White Box Testing for Automotive Software

Introduction

White box testing (also known as structure-based testing) has been an important part of safety-related software standards for many years, and is now part of the ISO 26262 set of standards for electrical and/or electronic (E/E) systems within road vehicles. According to ISO 26262-6 (on software product development), white box coverage measures must be achieved when performing software testing at the software unit and software integration stages of the development lifecycle. This article will introduce the concepts and practices of white box software unit testing and demonstrate how they can be applied to automotive software that complies with the ISO 26262-6 standard.

ISO 26262 Testing Requirements

ISO 26262 highly recommends the use of 'Requirements-based test' (for both functional and non-functional requirements) at all levels of the software and systems life cycle. The testing specified by ISO 26262 is focused on ensuring that all the safety requirements are tested and that no extra functionality has been added that is not specified or needed. Thus, every requirement must have a test case and all the delivered program code should be traceable to a requirement.

The white box testing requirements in ISO 26262 support this emphasis on the testing of the requirements. If full white box coverage is not achieved by running the existing (requirements-based) test cases, then this suggests that either:

- there is dead code in the program (code that can never be reached and executed); or
- there is extra functionality in the program that was not described in the specification; or
- the current set of test cases are inadequate and need to be supplemented.

Thus, the white box testing acts as both a check on the quality of the requirements-based tests (and the specifications) and a driver for creating additional test cases to ensure all requirements are covered by test cases.

White Box Software Unit Testing

The clause of ISO 26262-6 that defines the requirements for white box software unit testing is shown below. We should typically achieve full coverage unless there is a clear rationale for the inclusion of code that cannot be reached by test cases (e.g. code to handle unexpected error conditions).

> **ISO 26262-6 9.4.5 [Software Unit Level]**
>
> To evaluate the completeness of test cases and to demonstrate that there is no unintended functionality, the coverage of requirements at the software unit level shall be determined and the structural coverage shall be measured in accordance with the metrics listed in Table 12 [see Figure 1]. If the achieved structural coverage is considered insufficient, either additional test cases shall be specified, or a rationale shall be provided.

 ©Stuart Reid, 2017

ISO 26262 requires increasing levels of white box coverage criteria to be achieved as the level of risk (represented by the ASIL level) increases. These are specified in Table 12 of ISO 26262-6 (see Figure 1).

ISO 26262-6 Table 12: **Software Unit Level White Box Coverage Requirements**	**ASIL (Automotive Safety Integrity Levels)**			
	A	**B**	**C**	**D**
Statement	HR	HR	R	R
Branch	R	HR	HR	HR
MC/DC (Modified Condition/Decision Coverage)	R	R	R	HR
R – Recommended HR – Highly Recommended where, "the methods with the higher recommendation should be preferred".				

Figure 1: ISO 26262 Unit Level Test Coverage Requirements

As can be seen, the requirements are somewhat confusing, and need to be carefully interpreted. If you achieve full MC/DC, then you will have automatically achieved full branch coverage (MC/DC subsumes branch coverage). Similarly, full branch coverage subsumes statement coverage. Thus, there is no point in ever using more than one level of coverage on the same code (even though the table highly recommends two coverage criteria for software at ASILs B and D. Thus, the simplest interpretation of Table 12 is shown in Figure 2.

Practical interpretation of ISO 26262-6 Table 12	**ASIL**			
	A	**B**	**C**	**D**
Statement	✓			
Branch		✓	✓	
MC/DC (Modified Condition/Decision Coverage)				✓

Figure 2: Recommended Unit Level Test Coverage Requirements

We shall next describe each of the three required levels of coverage and the corresponding testing techniques that are used to derive test cases to achieve full coverage.

©Stuart Reid, 2017

Statement Coverage and Testing

Statement coverage and testing is based on exercising each executable statement in the code.

> **Executable Statements**
>
> Executable statements are the lines of source code that implement the logic in the program. Other code that is not executable includes comments and white space.
>
> When we measure statements in a program we consider the smallest atomic unit of computation to be a statement (it is either executed, or not). For instance, '`x = 7;`' is a single executable statement as it cannot be broken down any further and we cannot execute part of it without the rest. The code '`if x==7 then y = 4;`' is two executable statements, as if we execute this code with x not equal to 7 then the second statement ('`y=4`') will not be executed.

Procedure

1. Identify each of the executable statements in the source code – these are the test coverage items. With practice, this can be done directly from looking at the source code, but initially it may be easier if a model of the control flow paths through the source code is used (see the next section on branch testing for more on this topic).
2. Identify control flow subpaths through the code that reach executable statements for which there is no test case.
3. Determine the test input values that would cause each of the identified control flow subpaths through the code to be followed.
4. Complete the test cases by applying the set of test input values to the test item's specification to generate the expected results.

Coverage

Statement coverage is calculated as the percentage of executable statements that are exercised by the test cases.

Limitation of Statement Coverage and Testing

A specific limitation of statement testing can be seen by comparing it with branch testing, which requires decisions to be tested both true and false, whereas statement testing does not. For instance, if we consider the code:

```
if sensor_fail = false then
    print "sensor status OK";
endif
```

Just one test, with **sensor_fail** set to false, will exercise all the statements, and "sensor status OK" will be printed. We can imagine that this is as required by the specification, and so the test will pass, and we may be fooled into thinking that everything is fine. However, although we have achieved full statement coverage, we have not covered the situation when the sensor has failed (**sensor_fail =**

 ©Stuart Reid, 2017

true). What happens if we run this extra test? With **sensor_fail** set to true, then the above code outputs nothing. However, the code above has a defect in it – according to our imaginary specification, the code was supposed to activate a back-up sensor if the primary sensor failed, and so the code should have looked like this:

```
if sensor_fail = false then
    print "sensor status OK";
else
    activate (backup_sensor);
endif
```

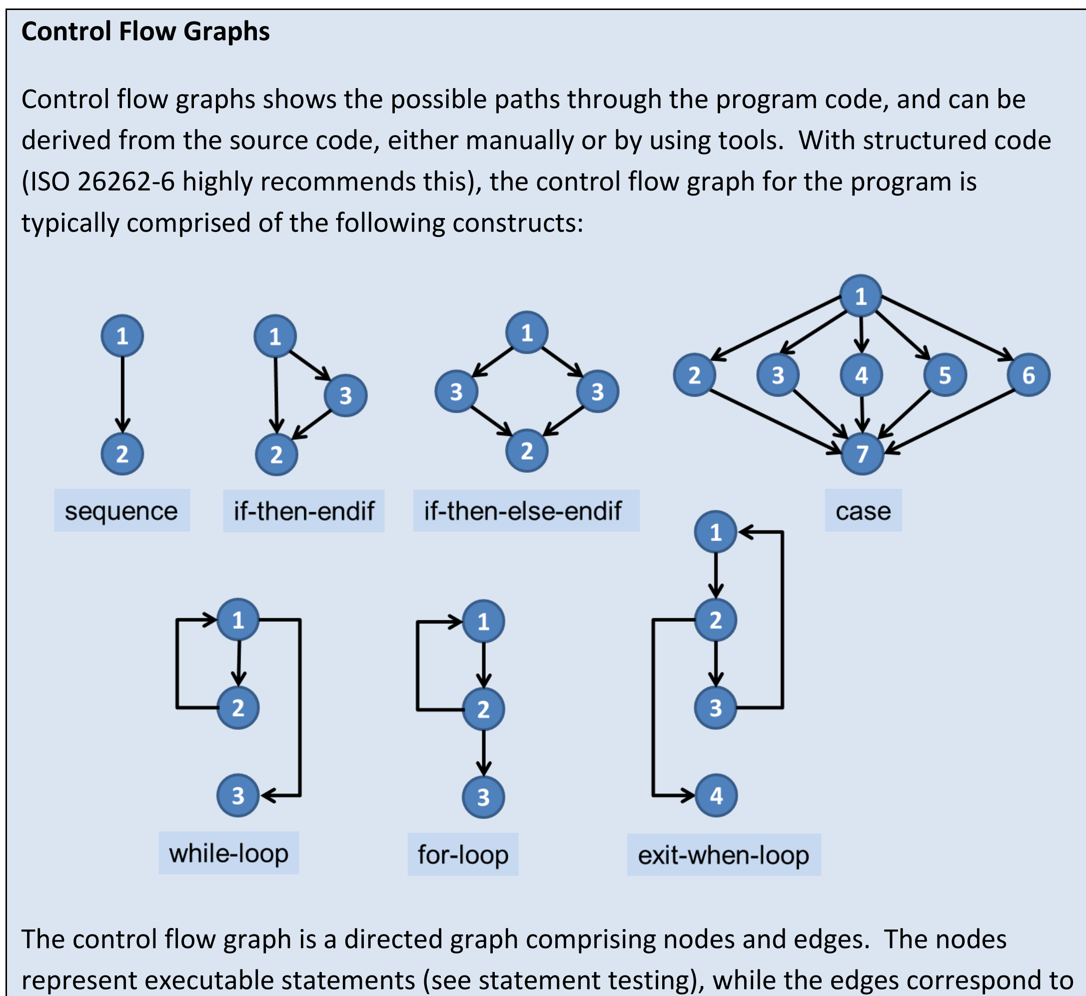
It is only when we perform testing to achieve full branch coverage (see next section), which requires both routes through the code to be taken (even if there are no statements on one of the routes), that we detect that the programmer missed out the second part of the if-statement, which looks as if it may be quite important!

Branch Coverage and Testing

Branch testing is focused on exercising each branch in the control flow of the program.

Control Flow Graphs

Control flow graphs shows the possible paths through the program code, and can be derived from the source code, either manually or by using tools. With structured code (ISO 26262-6 highly recommends this), the control flow graph for the program is typically comprised of the following constructs:



The control flow graph is a directed graph comprising nodes and edges. The nodes represent executable statements (see statement testing), while the edges correspond to

©Stuart Reid, 2017

transfers of control and are otherwise known as control flow branches, which are the basis of branch testing.

Procedure

1. Derive a control flow graph for the program. This can be done manually, or using a tool.
2. Identify each of the branches (edges) in the control flow graph – these are the test coverage items.
3. Identify control flow subpaths through the control flow graph that reach branches for which there is no test case.
4. Use the source code and corresponding control flow graph to determine the test input values that would cause each of the identified control flow subpaths to be followed.
5. Complete the test cases by applying the set of test input values to the test item's specification to generate the expected results.

Coverage

Branch coverage is calculated as the percentage of branches that are exercised by the test cases.

Decision Coverage

Decision coverage is very similar to branch coverage. While branch coverage is based on the branches through the control flow graph, decision coverage is based on the outcomes of any decision points in the program (the branches start at these decision points). Typical decisions points occur at '`if`' statements, '`while`' statements, and in '`case`' or '`switch`' statements. For most programs with a single entry point (ISO 26262-6 highly recommends this), 100% branch coverage and 100% decision coverage are equivalent. However, there is the special case of a program with no decisions (which occurs surprisingly frequently in real systems) – for this program, there is a single branch from the begin to the end statements of the programs - and so branch coverage requires a single test case, while because there are no decisions, decision coverage is not relevant for such a program.

Limitations of Branch Coverage and Testing

A specific limitation of branch testing becomes apparent when code with multiple conditions in decisions is considered. For instance, if we consider the code:

```
if (speed>3) AND (brake==true) then
    sound alarm;
endif
```

To satisfy full branch coverage, just two test cases are required, one when both conditions are true (the speed is greater than 3 and the brake is on) and another when at least one of the conditions is not true. The second case can be satisfied in three ways (see truth table in Figure 3). This means that to satisfy branch coverage, two of these possible combinations are not tested (and it gets worse if you consider decisions with more than two conditions).

 ©Stuart Reid, 2017

It is only when we perform testing to achieve higher levels of coverage, such as MC/DC (Modified Condition/Decision Coverage), that combinations of conditions are considered in the testing.

Modified Condition/Decision Coverage (MC/DC) and Testing

MC/DC testing is focused on checking that each condition in a decision can independently affect the outcome of the decision. It is an extension of branch (and decision) coverage applicable for higher risk situations.

Testing Options with Multiple Conditions

If we have multiple conditions in decisions, we have several options for testing these conditions and the possible combinations. To consider these options, we will start by looking at what we need to test all the possible combinations of true and false. Imagine, we have a relatively simple decision, with just two conditions, as shown below.

```
if (speed>3) AND (brake==true) then
    sound alarm;
endif
```

We can create a truth table for this, as shown in Figure 3.

TEST ID	Conditions		Outcome
	speed>3	brake==true	(speed>3 AND brake==true)
1	TRUE	TRUE	TRUE
2	TRUE	FALSE	FALSE
3	FALSE	TRUE	FALSE
4	FALSE	FALSE	FALSE

Figure 3: Two Condition Truth Table for AND

Truth tables are very easy to construct as the possible combinations of conditions follow a set pattern – and they also follow a set rule for the number needed (2^n combinations, where n is the number of conditions).

Using this example, we can now compare the different approaches to condition testing that are defined in ISO/IEC/IEEE 29119-4.

Branch Condition (BC) Testing Coverage
All conditions must be tested both true and false and the decision must also be tested both true and false. Tests 1 and 4 satisfy this requirement using the minimum number of tests. Test 1 must always be included as it is the only combination that results in the decision outcome being true, while test 4 ensures that both conditions are tested both true and false.

 ©Stuart Reid, 2017

Modified Condition Decision (MC/DC) Testing Coverage
All conditions must be shown to independently affect the outcome of the decision. First, we know we need to include test 1 as it is the only combination that results in the decision outcome being true. We next need to select a test to accompany it, that shows that the first condition can change the outcome, while the other condition remains constant. Test 3 satisfies this criterion as the first condition changes from true to false, while the second condition remains false, while the outcome changes from true to false (so the first condition can be seen to have independently affected the overall outcome). Finally, we need to select a test that shows the second condition can affect the outcome, again to accompany test 1 (as that is the only test with a true outcome). Test 2 does this by keeping the first condition constant (at true), changing the second condition to false, and so changing the outcome from true to false.

Branch Condition Combination (BCC) Testing Coverage
All combinations of true and false for each condition should be exercised. This is by far the simplest approach as all entries in the truth table should be tested.

The extended truth table in Figure 4 allows us to see the different tests needed to satisfy the different condition testing coverage criteria.

TEST ID	Conditions		Outcome	BC	MC/DC	BCC
	speed>3	brake==true	(speed>3 AND brake==true)			
1	TRUE	TRUE	TRUE	BC1	MCDC1	BCC1
2	TRUE	FALSE	FALSE		MCDC2	BCC2
3	FALSE	TRUE	FALSE		MCDC3	BCC3
4	FALSE	FALSE	FALSE	BC2		BCC4

Figure 4: Truth Table showing Tests for different Coverage Criteria

The only condition coverage criterion included in ISO 26262 is MC/DC, which, as we have seen, does not require all the condition combinations to be tested. In our example with two conditions, we need three tests to achieve full MC/DC coverage and four for BCC. However, by moving to three conditions, the numbers change to 4 and 8 tests, respectively. The formula for calculating the number of tests for MC/DC is n+1, while for BCC it is 2^n, where n is the number of conditions. Hence, the reason MC/DC is required by ISO 26262, and BCC is not. Whether this is a good reason for requiring MC/DC for high risk automotive software is debatable, and should be considered along with the limitations of MC/DC described later in this section.

Procedure

1. Derive a control flow graph for the program. This can be done manually, or using a tool.
2. Identify each of the decisions in the program and create a truth table based on the conditions.
3. Using the truth table, determine the combinations of conditions in the decision that need to be exercised to show that each condition can individually affect the outcome of the decision.

4. Identify control flow subpaths through the control flow graph that exercise an identified combination of conditions that can individually affect the outcome of a decision for which there is no test case.
5. Use the source code and corresponding control flow graph to determine the test input values that would cause each of the identified control flow subpaths to be followed.
6. Complete the test cases by applying the set of test input values to the test item's specification to generate the expected results.

Coverage

MC/DC (Modified Condition/Decision Coverage) is calculated as the percentage of conditions that independently affect the outcome of a decision that are exercised by the test cases.

Limitations of MC/DC (Modified Condition/Decision Coverage)

MC/DC (Modified Condition/Decision Coverage) testing has several limitations:

- It does not find all the defects that can be found by testing all the combinations of conditions. This means that some potential defects will be missed (about 6% of the defects possibly found by branch condition combination (BCC) testing are missed by using MC/DC testing).
- It is only an effective means of reducing the number of test cases compared to branch condition combination (BCC) testing when the number of conditions (n) within a decision becomes large and the difference between 'n+1' and '2^n' test cases becomes significant. For most programs, this is not the case, as programmers typically restrict the number of conditions in decisions to a reasonably low number.
- The complexity of identifying the minimal set of test cases that satisfy MC/DC is far higher than simply selecting every entry in the truth table, which satisfies the requirement for branch condition combination (BCC) coverage.
- When the conditions within a decision are not completely independent (e.g. 'if (A OR B) AND (A OR B) then' – has A twice), this will mean that testing will not be able to demonstrate that all conditions can independently affect the outcome of the decision.
- Programmers who find MC/DC complicated are often tempted to move the complex logic outside of the decisions, so that tools that measure MC/DC coverage do not identify these condition combinations as being missed by the testing.
- When a programming language uses short-circuit evaluation, such as C (used for most automotive software), then MC/DC testing cannot be completed. For example, when we already know that X is true, and we encounter 'if (X OR Y) then', short-circuiting will evaluate the outcome to true without ever evaluating and considering the value of Y, as whatever its value, the overall decision will always evaluate to true when X is true.

Achieving Full Coverage with White Box Testing

When white box coverage criteria are used, as required by ISO 26262-6, only full coverage of the statements, branches or MC/DC are considered acceptable.

White box coverage levels of less than 100% are of limited value. This is because we do not know the distribution of defects in the program code – and so we do not know how many defects are in the code that has not been exercised by tests. Also, when writing tests, we tend to initially create

tests that cover the easiest part of the program, leaving the more difficult and complex code until later. If we target test coverage levels of less than 100%, then those parts that are never exercised by tests tend to be in the more complex and difficult-to-understand parts of the program, which is also where the most severe defects are likely to be.

In practice, we measure the level of white box test coverage achieved by using tools. These tools are readily available, and relatively simple – they add extra code (known as instrumentation) to the test item and this extra code records when each of the original parts of the code are exercised by a test. One difficulty of adding extra code is that it may cause the program to behave differently from a functional perspective and so corrupt the test results. This can be checked by running the final set of test cases through both the original and instrumented code and checking that the outputs are the same.

Applicability of White Box Testing

White box testing should be used after black-box testing to supplement the black-box test cases to ensure the necessary level of white box coverage is achieved. It should provide complementary results to the black-box tests, which cannot detect defects associated with code that is not mentioned in the specification. Therefore, white box testing becomes more useful when the specifications are high-level and include few implementation details.

According to ISO 26262-6, and as shown in Figure 5, the white box testing techniques of statement testing, branch testing and MC/DC testing are all applicable for software unit testing – and this means that these techniques are normally applied by the programmers.

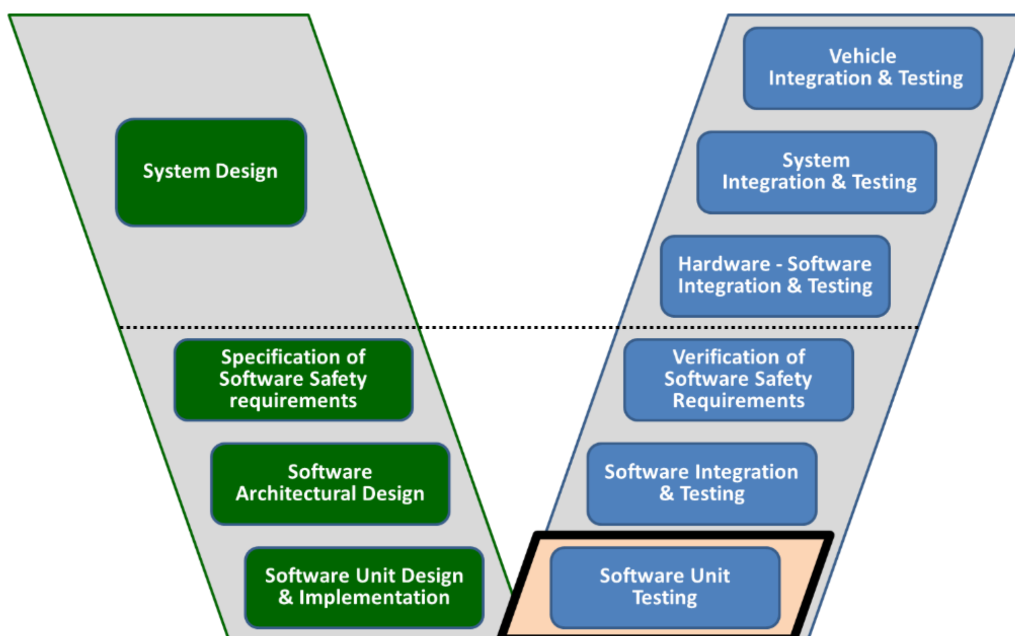


Figure 5: Statement Testing - Lifecycle Phase

If model-based development is used, the practices described here can also be applied to the models.

 ©Stuart Reid, 2017

Limitations of White Box Testing

White box testing has several limitations that are true of statement, branch and MC/DC testing techniques:

- It requires access to the program source code, which may present difficulties depending on the ownership of the source code.
- Tests are based on the source code, which means any missing functionality that is not in the source code will not be tested (therefore we must use black box and white box test techniques together).
- Exercising code to achieve 100% coverage does not guarantee correctness. A piece of code that works with one input is not guaranteed to work with all other inputs. For instance, the code may include 'y = 10/x;'. This will work with most inputs, but if we pass it a value of x = 0, then it will probably cause the program to crash.
- Some code, such as error handling code, can be difficult to reach unless we use special techniques, such as fault injection testing.
- The extra level of detail required when performing white box testing makes it far more expensive than black-box testing, in general.
- It requires the tester to have sufficient knowledge of the programming language to identify the code structures they want to exercise. However, we would normally expect programmers to perform white box testing.
- Tools are needed to measure the level of coverage achieved, and these tools tend to be language-specific.

©Stuart Reid, 2017