SQA CONSULTANT
Inventing the future....

Source Code Analysis for Highly Safety Critical Applications

White paper - August 2014

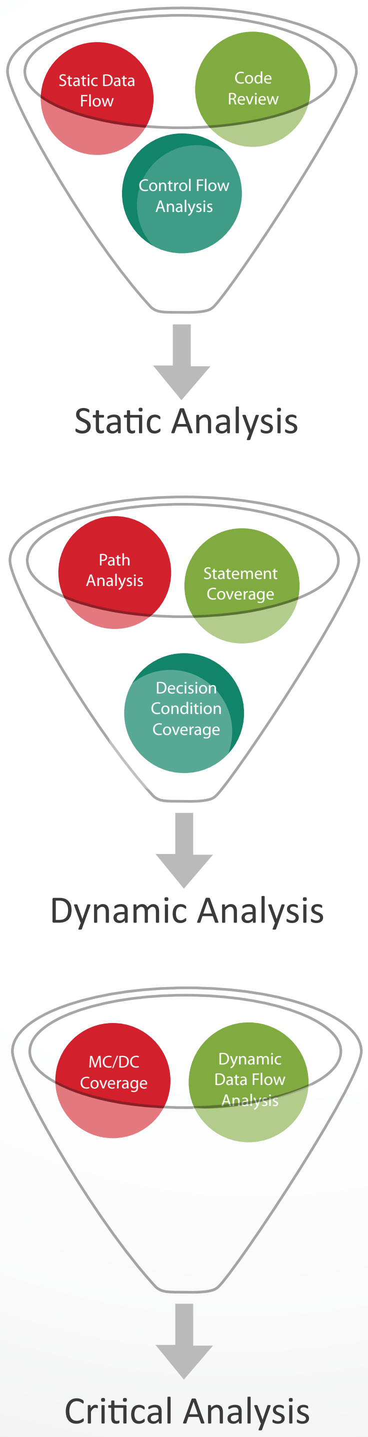
Table of Contents

Source Code Measurement Techniques

About this Paper

This document has been written to provide the answer to two questions:

- How can we increase reliability in our system by source code measurement techniques?
- What are the combinations of coding measurements to achieve safety certification of highly critical applications?



Reason for Source Code Analysis

While working in the safety critical industry we experience several software bugs which can be the cause of human deaths, environmental loss or financial loss for an organization. This paper will highlight one such problem, which caused life threat and financial loss situation for a well-known organization. But the main goal of this paper is to analyze how we can eliminate such hazardous situations from our software.

The following article is referenced from the EDN magazine and is available at http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences
On Thursday October 24, 2013, an Oklahoma court ruled against Toyota in a case of unintended acceleration that led to the death of one of the occupants. Central to the trial was the Engine Control Module's (ECM) firmware.

Embedded software used to be low-level code we'd bang together using C or assembler. These days, even a relatively straightforward, albeit critical, task like throttle control is likely to use a sophisticated RTOS and tens of thousands of lines of code.

For this research, EDN consulted Michael Barr, CTO and co-founder of Barr Group, an embedded systems consulting firm. As a primary expert witness for the plaintiffs, the in-depth analysis conducted by Barr and his colleagues illuminates a shameful example of software design and development, and provides a cautionary tale to all involved in safety-critical development, whether that is for automotive, medical, aerospace, or anywhere else where failure is not tolerable.
Barr's ultimate conclusions were that:

- Toyota's electronic throttle control system (ETCS) source code is of unreasonable quality.
- Toyota's source code is defective and contains bugs, including bugs that can cause unintended acceleration (UA).
- Code-quality metrics predict presence of additional bugs.
- Toyota's fail safes are defective and inadequate (referring to them as a "house of cards" safety architecture).
- Misbehaviors of Toyota's ETCS are a cause of UA

Static Code Analysis

It is source code analysis without executing the program. Following measurement techniques are discussed within this paper.

1. Code Review
2. Control Flow Analysis
3. Static Data Flow Analysis

Code Reviews

Purpose of Code Reviews

Mostly code reviews are used to achieve the following goals:

- To achieve level of reliability by reducing errors in the software. It could save testing time during the development process.
- To save maintenance cost in future. Different types of coding standards are followed in order to identify which part of code is undetermined.

These are the basic intentions of all code reviews and can be further extended such as:

- In some organizations regular code reviews are part of the process, senior or lead developers use their experiences to identify gaps and risks in the source code. Sometimes they use software tools for peer reviews but this approach has a few side effects e.g. mistakes get overlooked, clash of egos and huge time consumption.
- Code review with the team members is like a technical discussion and brainstorming sessions. The new ideas enable the team to take steps forward in terms of the techniques used.
- Sometimes code reviews are used to achieve some customer related standards or certifications. If someone is using third party code then it is necessary to make review process for acquired code.

Common Approaches to Code Reviews

There are many approaches to code reviews and a few of them are highlighted in the coming sections.

1. Peer Reviews
2. Code Analysis using Tools

Peer Reviews

Peer review is the evaluation of work by one or more people of similar competence to the producers of the work. It constitutes a form of self-regulation by qualified members of a profession within the relevant field.

Second version of this approach is code walkthrough from other team members. In this approach a person should perform a line-by-line review and identify inconsistencies, harmful areas and lack of clarity in the source code. This approach has a number of challenges such as:

- Allocating sufficient time
- Overcoming egos
- Planning to execute this process

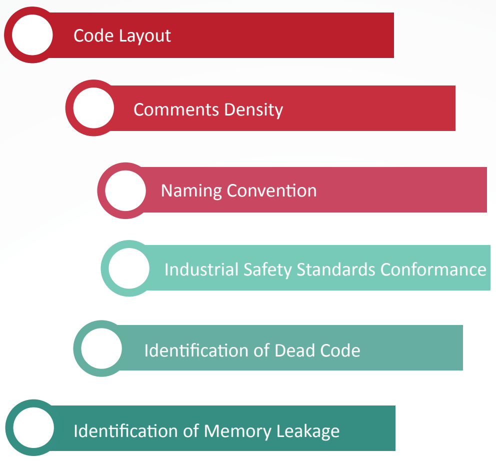
Practically this process involves the developer of the code with one or more reviewers; he presents his approach to other colleagues. Reviewers analyze the author's approach, add comments regarding logics and identify erroneous situations. One common document is prepared to add findings of the complete team in one source code file. These points are taken as action items for the next code review meeting.

If the above process is being executed on a third party source code then improvement or modification in the code is responsibility of the relevant party.

Code Analysis using Tools

It is the second approach in which some tools are used for Code reviews. Most of the organizations use this approach to save time and avoid failures by using some pre-defined industrial standards.

Software tools perform syntax, layout and structural analysis on the source code and report deviation from a predetermined set of coding standards.


Code Layout
Comments Density
Naming Convention
Industrial Safety Standards Conformance
Identification of Dead Code
Identification of Memory Leakage

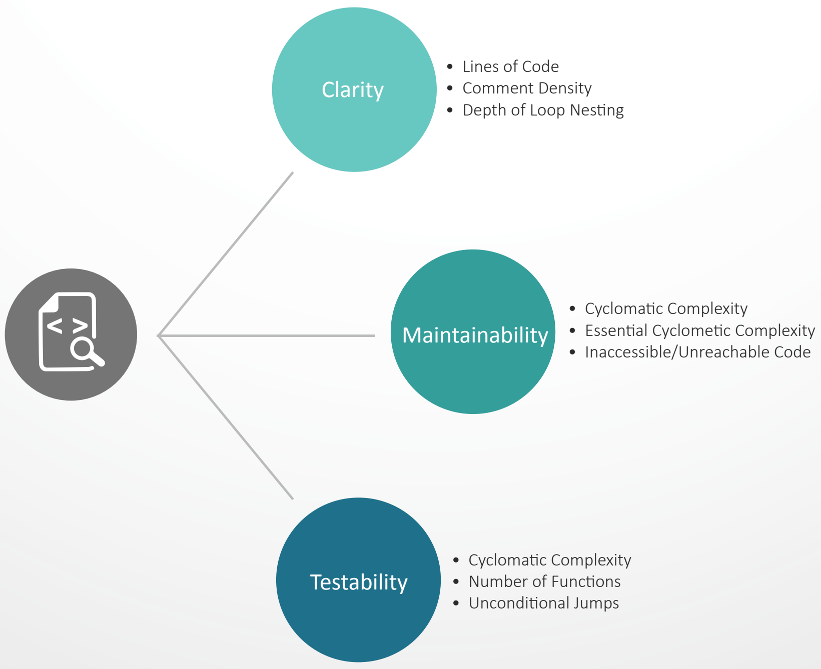
The main problems of this method are threefold:

- Tools report lots of deviations in legacy code
- Different tools are used for each language
- Different programming standards are required for each language

Control Flow Analysis

Purpose of Control Flow Analysis

Control Flow Analysis is the second level of code review by using software tools. This approach is usually used to verify structure of source code and improve its quality. The following measurements are taken during this analysis.



Lines of Code

The lines of code of the project baseline should be measureable and within some defined limit. Limits of these quality checks vary with organizational standards. This attribute takes a part in software clarity check.

Comment Density

Complete source code of the application should be properly commented. It is necessary that third person or external assessor can understand the flow of the application by using comments. Software tools should be used to measure the density of code comments. Density of code comments vary with organizational standards. This attribute also takes a part in software clarity check.

Loop Nesting

Higher number of loop nesting should be avoided; static analyzer tools are used to measure this violation. Due to higher number of loop nesting proper code review is not possible and additionally system performance also degrades. This attribute also takes a part in software clarity check.

Cyclomatic Complexity

Cyclomatic complexity of each individual function should be checked during the control flow analysis. Usually McCabe algorithm is used for the measurement of Cyclomatic Complexity of a function. A high number for the Cyclomatic Complexity value means code is difficult to test and maintain. On the other hand system performance is degraded with higher complexity.

It is a software matrix which is used to measure the complexity of the software program. Cyclomatic Complexity matrix is mainly based on the number of decisions/linearly independent paths in a program's source code.

$$V(G) = \text{No. of edges} - \text{No. of nodes} + 2$$

Essential Cyclomatic Complexity

Essential Cyclomatic Complexity of each individual function should also be checked during the control flow analysis. It is a rare algorithm and is not followed in common software analyzer tools. This algorithm checks whether the target function is properly structured or not. Normally this algorithm is used in Structure Programming Verification (SPV). Limit of this algorithm is also selectable and depends on organizational standards.

Inaccessible/Unreachable Code

Unreachable code is a program code fragment which is never executed. It only adds to the size of the program but neither causes any performance losses nor contributes to any computation. However, its presence may indicate some logical errors. Unreachable fragment of code could be removed without any modification in program.

Below is an example of unreachable code:

Example:

```
intfunc (int x)
{
int y = x*2;
   return y;
// Inaccessible code
if (y < 10)
{
    y += 10;
}
return y;
}
```

Number of Functions

It is recommended that number of functions per module should be defined in organizational standards. There shall be some limit on number of functions when modularity approach is followed in the project. This measure can balance the module density but it depends on the criticality of the application. It is not a compulsory check for safety critical applications but it can reduce testing time in the dynamic analysis.

Unconditional Jumps

Unconditional Jumps are strictly prohibited in safety critical applications. Unconditional Jumps always create uncertainty in the module structure. Let's see an example:

Unstructured Code	**Structured Code**
1. x = 0	1. FOR x= 1 TO 10
2. x= x + 1	2. PRINT x; " squared = "; x * x
3. PRINT x; " squared = "; x * x	3. NEXT x
4. IF x >= 10 THEN GOTO 6	4. PRINT "Program Completed."
5. GOTO 2	5. END
6. PRINT "Program Completed."	

Structure Programming Verification

A checklist could be created to ensure that complete code is properly structured. For example, in the following table we have taken three attributes from above sections and compared each function to decide whether the functions are properly structured or not.

Checklist

Function Name	Cyclomatic Complexity	Essential Complexity	Unconditional Jumps	Structure Programming Verification
Func_1	PASS	PASS	NO	YES
Func_2	PASS	PASS	NO	YES
Func_3	PASS	FAIL	NO	NO
Func_4	FAIL	PASS	NO	NO
Func_5	PASS	PASS	YES	NO

Static Data Flow

Static data flow analysis plays an important role in performance improvement of source code modules. It is a technique for gathering information about the possible set of values that data can take during the execution of the system.

Purpose of Static Data Flow Analysis

If we design a program to create, set, read, evaluate and destroy data; then we must consider the errors that could occur during those processes.

Some examples of data flow errors are mentioned below:

- Assigning an incorrect or invalid value to a variable. These kinds of errors include data-type conversion issues where the compiler allows a conversion but there are side effects that are undesirable.
- Incorrect input results in the assignment of invalid values.
- Failure to define a variable before using its value elsewhere.
- Incorrect path taken due to the incorrect or unexpected value used in a control predicate.
- Trying to use a variable after it is destroyed or out of scope.
- Redefining a variable before it is used.

Set-Use pairs are a notation of data flow. In Set-Use pair, we split the life cycle of a data variable into three patterns.

[d:] Defined

When the variable is defined, initialized or created

[u:] Used

When the variable is used in computation

[k:] Killed

When the variable is killed or destroyed

No.	Notation	Anomaly	Explanation
1.	~d	first define	Allowed.
2.	du	define-use	Allowed, normal case.
3.	dk	define-kill	Bug, data were never used.
4.	~u	first use	Potential bug, data were used without definition. It may be a global variable, defined outside the routine.
5.	ud	use-define	Allowed, data used and then redefined.
6.	uk	use-kill	Allowed,
7.	~k	first kill	Potential bug, data are killed before definition.
8.	ku	kill-use	Serious defect, data are used after being killed.
9.	kd	kill-define	Usually allowed. Data are killed and then redefined. Some theorists believe this should be disallowed.
10.	dd	define-define	Potential bug, double definition.
11.	uu	use-use	Allowed, normal case. Some do not bother testing this pair since no redefinition occurred.
12.	kk	kill-kill	Likely bug.
13.	d~	define last	Potential bug, dead variable? May be a global variable used in another context.
14.	u~	use last	Allowed. Variable was used in this routine but not killed off.
15.	k~	kill last	Allowed, normal case.

"~" notation is used to identify whether a variable is defined first or last e.g. ~x means the variable is defined first and then used and x~ means the variable is defined after use.

Data Flow Anomaly Example

```
1.     public static double paymentcal (int point)
2.     {
3.     doublepayment = 0;
4.     if (point > 0)
5.      {
6.      payment= 40;
7.      if (point>100)
8.       {
9.       if(point <=200)
10.       {
11.       payment = payment + (point - 100) * 0.5;
12.      }
13.      else
14.       {
15.       payment = payment+ 50 +(point-200) *0.1;
16.       if(payment >=100)
17.        {
18.        payment = payment * 0.9;
19.       }
20.      }
21.     }
22.    }
23.    returnpayment;
24.    }
```

Assuming some points were used, however, the payment is set to $40 in line 6. In line 7 we see if more than 100 points were used; in line 9 we check if more than 100 but less than 200 points were used. We calculate the extra points over 100 and add $0.50 for each one. If over 200 points, we take the base payment, add $50.00 for the first extra 100 points, and then bill $0.10 per point for all extra points. Finally, we calculate the discount if the payment is over or equal to $100.00.

Case	Anomaly	Type	Explanation
1.	~d (3)	First Define	normal case
2.	dd (3-6)	Define and Define	suspicious
3.	du	Define and Use	normal case
4.	(3-23)(6-11)(6-15)(15-18)	Use and Define	acceptable
5.	ud (11-11)(15-15)(18-18)	Use and Use	normal case
6.	uu (16-18)(16-23)	Use and Kill	normal case
7.	uk (23-24)	Kill Last	normal case
	k~ (24)		

Why Static Data Flow Analysis?

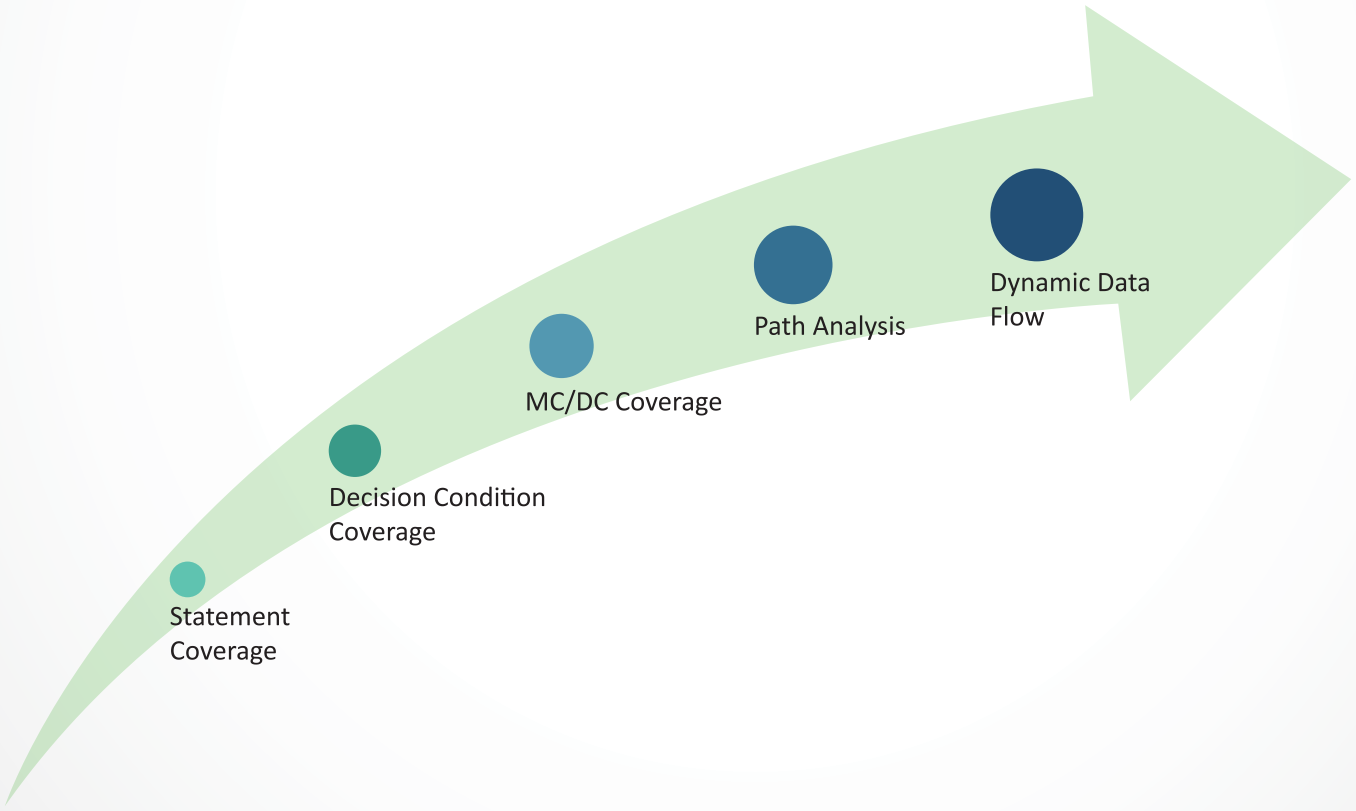
Data flow analysis is a strong technique to make the code reliable by scanning it in a systematic way such that the information about the variables, which are being used in the program, is collected and then conclusions can be made about the side effects of each variable.

Recommendations

Data flow is used to analyze expected and unexpected paths of the software. It is helpful in measuring the impact of one non-critical software module on critical software modules.

Dynamic Code Analysis

Dynamic analysis presents a clear view of the code at the time of execution. By using this analysis, teams become aware of all logical bugs, unreachable state of code, boundary conditions of loops, and run time behavior of variables.



In this section we will discuss following dynamic measurement techniques that are commonly used in safety and security critical applications validation:

- Statement Coverage
- Decision Condition Coverage
- MC/DC Coverage
- Path Analysis
- Dynamic Data Flow

Statement Coverage

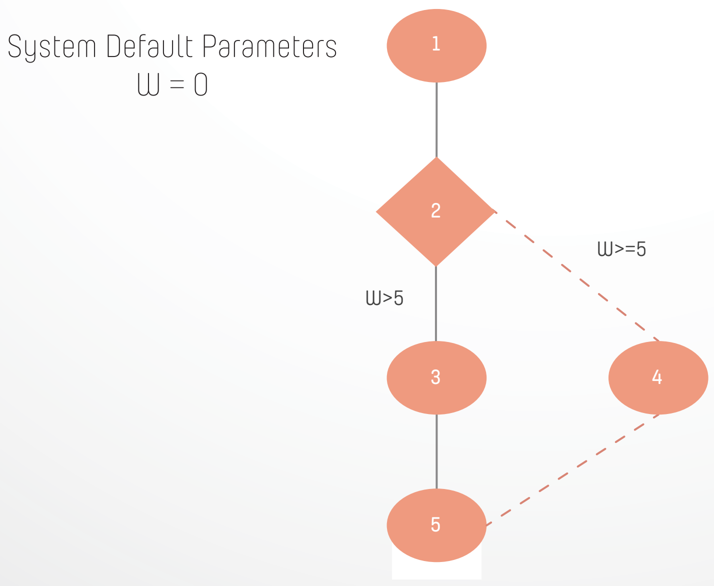
The statement coverage concept is that, every line of code should be exercised at least once during the process of testing. This coverage technique is also called line coverage or basic path coverage. This technique is usually executed by using control flow graphs. Every path which does not have any condition is called basic path and if we are traversing these paths then we are executing statement coverage. Software tools record these paths and prepare a report to show the percent coverage achieved.

Statement Coverage Example

During the basic path execution project relevant team can see the reaction of the system when some harmful statement occurs.

Following is an example of statement coverage:

- System is started with the default value of a variable W = 0.
- System will follow the basic path after startup 1->2->3->5.
- This path will be covered during the statement coverage.
- Second path is 1->2->4->5 and this path is only possible when a value of a variable w>=5.
- Second path will be a part of branch/decision coverage and it is described in the following sections:



Why Statement Coverage?

Statement coverage is the easiest coverage matrix in the dynamic code analysis. It helps the teams to find out bugs that may be inherent in the area which are rarely used. Positive aspect of this coverage is that it is not resource consuming and also builds assessor's confidence on the source code. This technique explores the paths as much as possible within its boundaries because the source code consists of a lot of conditions, loops and jumps.

Recommendations

Many standards and tools recommend that project teams should analyze their code at least to this level. It could eliminate infeasible area of code from your application and be beneficial during the code optimization.

Decision Condition Coverage

Decision condition coverage is extensively used throughout the software industry. It is a compulsory level for highly critical applications in order to make the code assessor confident of the code reliability.

This technique is hybrid form of decision and condition coverage hence the name "Decision Condition" coverage. Here we need to fulfill both decision and condition coverage.

Decision Condition Example

Source Code: if (A AND B)
To cover this decision through decision condition coverage software tool presents the following test cases:

A	**B**	**A AND B (Outcome)**
True False Fulfill Atomic	True False Fulfill Atomic Condition	True False Fulfill Decision Condition

Note that we were able to achieve decision condition coverage without adding any additional test cases. Normally we can achieve decision condition coverage without adding extra test cases; we just need to carefully design the test inputs to fulfill the two simultaneously.

Decision Coverage

In this technique the decision is made to execute both the true and false path once each.

Consider example: if (A AND B) condition occurs in the source code, the decision coverage software tool will present the following test cases:

A	B	A AND B (Outcome)	Possible Decisions
True	True	True	Decision-1
True	False	False	Decision-2

Condition Coverage

This technique focuses more on the atomic conditions available in the decision rather than the decision outcome itself. We need to make all atomic conditions once true and once false irrespective of the decision outcome.

Consider example: if (A AND B) condition occurs in the source code, the condition coverage software tool will present the following test cases:

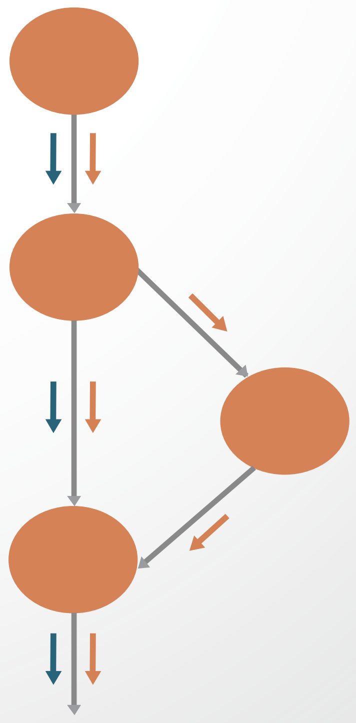
A	B	A AND B (Outcome)
False (Case-1)	True (Case-1)	False
True (Case-2)	False (Case-2)	False

Why Decision Condition Coverage?

Using statement coverage, we were able to execute all lines but not all branches of code. The disadvantage of covering only statement coverage is that we might miss a critical bug which is hidden in some other branches.
Consider the example below:

1. x = 0;
2. if (a > b) then
3. x = 3;
4. else
5. Rep = 63/x;

Test 1: Black arrows
a = 3, b = 2, Rep = 6



Test 2: Grey arrows
a =2, b = 3, Rep =? (CRASH)

In the above example, Test 1 gives 100% statement coverage. However, if we execute Test 2 (to cover decision condition coverage) we will find that execution of line 5 will cause the software to crash as x is 0 and division by 0 is not defined. This is an example of a bug unearthed by decision coverage which would have been overlooked if we relied only on statement coverage.

Recommendations

It is strongly recommended for software engineers to maximize coverage to at least this level. Any unnecessary or infeasible branches should be removed in order to improve the efficiency and compactness of the code.

MC/DC Coverage

MC/DC (Modified Condition/Decision Condition) measurement is considered to be the highest and most powerful technique in the software industry. To ensure the software reliability for airborne systems, Radio Technical Commission for Aeronautics (RTCA) created a guideline and made this technique a compulsory part of safety certification.

This level of coverage is considered stronger because we add another factor to what we were already testing in decision condition coverage. Our bug hypothesis states that we might find a bug hiding in that last little space that we have not tested. MC/DC requires that each atomic condition be tested both ways and that decision coverage must be satisfied. It then adds one more factor as shown in the chart.

Let's put the theory to test with a project example: in this example we will perform both decision condition and MC/DC analysis.

At least one test where the decision outcome would change if the atomic condition X were **TRUE**

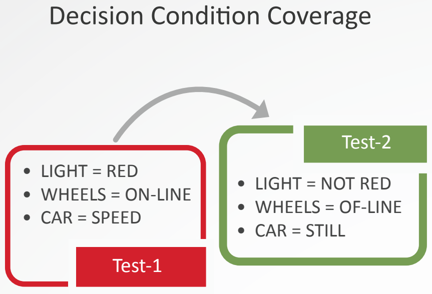
At least one test where the decision outcome would change if the atomic condition X were **FALSE**

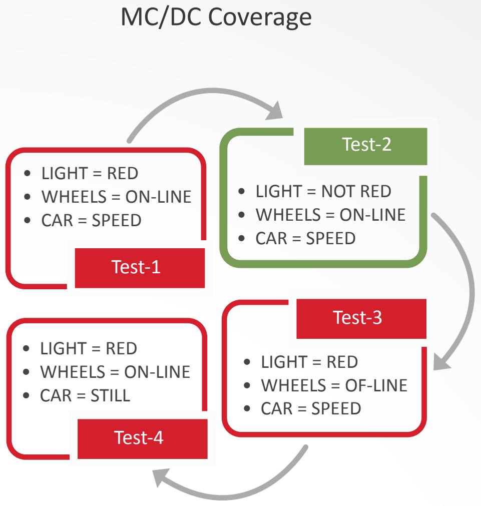
Each different atomic condition has tests that meet requirements 1 and 2

Problem Statement: Consider an automatic traffic violations capture system which activates a camera snapshot whenever a car`s wheels are on/over the line marking the start of intersection, the traffic light is RED and the car is speeding. To drill down the logic implemented, we have the following pseudo code:

Source Code Condition

IF (light=RED &&wheels =On-line &&car =SPEED)





Test Cases	LIGHT = RED	WHEELS = ON-LINE	CAR = SPEED	OUTCOME
Decision Condition Coverage				
Test-1	TRUE	TRUE	TRUE	TRUE
Test-2	FALSE	FALSE	FALSE	FALSE
MC/DC Coverage				
Test-1	TRUE	TRUE	TRUE	TRUE
Test-2	FALSE	TRUE	TRUE	FALSE
Test-3	TRUE	FALSE	TRUE	FALSE
Test-4	TRUE	TRUE	FALSE	FALSE

To achieve maximum coverage we need to execute each combination which results in eight test cases for three inputs (2^n). We can achieve similar outcomes from MC/DC coverage but with lesser number of test cases. If there are n atomic conditions, MC/DC can normally be achieved in n+ 1 test cases.

Interesting Facts

For highest level of coverage required under FAA DO/178C Level A, MC/DC coverage is performed. RCTA and CENELEC standards state that the software that can cause catastrophic impacts on human life should be verified through MC/DC coverage. In other words all the planes that come out of Boeing production have their software verified through MC/DC coverage technique.

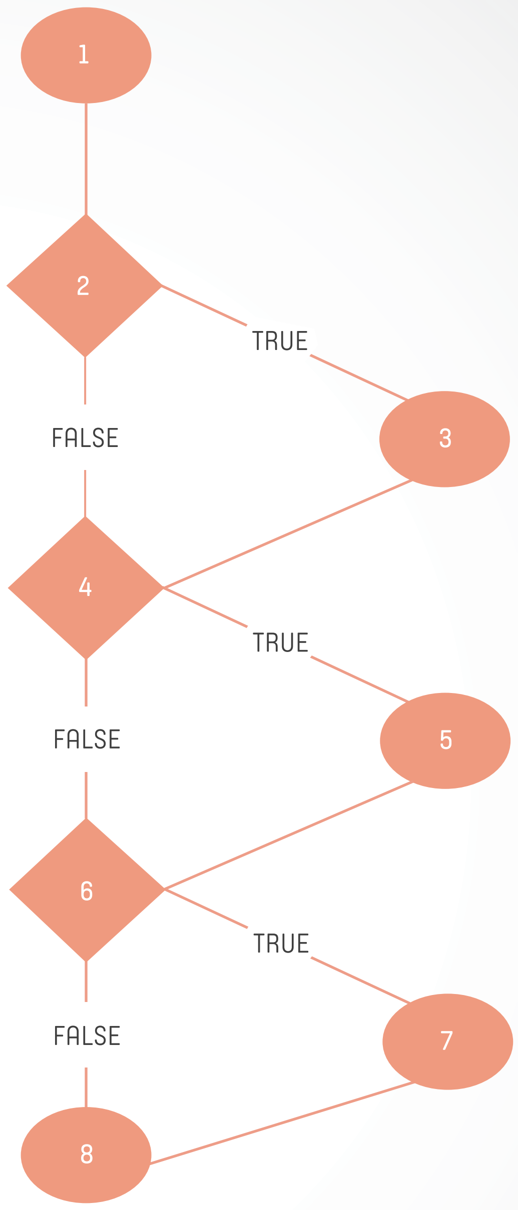
Path Analysis

Introduction

The main goal behind path analysis is that outcomes of every Cyclomatic path should be exercised at least once. Decisions of Cyclomatic path should be recorded in a report and analyzed to know which paths could be harmful for the application. In above stated dynamic coverage methods like statement and decision condition there will still be some possibilities of hidden defects, we can increase degree of coverage by execution of Cyclomatic paths. Although we have achieved a high level of coverage from statement and decision condition but some experts refer that additional paths should also be traversed at least once in the project life cycle even when nature of application is highly security/safety critical.

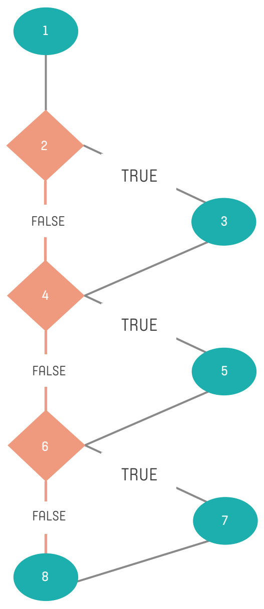
McCabe software is quite famous in path analysis techniques. Following are some examples of path analysis.

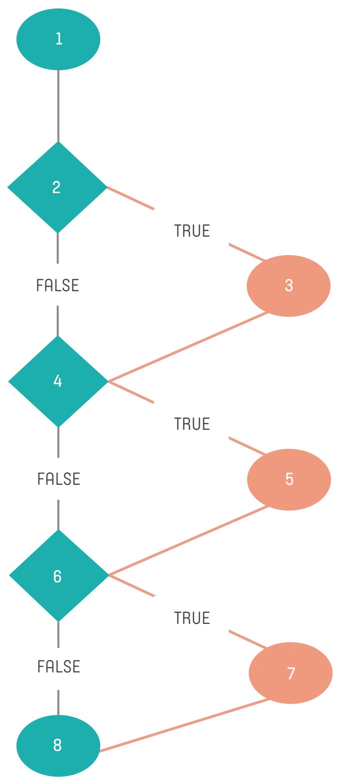
```
void copyStr (char** dest, char** src, int start, int end ){
intToCopy = 1;
intlastpos = strlen(*src)-1;
if (end >lastpos){
end = lastpos;
}
If (start < 0) {
start = 0;
 }
if(end > start){
ToCopy += (end-start);
}
Strncpy (*dest, (*src)+start, ToCopy);
}
```

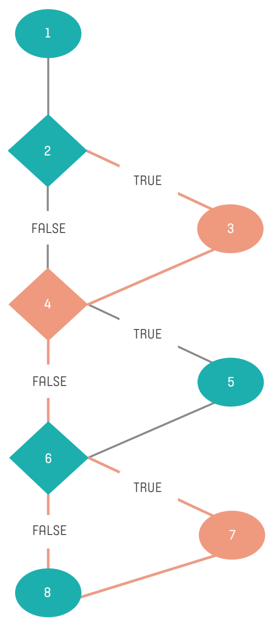
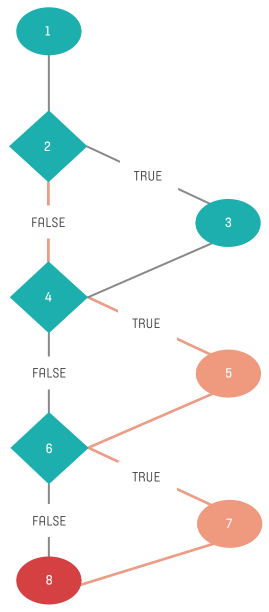


Above function contains four Cyclomatic paths. Two paths will be exercised during decision condition coverage but additional two paths need to be exercised for complete path analysis.

Coverage	Test Cases	Flow Graph

Statement + Decision Condition	char* original = "Hello";
	char* copy = (char*)malloc (80);
	Test-1:
	copyStr (&copy, &original, -500, 500);
	
	Blue area has been executed with this test data
		Test Case-1
		 1
		2
		TRUE
		3
		FALSE
		4
		TRUE
		5
		FALSE
		6
		TRUE
		7
		FALSE
		8

	Statement + Decision Condition char* original = "Hello"; char* copy = (char*)malloc (80); Test-1: copyStr (&copy, &original, -500, 500); Blue area has been executed with this test data Test-2: copyStr (&copy, &original, 0, 0); Blue area has been executed with this test data Statement + Decision Condition is executed 100 %	Test Case-2  1 2 TRUE 3 FALSE 4 TRUE 5 FALSE 6 TRUE 7 FALSE 8

Path Coverage	Additional two test cases are required to complete path analysis Test-3: copyStr (&copy, &original, -10, 0);	Test Case-3 
	Test-4: copyStr (&copy, &original, 1000, 100); Note: This test path exercises the out-of-bounds access of a string. If you analyze the test data, you will conclude that string buffer is smaller than base string size. It is a common mistake that is mostly overlooked in the code review and decision condition analysis. It is a string manipulation defect which is strictly prohibited in the security critical certification standards.	Test Case-4 

Why Path Analysis?

Path analysis is a different measurement technique with respect to code coverage methods. The code coverage like statement and decision condition are more focused on code structuring and their decisions, whereas path analysis measurement developed by McCabe software highlights the Cyclomatic paths which are directly proportional to the complexity level of the functions. McCabe group insists code analysts to exercise Cyclomatic paths at least once in the project life cycle so that target application will be free from security vulnerabilities.

Recommendations

Path analysis technique is especially recommended for security critical applications. This technique also covers decision coverage at higher level, so it can also be used instead of decision condition coverage in different types of projects. Before using this technique in safety critical applications it is necessary to contact your respective standards and certification bodies.

Dynamic Data Flow

Introduction

It is a powerful measurement technique that traces the test paths initiated by control flow test data. Code coverage techniques focus on execution of control flow whereas data flow report focuses on run time utilization of variables during the control flow.
In the static data flow technique, as discussed above, tools are used to analyze the code and estimate the paths in which code will be executed. On the other hand, in dynamic data flow technique tools trace these paths by using local and global variable utilization. The most beneficial part of this technique is that it shows the impact of software functions and variables to other functions and variables while data is executing. Safety agencies like RCTA are quite concerned about the data usage and impact of data on other safety or non-safety modules in the application.

Why Dynamic Data Flow?

Dynamic data flow analysis identifies and narrows down the scope of software functions. It gives clear idea to code analysts about the interface between software modules and side effects of each impact.

Recommendations

This technique is recommended for airborne and all other systems which require certification from RCTA. The advantage of this technique is that tester does not have to write any new test data for the measurement so test case writing time is eliminated as a result. It is a supportive technique in highly critical applications.

Coding Measurements for Safety Critical Application

This section will cover the coding measurement techniques mapping with the safety critical applications. All information in this section is inherited from safety standards either IEC 61508 branch standards or DO-178C standard.

CENELEC IEC 61508 safety standards use the following approach.

SIL 1

- Code Review

SIL 2

- Code Review
- Control Flow Analysis

SIL 3

- Code Reivew
- Control Flow Analysis
- Statement Coverage
- Decision Condition Coverage

SIL 4

- Code Reivew
- Control Flow Analysis
- Statement Coverage
- Decision Condition Coverage
- Dynamic Data Flow (Supportive Measurement)

RCTA DO178B standards use the following approach.

System Level A

- Statement Coverage
- Static Data Flow Analysis
- Dynamic Data Flow Analysis
- Decision Condition Coverage
- MC/DC Coverage

System Level B

- Statement Coverage
- Static Data Flow Analysis
- Dynamic Data Flow Analysis
- Decision Condition Coverage

System Level C

- Statement Coverage
- Static Data Flow Analysis
- Dynamic Data Flow Analysis

System Level D

- Formal Code Review

Delivered Quality Control Systems

Team of Functional Safety Experts

Delivered Highly Critical Applications

SQA CONSULTANT

Inventing the future....

Contact Us

Explore ways to use our expertise in growing your business while establishing a valuable partnership with us.

Contact our consultants at:

Phone: +1.412.533.1700 (Ext: 585)
E-mail: info@sqaconsultant.com
Website: www.sqaconsultant.com