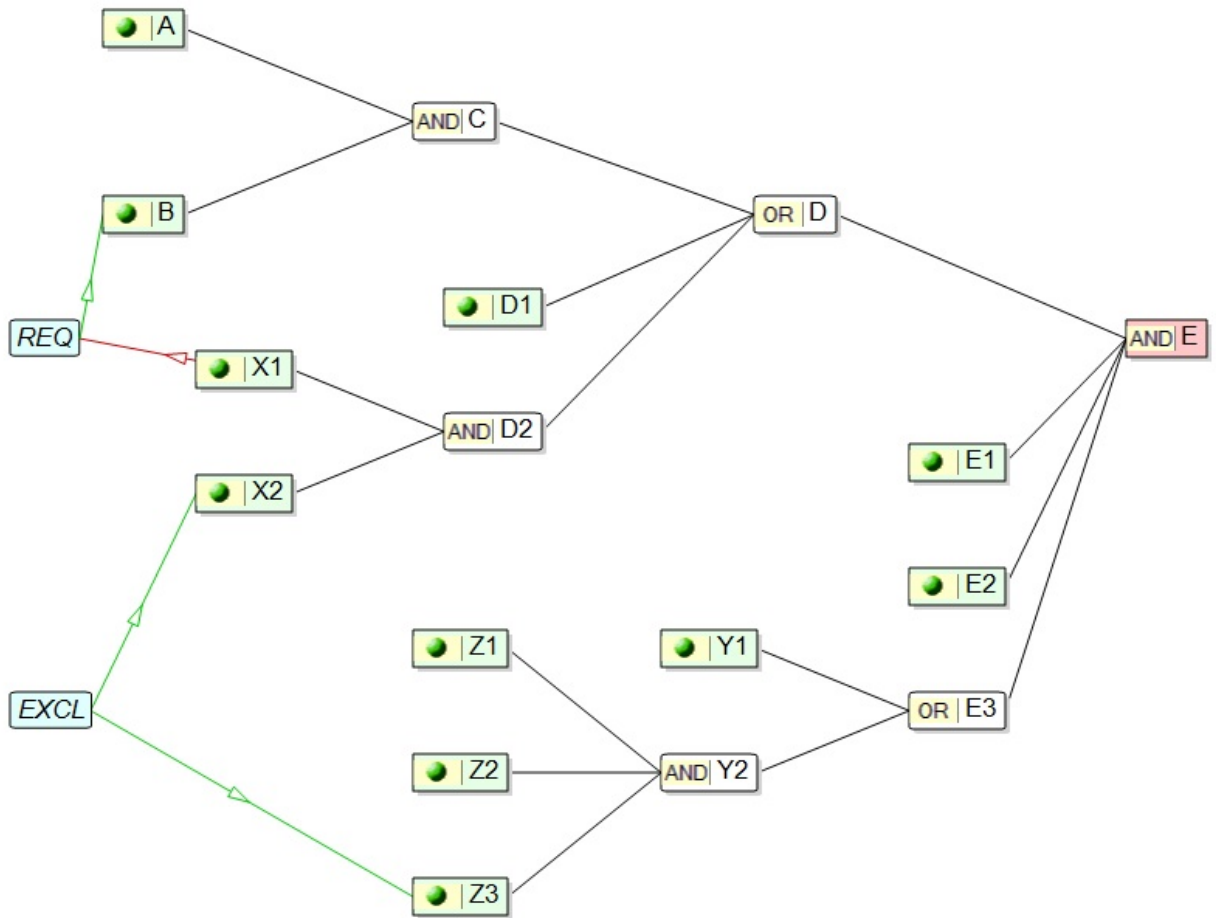
27

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.

A byproduct of the math is that some functional variations are flagged as "untestable". That means that they are valid combinations of data/states but that given the overall logic and what is naturally observable there is no way to guarantee you would see every possible defect. The solution is to insert diagnostic probe points at key steps to make things that are normally not observable to become observable. All complex hardware components have such structures. Software is the only complex product usually built without them. This is not because we do not need them. It is because the test approaches have been too primitive to identify where they are needed.

Graphing is the only test approach that addresses the critical issue of the observability of defects.

While the model focuses on the decision logic this process has been used to test applications that are decision intense as found in banking and insurance but also those that are computationally intense such as weather forecasting. In fact it has been applied to all classes of applications from embedded to super-computer based systems and everything in between.

Note that while there are many tools that support Path Coverage and Pair Wise, only the BenderRBT Test Design tool designs tests from Cause-Effect Graphs.

Allow for flexible requirements formats

Graphing can be applied to specifications written in any format.

Eliminate ambiguities

The tester either reviews the specification for ambiguity prior to creating the graph or does it concurrently, noting the ambiguities in the graph itself. The ambiguity checklist used is:

- Dangling Else
- Ambiguity Of Reference
- Scope Of Action
- Omissions
    - Causes Without Effects
    - Effects Without Causes
    - Missing Causes
    - Missing Effects
    - Complete Omissions
- Ambiguous Logical Operators
    - Or, And, Nor, Nand
    - Implicit Connectors
    - Compound Operators
- Negation
    - Scope Of Negation
    - Unnecessary Negation
    - Double Negation

- Ambiguous Statements
    - Verbs, Adverbs, Adjectives
    - Variables, Unnecessary Aliases
- Random Organization
    - Mixed Causes And Effects
    - Random Case Sequence
- Built-In Assumptions
    - Functional/Environmental Knowledge
- Ambiguous Precedence Relationships
- Implicit Cases/Variables/Actions
- Etc.
- I.E. Versus E.G.
- Temporal Ambiguity
- Boundary Ambiguity

**Identify and clarify implicit requirements**

This is a key strength the graphing process. As you draw the graph you ask two questions: if you have this input(s) do you always get this output?; if you get this output did you always have this input(s)? Let's take an example. We start off with a specification which only says "If you go through a red light you will get a ticket". That gets drawn as:
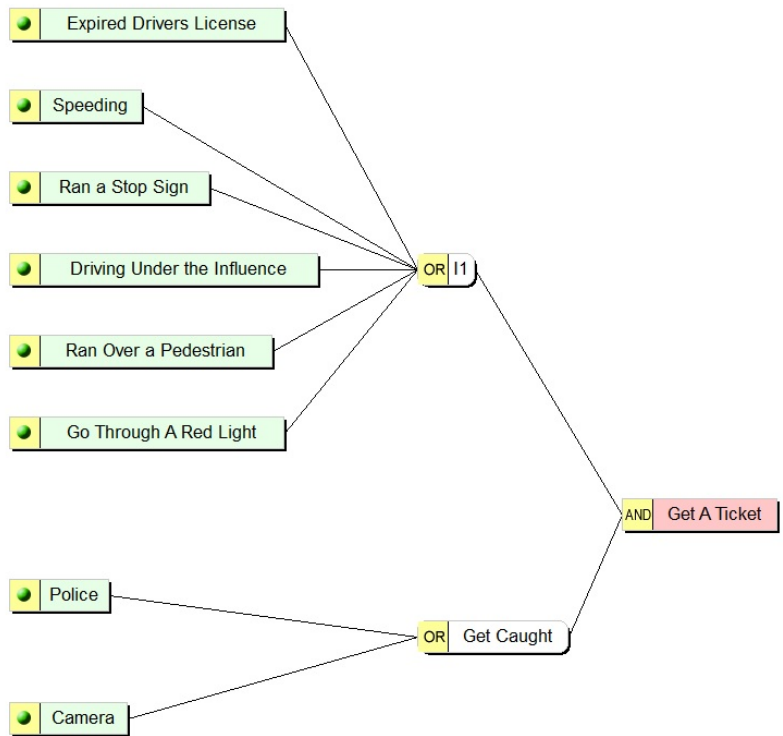
So if you go through a red light do you always get a ticket? No, you have to get caught:
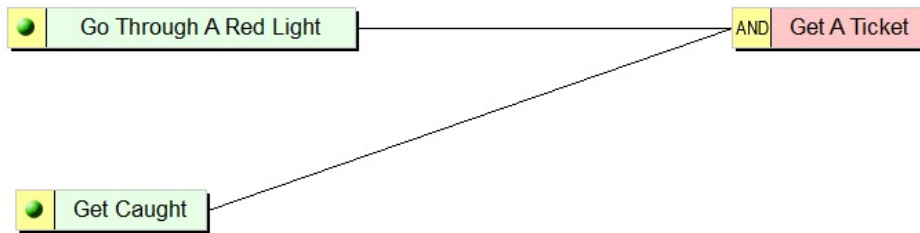

So how do you get caught? Answer: by the police of a photo system:
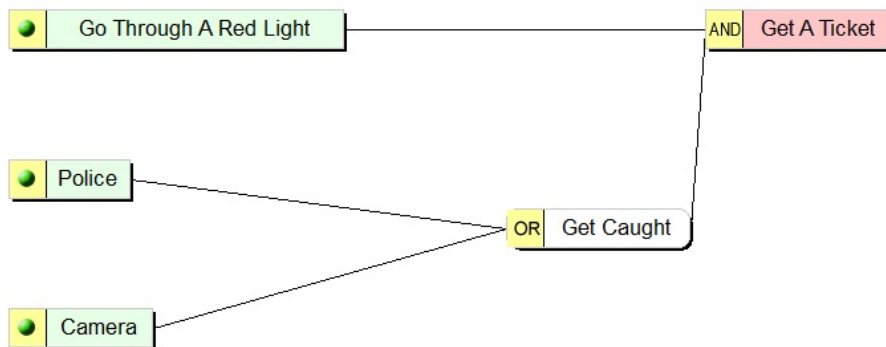
If you got a ticket did that mean you went through a red light? Answer: no, there are lots of things for which you can get a ticket:

How would a camera catch someone driving with an expired license? Answer: actually the camera can only catch people going through a red light. It requires the police to be caught for the other infractions:
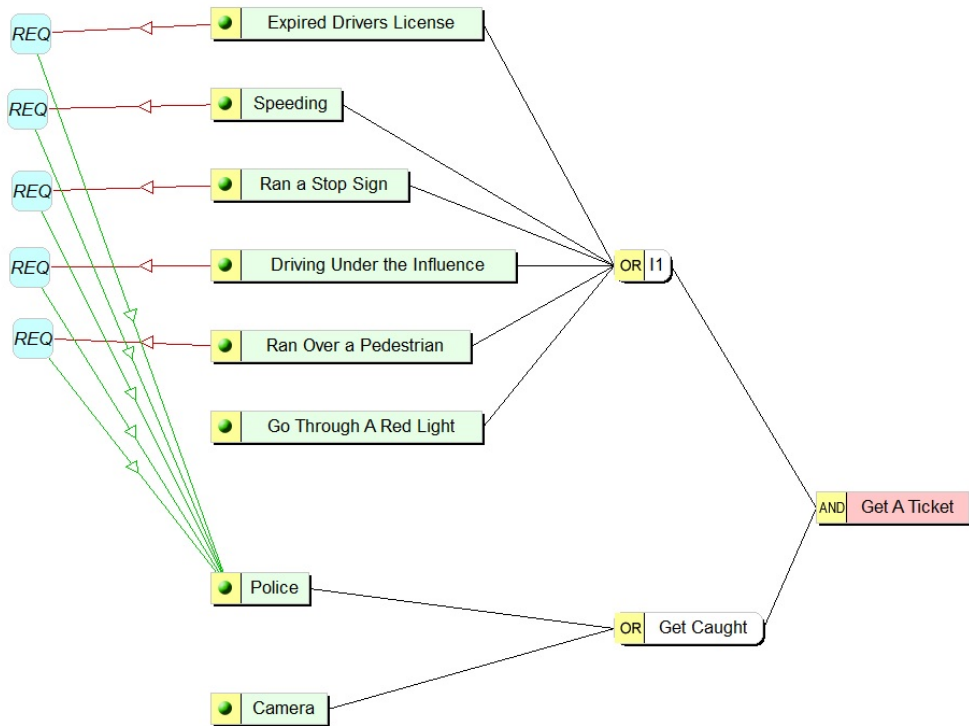
So you started with a simple one line specification which was greatly expanded and clarified in creating the graph.

Clarify the sequence of steps

Building the graph requires that you know the sequence of steps. If they are not clear in the specification graphing forces the issue to get them clarified.

Clarify the inherent concurrency of the steps

The graph clearly shows which steps can happen concurrently.

Clarify the logical relationships

The logical relations are a fundamental part of the graph.

Verify intra-functional logical consistency

When the graph is compiled into tests any intra-functional logical inconsistencies are identified. This is done by identifying which functional variations are infeasible. Some infeasible variations are legitimately infeasible – i.e. they violate a Constraint which defines that combination as being physically impossible. However, other infeasible variations should be feasible but for the problem with conflicts in the logic. Steps that are always false or always true also get flagged.

Verify inter-functional logical consistency

By supporting the full set of constraints graphing can identify any inter-functional logical inconsistencies. Again these show up as infeasible variations that the user/tester thought should be feasible and as steps always stuck at one state.

Determine the expected results for each test

Tests designed via graphing include the expected results.

Factor in the boundary constraints in designing tests

Graphing factors in the full set of Constraints in designing the tests.

Do strong negative testing

For each step/relation in the graph, tests will include what happens when it is true and what happens when it is false. So you get very strong negative testing via graphing.

Ensure the observability of defects

As discussed above this is what makes graphing unique in the world of software testing. It is the only test design approach that factors in the observability of defects. The principal is called propagating a defect to an observable point.

Reduce the impossibly large number of theoretically possible tests down to a reasonable number

Not only does graphing produce the most effective set of tests, it produces the smallest set of tests by a wide margin. This is because the hardware algorithms had to scale up to huge numbers of gates and still produce relatively small test libraries. Taking advantage of this, graphing generally will produce, at minimum, twice the coverage with half the tests. This is based on studies done by a number of major clients.

This is turn has a huge impact on the effort to build the executable tests, run the tests, verify the results, and maintain the tests. Given that you would have one fourth the number of tests for equivalent coverage and that it takes three to five X the effort to build the executable tests, you would have a 12X reduction in the effort to build the tests.

Also note that the BenderRBT Test Design tool is unique is its ability to remember what tests it designed previously. So as the rules change, as reflected in modifications to the graph, BenderRBT can first use the existing tests as a base. It tells you what you need to do modify them to bring them in sync with the modified rules. It then tells you what additional tests you need to get back to 100% coverage. All other test design tools ignore the investment made previously in implementing the tests and just give you a new set.

Ensure high test coverage

The coverage criteria for graphing is 100% of all functional variations fully sensitized for the observability of defects. It is the most rigorous software black box test criteria. It is a superset of the others. It also gives you very high code coverage. Tests designed from graphing generally cover 80% to 90%+ of the code as measured by statements and branch coverage. As previously noted is also gives you MCDC coverage for those steps in the code that correspond to steps in the specifications. You then have only a minor increment to complete the coverage gap based on design dependent / coding dependent structures.

Be able to support Agile projects

Graphing is frequently done from just User Stories. The approach is as shown above in the red light → ticket problem. The tests designed from graphing are totally non-redundant and include the expected results. The set of tests is smaller than would have been created by the other approaches so it makes user reviews of the tests more manageable. In the BenderRBT tool there is also a feature which sorts the information in the graph another way and actually generates the "as built" functional specification. This then fully supports Acceptance Test Driven Development which a process that looks like this:
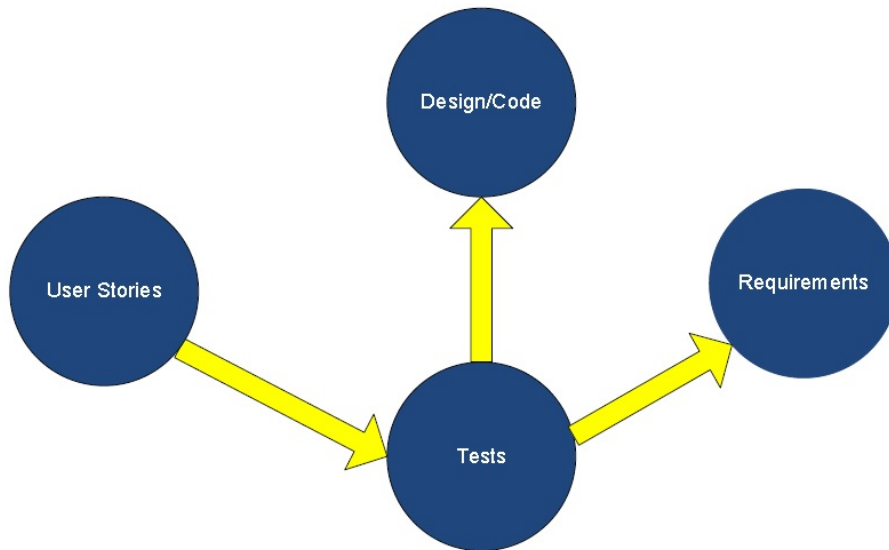
This in turn ensures that:
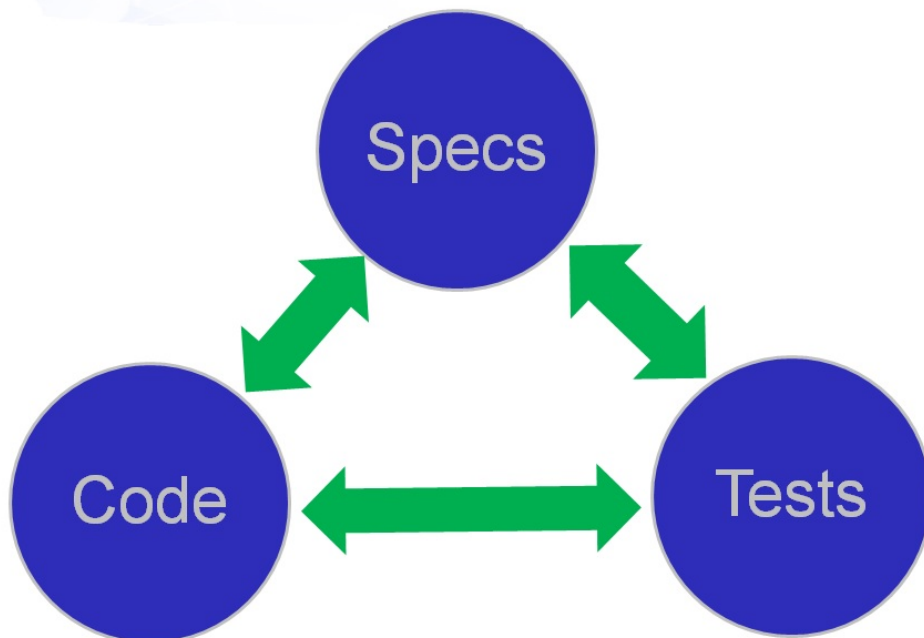
That is the set of specs and the set of code and the set of tests are provably all in sync. So graphing is by far the strongest approach for agile projects.