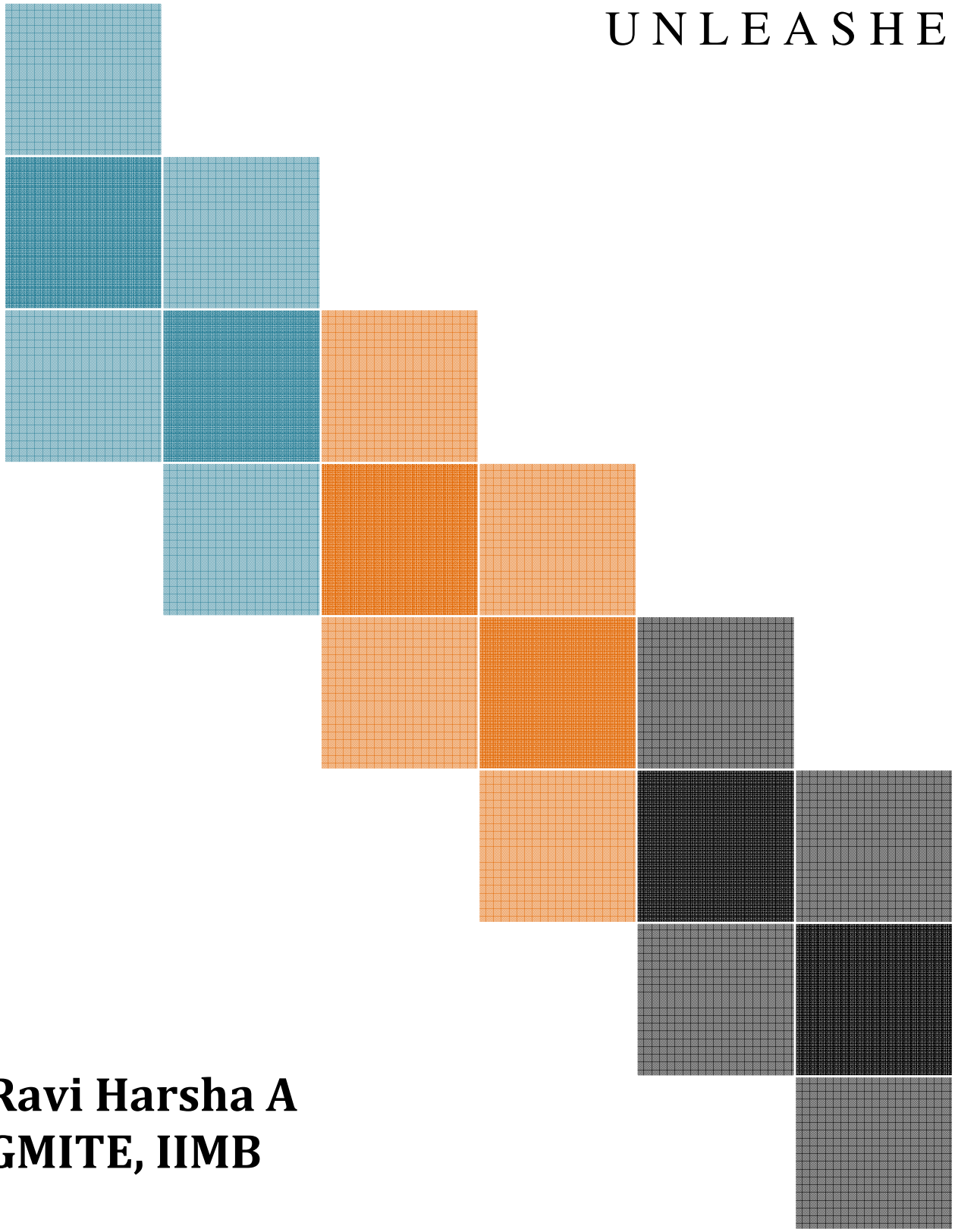
Verification & Validation

UNLEASHED



Ravi Harsha A
GMITE, IIMB

MODIFICATION HISTORY

REVISION	DATE	AUTHOR	REMARKS
V_an_V_Unleashed_i01	01-Jan-2004	Ravi HARSHA A	Draft Creation and Initial Release
V_an_V_Unleashed_i02	01-May-2004	Ravi HARSHA A	Updated with revised Technical Requirements.
V_an_V_Unleashed_i03	18-Aug-2004	Ravi HARSHA A	Updated with review remarks..
V_an_V_Unleashed_i04	01-Dec-2004	Ravi HARSHA A	Updated with revised Technical Information.
V_an_V_Unleashed_i05	20-May-2005	Ravi HARSHA A	Updated the Checklists section.
V_an_V_Unleashed_i06	30-Dec-2009	Ravi HARSHA A	Update of HSIT/SSIT/UT with examples.

NOTA:

Everyone is of the opinion that irrespective of the lifecycle model used for software development, embedded avionics software has to be tested. Efficiency and quality are best served by testing the software as early in the lifecycle as practical, with full regression testing whenever changes are made.

This reference document addresses a basic question often posed by developers who are new to the concept of thorough verification and validation: Why bother to "verify and validate"? Through this document a sincere attempt has been made to answer this question in a methodically phased manner.

Hence any references to the collection of technical contents present in this document with that of any technical documents of any individual or standards or organization or institutes or governing bodies is very much un-intentional and it is purely co-incidental.

- The Author

Ravi Harsha Profile -

- Acquired ITES experience of 10 years in Aerospace & Defense (embedded avionics domain.)
- 03 years Project/Program Management and 05 years Delivery Management expertise.
- Managed long term (> 03 yrs), large teams (60 Engineers) and high revenue projects (> $3 million).
- Driving value creation which resulted in cost saving of 6500 hours to customer over 3.5 years.
- Execution experience of diversified roles of Onsite Coordinator, Engagement Manager.
- Wide experience in working across cultures (France'01, Germany'05, Canada'06 and USA'08).
- Built teams for various embedded avionics software projects at offshore and onsite.
- Managed long term (> 1 year), large Airbus 380 projects (over 20 Engineers).
- Experience of working with top Aerospace & Defense customers' such as Airbus, Boeing, Pratt & Whitney Canada, Hamilton Sundstrand, Nord-Micro, Thales Avionics, Snecma Group.

ABBREVIATIONS

CASE	**C**omputer **A**ided **S**oftware **E**ngineering
CC1	**C**ontrol **C**ategory **1**
CC2	**C**ontrol **C**ategory **2**
CCB	**C**hange **C**ontrol **B**oard
CI	**C**onfiguration **I**tem
CM	**C**onfiguration **M**anagement
CO	**C**hange **O**rder
COTS	**C**ommercial **O**ff **T**he **S**helf
CR	**C**hange **R**equest
CRC	**C**ompliance **R**eview for first **C**ertification
CSCI	**C**omputer **S**oftware **C**onfiguration **I**tem (Executable Object code)
CSI	**C**omputer **S**oftware **I**tem
DER	**D**esignated **E**ngineering **R**epresentative
FDR	**F**irst **D**elivery **R**eview
FFR	**F**irst **F**light **R**eview
HOOD	**H**ierarchical **O**bject **O**riented **D**esign
LLR	**L**ow **L**evel **R**equirement
LLT	**L**ow **L**evel **T**est
PAD	**P**roduct **A**rchitectural **D**escription
PDL	**P**roduct **D**evelopment **L**eader
PDP	**P**roduct **D**evelopment **P**lan
PECI	**P**roduct Life Cycle **E**nvironment **C**onfiguration **I**ndex
PFD	**P**roduct **F**unctional **D**escription
PM	**P**roduct **M**anager
PPR	**P**lanning **P**rocess **R**eview
PR	**P**roblem **R**eport
PRB	**P**roblem **R**eporting **B**oard
PRD	**P**roduct **R**equirements **D**ata
PSAC	**P**lan for **S**oftware **A**spects of **C**ertification
PVVL	**P**roduct **V**alidation and **V**erification **L**eader
SADT	**S**tructured **A**nalysis and **D**esign **T**echnique
SAP	**S**oftware **A**cceptance **P**lan
SAS	**S**oftware **A**ccomplishment **S**ummary
SATP	**S**oftware **A**cceptance **T**est **P**rocedures
SATR	**S**oftware **A**cceptance **T**est **R**esults
SCI	**S**oftware **C**onfiguration **I**ndex

SCM	**S**oftware **C**onfiguration **M**anagement
SCML	**S**oftware **C**onfiguration **M**anagement **L**eader
SCMP	**S**oftware **C**onfiguration **M**anagement **P**lan
SCMR	**S**oftware **C**onfiguration **M**anagement **R**ecords
SDDD	**S**oftware **D**etailed **D**esign **D**escription
SDL	**S**oftware **D**evelopment **L**eader
SDP	**S**oftware **D**evelopment **P**lan
SDR	**S**oftware **D**esign **R**eview
SIP	**S**oftware **I**ntegration **P**lan
SIRD	**S**oftware **I**nterface **R**equirement **D**ata
SITP	**S**oftware **I**ntegration **T**est **P**rocedures
SITR	**S**oftware **I**ntegration **T**est **R**esults
SM	**S**oftware **M**anager
SOW	**S**tatement **O**f **W**ork
SPDD	**S**oftware **P**reliminary **D**esign **D**escription
SPDR	**S**oftware **P**reliminary **D**esign **R**eview
SQA	**S**oftware **Q**uality **A**ssurance
SQAL	**S**oftware **Q**uality **A**ssurance **L**eader
SQAP	**S**oftware **Q**uality **A**ssurance **P**lan
SQAR	**S**oftware **Q**uality **A**ssurance **R**ecords
SRD	**S**oftware **R**equirements **D**ata
SRR	**S**oftware **R**equirements **R**eview
STD	**S**oftware **T**raceability **D**ata
SUTP	**S**oftware **U**nit **T**est **P**rocedures
SUTR	**S**oftware **U**nit **T**est **R**esults
SVVL	**S**oftware **V**alidation and **V**erification **L**eader
SVVP	**S**oftware **V**alidation and **V**erification **P**lan
SVVR	**S**oftware **V**alidation and **V**erification **R**esults
TC	**T**echnical **C**heck
TRR	**T**est **R**eadiness **R**eview

- TABLE OF CONTENTS -

1.0 SCOPE

1.1 Document Identification

The objective of this reference document is to introduce the beginners to the world of verification and validation of the software used in the real time embedded airborne avionics. This will also enable users of this document to understand the importance the quality of the software being currently being developed and/or being verified and finally validated with safety perspective, and thereby make an attempt to improve the quality of their software products.

1.2 Intended audience

The target audience for this technical reference document includes:

- ✓ Software Verification and Validation Engineers (Testers);
- ✓ Software Development Engineers (Designers and Developers);
- ✓ Managers of software designers, developers and testers;
- ✓ Software Technical Controllers.
- ✓ Procurers of software products or products containing software;
- ✓ Software Quality Assurance (SQA) managers and analysts;
- ✓ Academic researchers, lecturers, and students;
- ✓ Developers of related standards.

1.3 Document Overview

The topics covered in this reference document include:

- ✓ Verification Process: A DO-178B perspective.
- ✓ VEE life cycle model.
- ✓ Reviews during VEE life cycle model.
- ✓ Software Verification Strategy.
- ✓ Software Testing Process.
- ✓ Software Integration Testing (HSIT).
- ✓ Unit Testing (UT).
- ✓ Inspection Process
- ✓ Assemble Testing
- ✓ Source to Object Analysis

2.0 INTRODUCTION

The *quality* and *reliability* of software is often seen as the weak link in industry's attempts to develop new products and services. The last decade has seen the issue of software quality and reliability addressed through a growing adoption of design methodologies and supporting CASE tools, to the extent that most software designers have had some training and experience in the use of formalized software design methods.

Unfortunately, the same cannot be said of software testing. Many developments applying such design methodologies are still failing to bring the quality and reliability of software under control. It is not unusual for 50% of software maintenance costs to be attributed to fixing bugs left by the initial software development; bugs which should have been eliminated by thorough and effective software testing.

Validation is a key activity that is essential to checking the correctness of the design and implementation of a software product against some pre-defined criteria. It aims at finding software defects (design and implementation errors) early in the development process to reduce the costs of removing these defects. These costs have been shown to increase with progress in the soft- ware development process.

Verification & Validation activities typically occupy more than half of the overall system engineering efforts. This is particularly true for embedded systems that have to fulfill more stringent non-functional requirements than typical information systems. Verification is an ongoing activity throughout the development process, and validation is traditionally applied before the system becomes operational. Verification and Validation are typical and fundamental quality assurance techniques that are readily applied throughout the entire system development process.

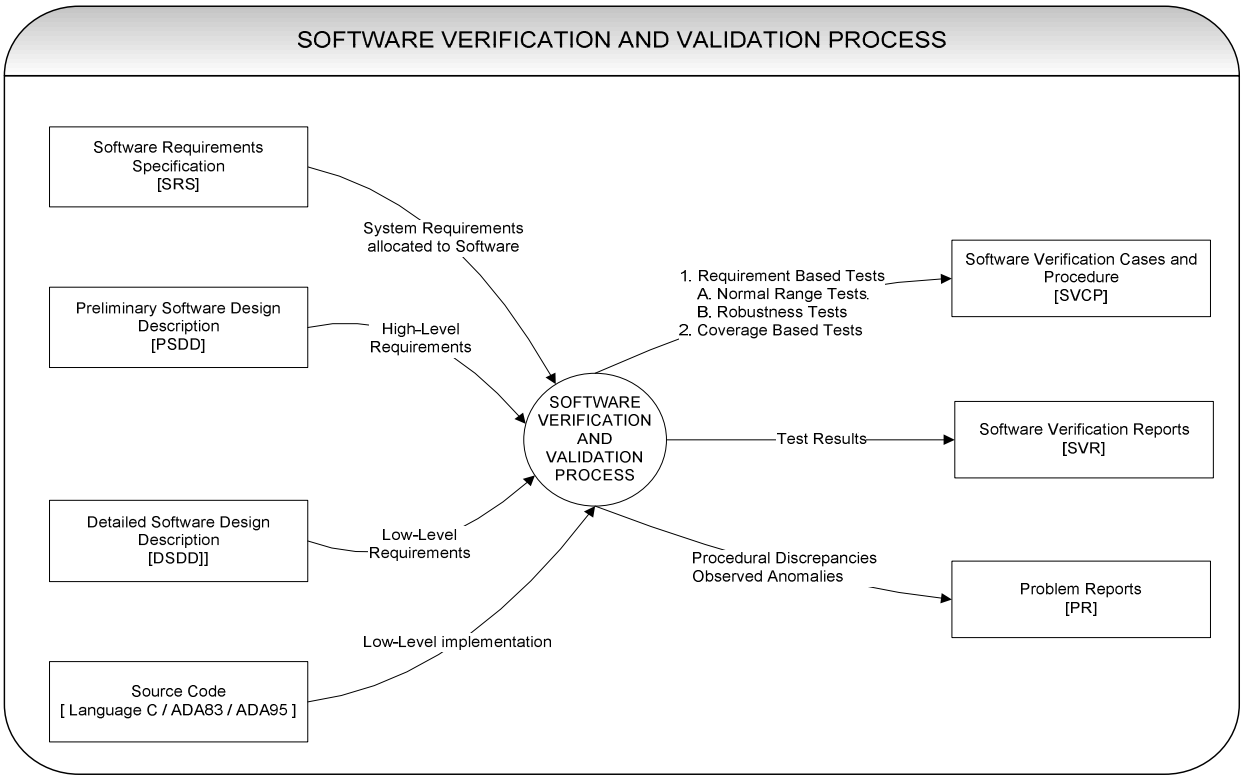
In fact, every single software engineering activity is geared towards two distinct dimensions only*: attaining quality* and *reducing development costs* and the two dimensions are interdependent. A high required level of quality usually leads to high development costs, although we cannot really say that in contrast high development costs inevitably lead to high quality.

In the embedded domain, quality should receive very special attention, because such systems often perform critical tasks whose failure may lead to hazards and harm to assets and human life. Embedded real-time systems are therefore traditionally subject to stringent quality assurance methods, and these are expensive. Especially formal methods and formal verification are traditionally only applied in extremely safety-critical applications such as the avionics or such as nuclear power plants etc.

Since more recently, embedded real-time systems are becoming more and more disposed in most technical domains and this is mainly through the increased integration of services over networks, competitive advantages of software over hardware, and the combination of devices and information systems.

3.0 SOFTWARE VERIFICATION PROCESS: A DO-178B PERSPECTIVE

3.1 INPUT / OUTPUT REPRESENTATION



3.2 PROCESS DOCUMENTS

3.2.1 INPUTS

- Software Requirements Specification [SRS].
- Preliminary Software Design Description [PSDD].
- Detailed Software Design Description [DSDD].
- Source Code.

GUIDELINES & STANDARDS	TEMPLATES	CHECKLISTS
➢ Regulatory Requirements [DO-178B]. ➢ Review and Analysis Guide Lines. ➢ Software Testing Guidelines.	➢ Software Verification Cases & Procedure [SVCP]. ➢ Software Verification Results [SVR]. ➢ Problem Reports [PR].	➢ Peer Review Checklist for • Low-Level Tests. • Software-Software Integration Tests. • Hardware-Software Integration Tests.

3.2.2 OUTPUTS

- Software Verification Cases and Procedure [SVCP] document.
- Software Verification Results [SVR] document.
- Problem Reports [PR] document.

3.3 DOCUMENTS TRACEABILITY WITH DO-178B

DOCUMENTS NAME	PROCESS NAME	ADDRESSED DO-178B SECTIONS
SRS Review Checklist	REVIEW AND ANALYSIS PROCESS APPLIED TO SYSTEM REQUIREMENTS ALLOCATED TO SOFTWARE (HIGH-LEVEL REQR)	6.1.a, 6.1.e, 6.2.a, 6.2.d,6.2.e,6.3.1, 11.13a
PSDD Review Checklist	REVIEW AND ANALYSIS PROCESS APPLIED TO SOFTWARE ARCHITECTURE	6.1.c, 6.1.e, 6.2.d, 6.2.e, 6.3.3, 11.13a
SDDD Review Checklist	REVIEW AND ANALYSIS PROCESS APPLIED TO SOFTWARE LOW-LEVEL REQUIREMENTS	6.1.b, 6.1.e, 6.2.d, 6.2.e, 6.3.2, 11.13a
Source Code Review Checklist	REVIEW AND ANALYSIS PROCESS APPLIED TO SOURCE CODE	6.1.d, 6.1.e, 6.2.d, 6.2.e, 6.3.4, 6.4.4.3.c, 6.4.4.3.d, 11.13a
Build Review Checklist	REVIEW AND ANALYSES PROCESS APPLIED TO OUTPUTS OF THE INTEGRATION PROCESS	6.1.e, 6.2.d, 6.2.e, 6.3.5, 11.13a
Software Verification Cases and Procedures (SVCP),	SOFTWARE TESTING PROCESS	11.13b, , 11.13c
Software Verification Reports (SVR),	SOFTWARE TESTING PROCESS	11.14
Technical Control (TC) / Peer Review Checklist for Low-Level Tests.	SOFTWARE TESTING PROCESS	6.2.d, 6.2.e, 6.4.1, 6.4.2, 6.4.3.a, 6.4.4.1, 6.4.4.2, 6.4.4.3,6.4.4.3.a, 6.4.4.3.b
Technical Control (TC) Checklist for SOFTWARE-SOFTWARE Integration Tests (SSIT)	SOFTWARE TESTING PROCESS	6.2.d, 6.2.e, 6.4.1, 6.4.2, 6.4.3.b, 6.4.4.1, 6.4.4.2, 6.4.4.3.a, 6.4.4.3.b
Technical Control (TC) Checklist for HARDWARE-SOFTWARE Integration Tests (SSIT)	SOFTWARE TESTING PROCESS	6.2.b, 6.2.c, 6.2.d, 6.2.e, 6.4.1, 6.4.2, 6.4.3.c, 6.4.4.1, 6.4.4.2, 6.4.4.3
Problem Reports	APPLICABLE TO ALL PROCESSES OF SDLC.	11.17

4.0 SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

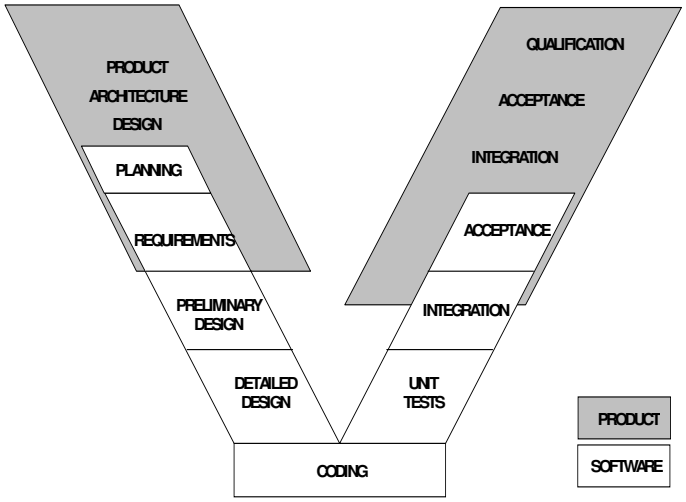
Project life cycles process definitions will differ from project to project depending on system functionality, complexity, criticality level, re-use strategies, hardware availability, prototyping strategies etc. The process definitions, including activities, sequencing between processes and responsibilities are defined in the project software plans.

The various activities which are undertaken when developing software are commonly modeled as a software development lifecycle. The software development lifecycle begins with the identification of a requirement for software and ends with the formal verification of the developed software against that requirement. The software development lifecycle does not exist by itself; it is in fact part of an overall product lifecycle.

There are a number of different models for software development lifecycles. One thing which all models have in common is that at some point in the lifecycle, software has to be verified and also validated. This document outlines the V lifecycle model software development lifecycles, with particular emphasis on the verification and validation activities emphasizing on *sequential lifecycle* model.

4.1 V Life Cycle Model

The following set of lifecycle phases fits in with the practices of most avionics software development.

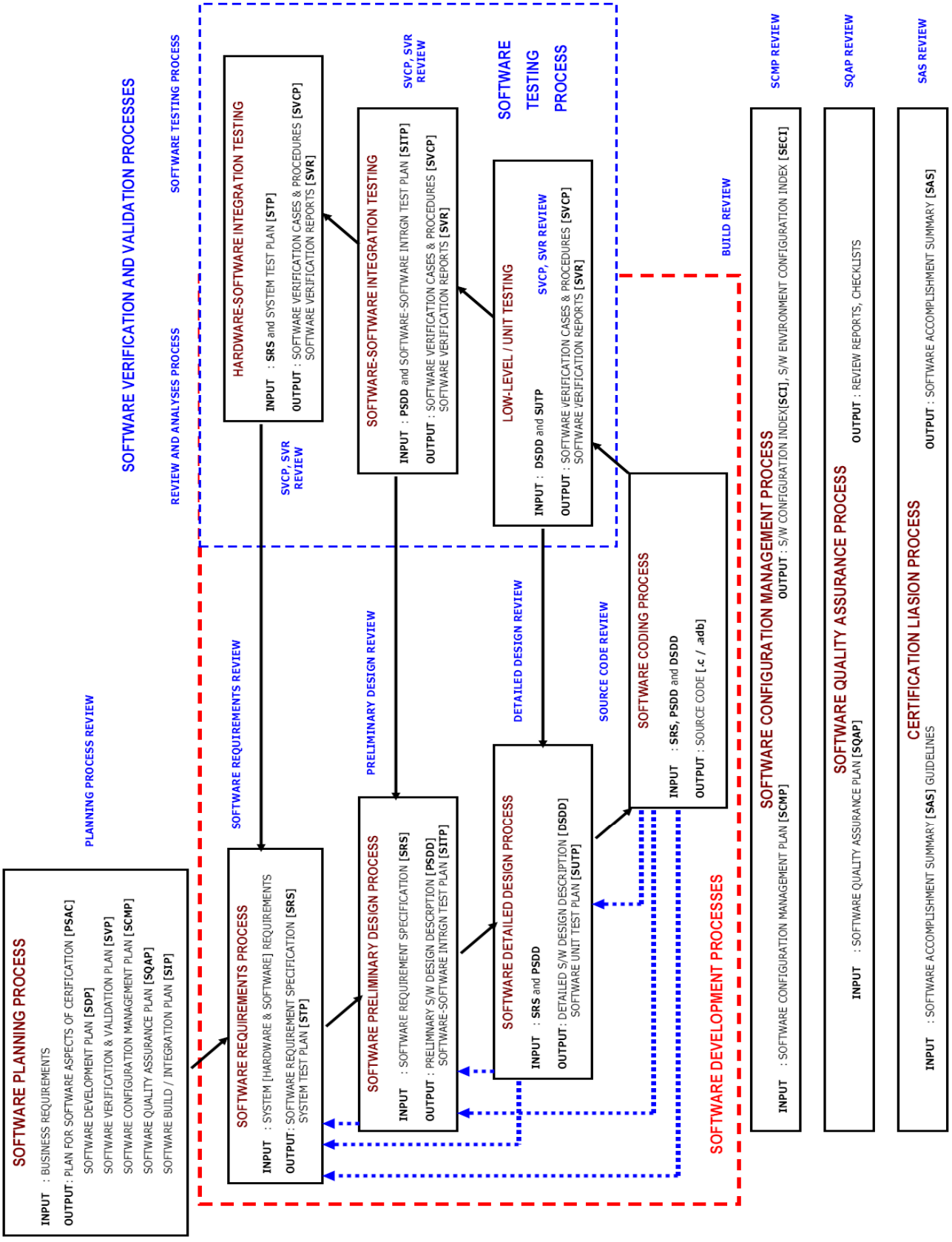


Code and Unit Test phase, in which each component of the software is coded and tested to verify that it faithfully implements the detailed design.

Integration Testing Phase, in which progressively larger groups of tested software components are integrated and tested until the software works as a whole.

System Testing Phase, in which the software is integrated to the overall product and tested.

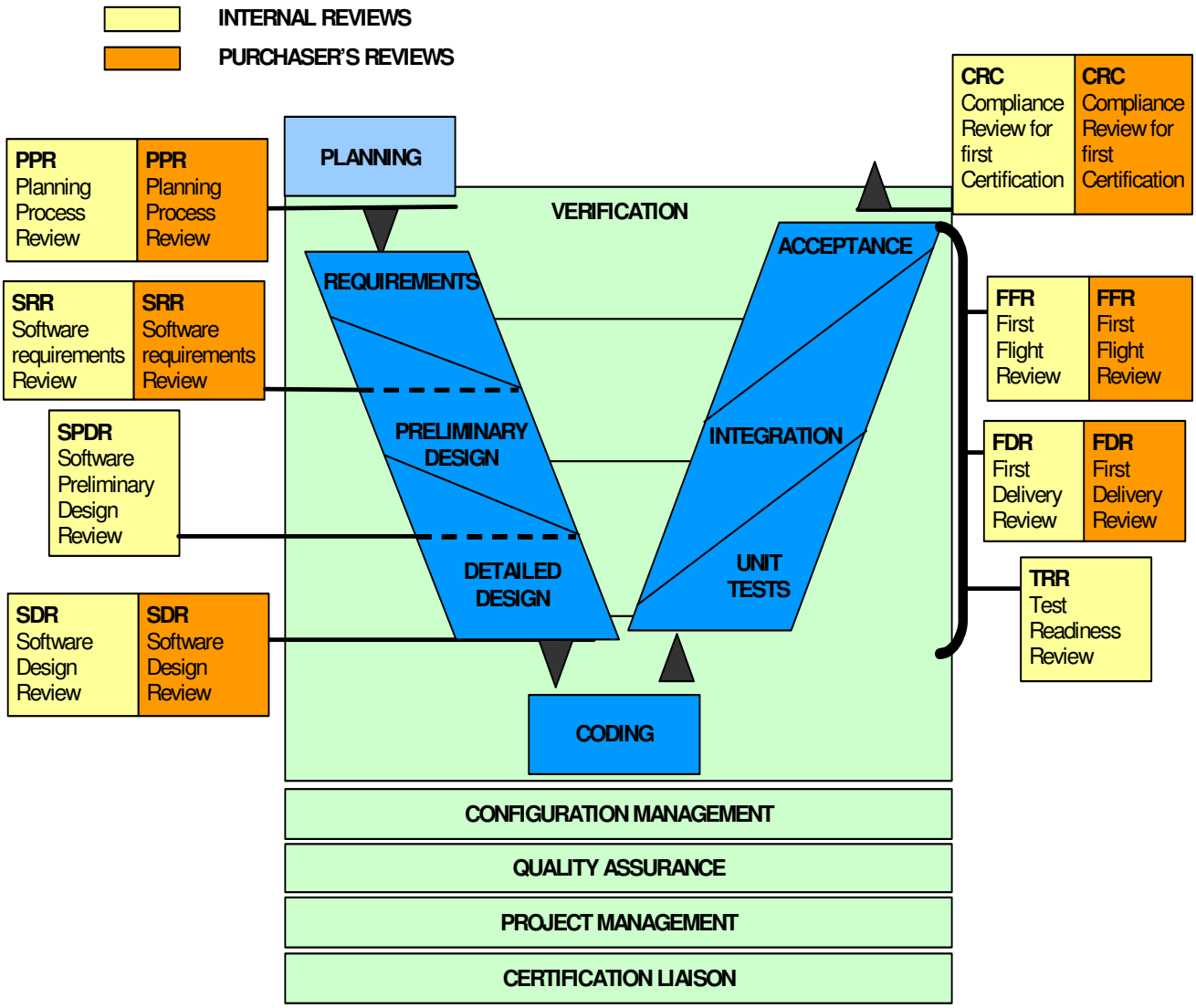
Acceptance Testing Phase, where tests are applied and witnessed to validate that the software faithfully implements the specified requirements. A common mistake in the management of software development is to start by badly managing a development within a V or waterfall lifecycle model, which then degenerates into an uncontrolled iterative model. This is another situation which we have all seen causing a software development to go wrong.


SOFTWARE VERIFICATION AND VALIDATION PROCESSES
REVIEW AND ANALYSES PROCESS
SOFTWARE TESTING PROCESS
SOFTWARE PLANNING PROCESS
INPUT : BUSINESS REQUIREMENTS
OUTPUT: PLAN FOR SOFTWARE ASPECTS OF CERIFICATION [PSAC]
SOFTWARE DEVELOPMENT PLAN [SDP]
SOFTWARE VERIFICATION & VALIDATION PLAN [SVP]
SOFTWARE CONFIGURATION MANAGEMENT PLAN [SCMP]
SOFTWARE QUALITY ASSURANCE PLAN [SQAP]
SOFTWARE BUILD / INTEGRATION PLAN [SIP]
PLANNING PROCESS REVIEW
SOFTWARE REQUIREMENTS PROCESS
INPUT : SYSTEM [HARDWARE & SOFTWARE] REQUIREMENTS
OUTPUT: SOFTWARE REQUIREMENT SPECIFICATION [SRS]
SYSTEM TEST PLAN [STP]
SOFTWARE REQUIREMENTS REVIEW
SOFTWARE PRELIMINARY DESIGN PROCESS
INPUT : SOFTWARE REQUIREMENT SPECIFICATION [SRS]
OUTPUT : PRELIMINARY S/W DESIGN DESCRIPTION [PSDD]
SOFTWARE-SOFTWARE INTRGN TEST PLAN [SITP]
PRELIMINARY DESIGN REVIEW
SOFTWARE DETAILED DESIGN PROCESS
INPUT : SRS and PSDD
OUTPUT: DETAILED S/W DESIGN DESCRIPTION [DSDD]
SOFTWARE UNIT TEST PLAN [SUTP]
DETAILED DESIGN REVIEW
SOFTWARE CODING PROCESS
INPUT : SRS, PSDD and DSDD
OUTPUT : SOURCE CODE [.c / .adb]
SOURCE CODE REVIEW
SOFTWARE DEVELOPMENT PROCESSES
BUILD REVIEW
HARDWARE-SOFTWARE INTEGRATION TESTING
INPUT : SRS and SYSTEM TEST PLAN [STP]
OUTPUT : SOFTWARE VERIFICATION CASES & PROCEDURES [SVCP]
SOFTWARE VERIFICATION REPORTS [SVR]
SVCP, SVR REVIEW
SOFTWARE-SOFTWARE INTEGRATION TESTING
INPUT : PSDD and SOFTWARE-SOFTWARE INTRGN TEST PLAN [SITP]
OUTPUT : SOFTWARE VERIFICATION CASES & PROCEDURES [SVCP]
SOFTWARE VERIFICATION REPORTS [SVR]
SVCP, SVR REVIEW
LOW-LEVEL / UNIT TESTING
INPUT : DSDD and SUTP
OUTPUT : SOFTWARE VERIFICATION CASES & PROCEDURES [SVCP]
SOFTWARE VERIFICATION REPORTS [SVR]
SVCP, SVR REVIEW
SOFTWARE TESTING PROCESS
SOFTWARE CONFIGURATION MANAGEMENT PROCESS
INPUT : SOFTWARE CONFIGURATION MANAGEMENT PLAN [SCMP]
OUTPUT : S/W CONFIGURATION INDEX[SCI], S/W ENVIRONMENT CONFIGURATION INDEX [SECI]
SCMP REVIEW
SOFTWARE QUALITY ASSURANCE PROCESS
INPUT : SOFTWARE QUALITY ASSURANCE PLAN [SQAP]
OUTPUT : REVIEW REPORTS, CHECKLISTS
SQAP REVIEW
CERTIFICATION LIASION PROCESS
INPUT : SOFTWARE ACCOMPLISHMENT SUMMARY [SAS] GUIDELINES
OUTPUT : SOFTWARE ACCOMPLISHMENT SUMMARY [SAS]
SAS REVIEW

4.2 Reviews During VEE Life Cycle Model Of SDLC

Reviews provide an assessment of the accuracy, completeness and verifiability of the software requirements, software architecture and the source code. This activity is applied to the results of software development processes and software verification processes. The difference between reviews and analyses is that ***review provides qualitative assessment of correctness while analyses provide repeatable evidence of correctness***.

A review may consist of an inspection of an output of a process guided by a checklist or similar aid. An analysis may examine in detail the functionality, performance, traceability and safety implications of a software component, and its relationship to other components within the airborne system or equipment.



4.3 Data to Be Made Available For Reviews

DATA	PPR			SRR			SPDR			SDR			TRR			FDR			FFR			CRC		
DO-178B Software Level	A B	C	D	A B	C	D	A B	C	D	A B	C	D	A B	C	D	A B	C	D	A B	C	D	A B	C	D
PSAC	Y	Y	Y																					
SDP	Y	Y	Y	C	C	C	C	C	C	C	C		C	C		C	C	C	C	C	C	C	C	C
SVVP	Y	Y	Y	C	C	C	C	C	C	C	C		C	C		C	C	C	C	C	C	C	C	C
SCMP	Y	Y	Y	C	C	C	C	C	C	C	C		C	C		C	C	C	C	C	C	C	C	C
SQAP	Y	Y	Y	C	C	C	C	C	C	C	C		C	C		C	C	C	C	C	C	C	C	C
SRS	Y	Y		C	C		C	C		C	C		C	C		C	C		C	C		C	C	
SDS	Y	Y		C	C		C	C		C	C		C	C		C	C		C	C		C	C	
SCS	Y	Y	①	C	C	①	C	C	①	C	C		C	C		C	C	①	C	C	①	C	C	①
SRD				Y	Y	Y	C	C	C	C	C		C	C		C	C	C	C	C	C	C	C	C
SPDD							Y	Y	Y	Y	Y		C	C		C	Y	Y	C	C	Y	C	C	C
SDDD										Y	Y		C	C		C	Y		C	C		C	C	
Source Code													Y	Y		Y	Y	Y	C	C	Y	C	C	C
SUTP										Y	Y		Y	Y		Y	Y		C	C		C	C	
SIP / SITP							Y	Y		Y	Y		Y	Y		Y	Y		C	C		C	C	
SAP / SATP				Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		Y	Y	Y	C	C	C	C	C	C
SUTR / SITR													Y	Y		Y	Y		C	C		C	C	
SATR													Y	Y		Y	Y	Y	C	C	C	C	C	C
SVVR / TC				Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		Y	Y	Y	C	C	C	C	C	C
STD				Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		Y	Y	Y	C	C	C	C	C	C
SCI													Y	Y		Y	Y	Y	C	C	C	C	C	C
PECI	Y	Y	Y	C	C	C	C	C	C	C	C		C	C		C	C	C	C	C	C	C	C	C
PR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y
SQAR	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y
SCMR				Y	Y	Y	Y	Y	Y	Y	Y		Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y
SAS																Y	Y	Y	Y	Y	Y	Y	Y	Y

Y (Yes) = The corresponding data need to be submitted to this review

C = Changes to this data need to be submitted to this review

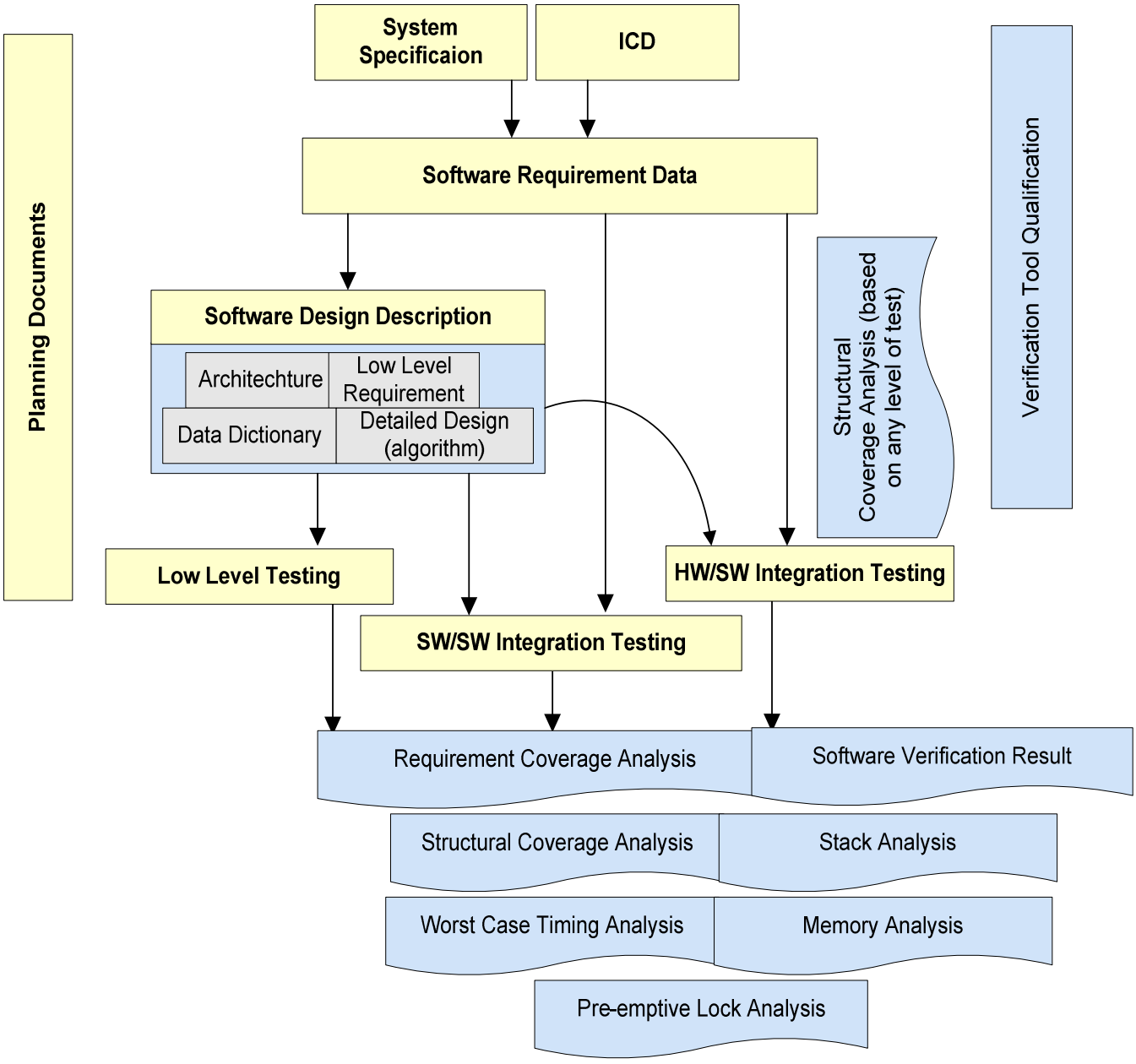
① = Requested, **Y** for PPR and **C** for the other reviews

For level D, SPDD and SDDD's can be merged in one document SDD.

4.4 Verification and Validation

It is the process of ensuring that software being developed or changed will satisfy functional and other requirement (validation) and each step in the process of building the software yields the right products (verification). V&V refers to a set of activities that are aimed at making sure the software will function as required.

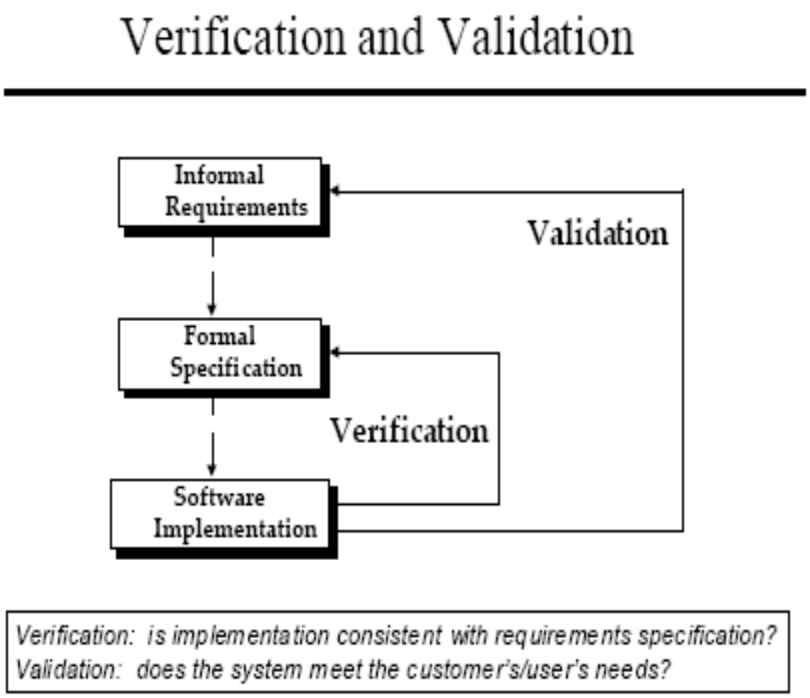
The following diagram depicts the software development and verification phases inter-relationship.



V&V are intended to be a systematic and technical evaluation of the software and associated products of the development and maintenance processes. Reviews and tests are done at the end of each phase of the

development process to ensure software requirements are complete and testable and also the design, code, documentation and data satisfy those requirements.

- Verification refers to the set of activities that ensure that correctly implements a specific function.
- Validation refers to a different set of activities that ensure that software that has been built is traceable to client requirements.



The purpose of verification is to detect defects in a product. Any intermediate work product or integrated subsystem, as well as the full, integrated system is a candidate subject for verification. Consequently, the practice of Continuous verification takes the stance that verification should be an ongoing task rather than being relegated to the end of the development life cycle.

One important aspect of an early verification effort is deciding what verification strategy to use for each of the requirements and components of the overall system.

From this point onwards, the document explains various aspects of software testing as part of verification and validation as mandated by DO-178B. This document mainly focuses on the following levels of testing:

A. Hardware-Software Integration Testing (HSIT).
B. Software-Software Integration Testing (SSIT).
C. Unit Testing (UT).

5.0 SOFTWARE VERIFICATION STRATEGY

Verification of airborne software has two complementary objectives.

01. To demonstrate that the software satisfies its requirements.
02. To demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions, as determined by the system safety assessment process, have been removed.

To achieve the above objective, test cases should be requirements based so all testing activities provide verification of the requirements. Each test case defined in the Software Verification Cases & Procedures (SVCP) documents must be traced as follows:

1. **For HW-SW Integration:** Trace to the Software Requirements Document (High level requirements)
2. **For SW Integration:** Trace to the Software Requirements Document (High level requirements) and the Preliminary Software Design Description (Low level requirements)
3. **For Module Test:** Trace to the Detailed Software Design Description (Low level requirements)

Software verification can be optimized by starting with System Level Tests and then additionally develop Integration Tests and Unit Level Tests to achieve 100% Functional and 100% Structural Coverage.

The basic test stop criteria applicable for all levels are defined below:

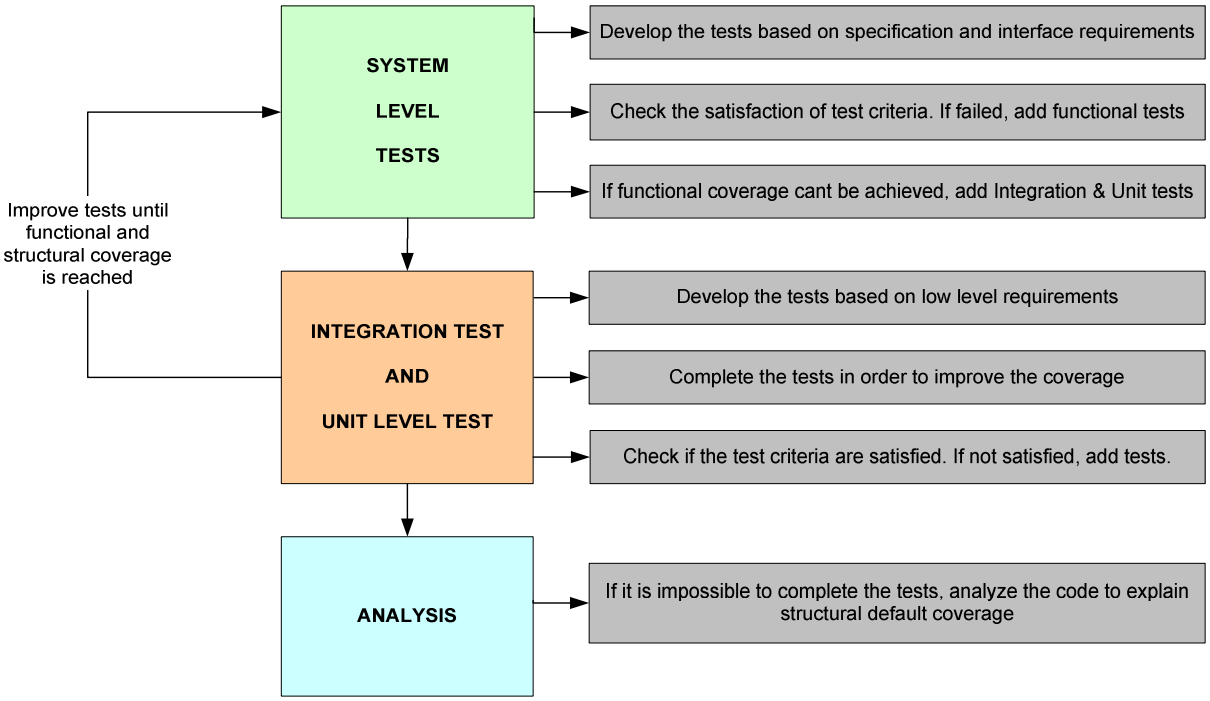
1. 100% of Functional Coverage.
2. 100% of Statement Structural Coverage.

5.1 Software Test Priorities

The priorities on the different type of tests are

LEVEL	SYSTEM TEST	INTEGRATION TEST	UNIT TEST
Test Realization Priority	**1**	**2**	**3**
Tests classes planned	Real Time Functional Performances for External Interfaces	Real Time Functional Performances for External Interfaces	Functional
Functional coverage of HIGH LEVEL requirements and Interfaces requirements	Objective 100%	Adding tests to reach 100% coverage	Adding tests to reach 100% coverage
Functional coverage of LOW LEVEL requirements	-	Objective 100%	Adding tests to reach 100% coverage

This can be further simplified as



5.2 Software Test Plan

The realization of software test priorities can be kicked-off with preparation of the Software Test Plan [STP]. This STP provides the guidelines to verify a software product in a phased incremented [from unit test to system test] or decremented [system test to unit test] manner. Main contents of a typical STP under DO-178B certification requirement are as follows:

SOFTWARE TEST ENVIRONMENT	FORMAL QUALIFICATION TEST IDENTIFICATION	DATA RECORDING, REDUCTION & ANALYSIS
➢ Software items ➢ Hardware and firmware items ➢ Proprietary & Govt. rights ➢ Installation, testing, and control	➢ General test requirements • General test philosophy • General qualification method • Software integration • Regression tests • Object files used for tests ➢ Test classes ➢ Test levels and Test definitions • Software Validation Test • HW/SW Integration testing • SW/SW integration testing • CSU testing (Low Level Test) • Static code analysis ➢ Test schedule	➢ Data recording ➢ Test Results ➢ Analysis of results

Software test plan also contains standards that define the guidelines and procedures to be followed when conducting verification activities. The following items direct the creation of a software test plan.

5.2.1 Software Verification Process Inputs

The inputs to the software verification process are as follows:

- Interface Control Definition (ICD)
- Requirements
  - System Requirements Document (SRD) consisting both Hardware and Software
- Design
  - Software Design Description (SDD)
- Functional Source Code

5.2.2 Software Standards

Verify adherence to the following standards:

- RTCA DO-178B Section 6.0 .
- Engineering Design Manual (EDM), Section 3.3 (Verification Activities)
- Software Requirements Standards
- Software Design Standards
- Software Code Standards

5.2.3 Independence Criteria

- The verification team shall be independent of the engineering design teams.
- Those responsible for performing the verification activities shall be identified.
- Independence shall be maintained for software updates, software test creation and document reviews.

This above satisfies the required verification independence in DO-178B for Level A criticality software.

5.2.4 Review Procedures

The following review procedures shall be applied:

1. **DOCUMENT WALKTHROUGH :** For documents that do not have a specific document checklist
   - Read document to identify readability and content conflicts.
   - Complete a document walkthrough.
   - Include in the document folder for verification record purposes.
   - Deviations or problems identified or corrected using the problem reporting process as detailed in SCMP.
2. **CODE WALKTHROUGH**
   - Create a module folder for relevant code module.
   - Complete a Software Module Inspection Checklist.
   - Deviations or problems identified or corrected using the problem reporting process as detailed in SCMP.

3. UNIT TEST WALKTHROUGH

- UT walkthroughs are conducted after each UT is complete to verify the results and test cases selected.
- Create a module test folder.
- Complete a Software Module Test Walkthrough Checklist.
- Deviations or problems identified or corrected using the problem reporting process as detailed in SCMP.

4. SOFTWARE INTEGRATION REVIEW

A review of the integration process shall be accomplished by use of the Software Integration Checklist using the data defined below:

- **Linking data:** The information required to link the software components into the integrated software.
- **Loading data:** The information required to load the software into memory after integration.
- **Memory map:** The layout of the memory after software has been loaded detailing relevant addresses.

The following shall be considered as defined by the checklist:

- Incorrect hardware address
- Memory overlaps
- Missing software components

NOTE: The level of coverage shall also be determined by performing the following reviews and analysis

1. **Inspection or peer reviews of all procedures and tests sequences.**
2. **Inspection of all test results.**

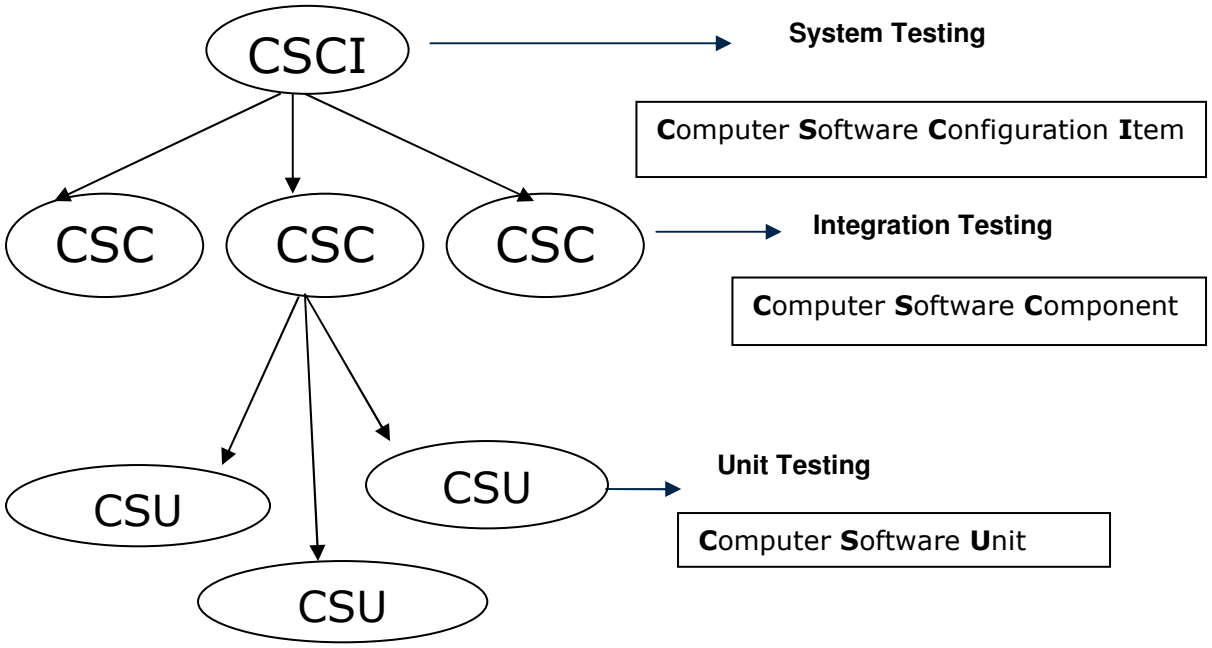
6.0 SOFTWARE TESTING PROCESS

6.1 Software Testing Levels

The software testing can be broken down into following levels as shown in the diagram. For the purpose of documentation following terms are used for segregation level of software functions between HSIT, SSIT and UT.

01. **Computer Software Configuration Item (CSCI):**
    It denotes the group of software treated as a single entity by a configuration management system.
02. **Computer Software Component (CSC):**
    It denotes functional block of software subset that is grouped for specific functionality.
03. **Computer Software Unit (CSU):**
    It denotes lowest level public function software unit.

The inter relationship between the CSCI; CSC and CSU are as shown below.

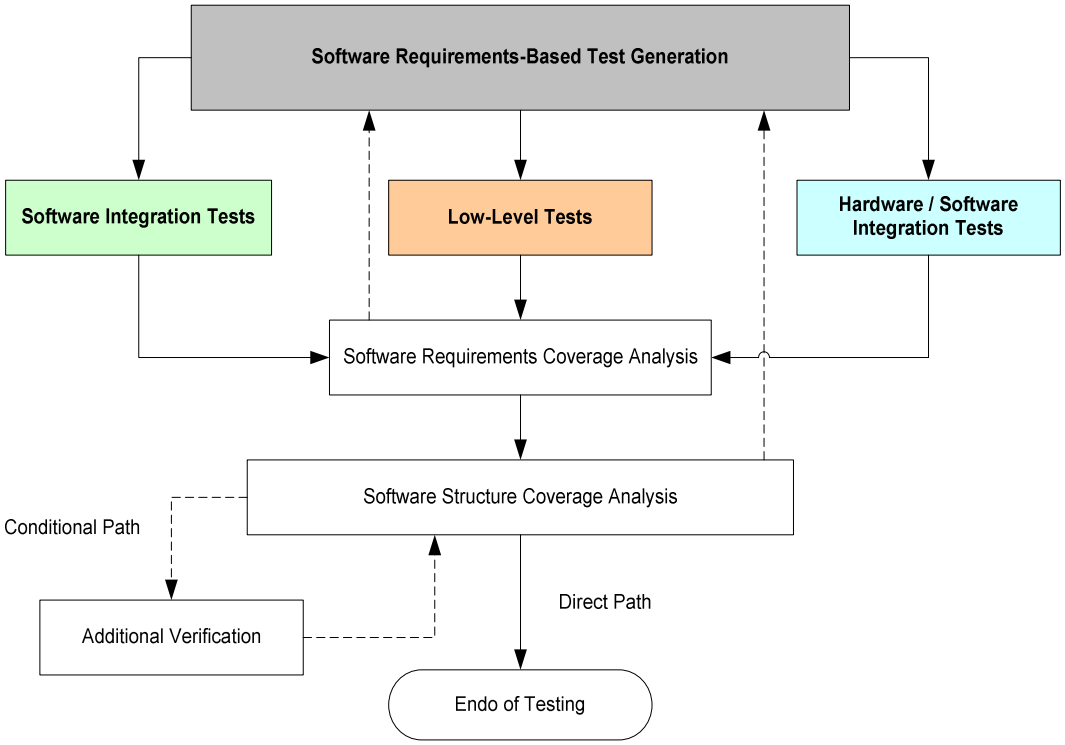


6.2 Requirements Based Testing (RBT)

The software verification test cases are to be created based on the software requirements specification. Test cases should be developed to verify correct functionality and to establish conditions that reveal potential errors.

01. The first step is to develop functional and robustness tests to completely test and verify the implementation of the software requirements.
02. The second step is to measure coverage (functional and structural). The measure of structural coverage will help provide an indication of the software verification campaign completion status.

The tests stop criteria is not limited to a specific step but rather applied for all tests. For example, some high level requirements can be covered by integration tests, i.e., structural coverage are measured on all tests levels. Software requirements coverage analysis should determine what software requirements were not tested. Structural coverage analysis should determine what software structures were not exercised.



6.3 Requirements Based HSIT Objectives

The objective of HSIT is to ensure that software in the target computer will satisfy the high-level requirements. Typically HSIT should be limited to areas which require hardware interfaces like

A. Hardware Interface Requirement.
B. Power Up/Down sequence.
C. NV RAM storage.
D. Real Tim Process (Scheduler or Interupt) timing requirements.
E. Protocol Testing (E.g. ARINC, Boot Loader)
F. Worst Case Timing and Stack Margin Test.

NOTE: *It is noted that a lot of projects perform the integration test on the black box on System Test Environment (or closer). While the practice is not incorrect, the task calls out for more skills, environment availability and the constraints around it, thereby affecting the schedule, effort and cost. Performing SSIT and selected on HSIT could be best suited, subjected to the approvals of the certification plans.*

The following shall be considered when writing tests to verify high level requirements:

1. Does the high level requirement that you are defining a test case for meet the system requirement?

   To verify this:

   - Check the results from the review of the Software Requirements (SRD Checklist).
   - Trace back to the system requirements.

2. Is the accuracy correct, if applicable?

   To verify this:

   - Check the results from the review of the Software Requirements.
   - Trace back to the system requirements.

3. Is the requirement consistent with others? i.e. no conflicts or contradictions.

   To verify this:

   - Check the results from the review of the Software Requirements (SRD Checklist).
   - Trace back to the system requirements.

4. Has compatibility with the target system been considered?

   This will typically be verified during the execution of informal tests.

5. Does this requirement conform to the Software Requirements Standards?

   To verify this:

   - Check the results from the review of the Software Requirements (SRD Checklist).
   - Verify against the Software Requirements Standards.

6. Does the requirement trace to the relevant system requirement?

   To verify this:

   - Check the results from the review of the Software Requirements (SRD Checklist).
   - Verify the software requirement traces to the system requirement from the SRD.

7. Are any algorithms associated with this requirement accurate and have correct behaviour?

   To verify this:

   - Check the results from the review of the Software Requirements (SRD Checklist).
   - Run an informal test to verify algorithm.

6.4 Requirements Based SSIT Objectives

The objectives of requirements-based software/software integration testing is to ensure that software components interact correctly with each other and satisfy the software requirements and software architecture.

SW/SW Integration Test involves:

1. Testing the interaction ***within*** the software components
2. Testing the interaction ***between*** the software components

The term "component" here refers to the functional block of software subset that is grouped for specific functionality. Example: Scheduler, Mode Logic, ARINC, NVM, BIT, Control Loop etc.

The following shall be considered when writing tests to verify software architecture:

1. Does the relevant piece of architecture that you are defining test cases for meet the software requirement?

   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Trace back to the software requirements.

2. Software components and the relationships between them i.e. data and control flow shall be verified to be correct. This is achieved through Software Integration as detailed in these standards in section 5.5.

   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Conduct a Software Integration test (Section 5.5).

3. Has compatibility with the target system been considered?

   This will typically be verified during the execution of informal tests.

4. Does the relevant architecture conform to the Software Design Standards?

   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Verify against the Software Design Standards.

5. Is the architecture as defined verifiable? i.e. Can you define test cases to adequately test this piece of software architecture as defined by these standards and the SVP.

   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Conduct informal tests to make sure the architecture is verifiable.

6.5 INPUTS AND OUTPUTS FOR HSIT AND SSIT

INPUTS	OUTPUTS
➢ Software Requirements Data ➢ Software Design Data including Software Architecture ➢ Testing Standards, Software Verification Plan ➢ Hardware Interface document as applicable in project ➢ Communication Interface document example, ARINC, CAN etc as applicable in project ➢ Data Dictionary – Can be present as part of REQR.. / ICD etc	➢ Software Verification Cases and Procedure Document ➢ Test Case, Test Procedure ➢ Traceability Matrix, HLR ⇔ Test Case, Test Case ⇔ Test Procedure. In some cases the test traces to LLR also ➢ Test Report ➢ Review Record/Checklist ➢ Configuration records

6.6 Requirements Based UT Objectives

The objectives of Low level testing are:

1. To verify compliance of each component with respect to its low -level requirements
2. To verify the response of each component to normal as well as abnormal conditions
3. To generate the measure of structural coverage as per DO-178B applicable level. It should be noted that the structural coverage can be attained at any level of test and not necessarily by low level test.

As per DO-178B Section 6.4, if a test case and its corresponding test procedure are developed and executed for hardware/software integration tests or software integration testing and satisfy the requirements-based coverage and structural coverage, it is ***not necessary to duplicate the test for low-level testing***. Substituting nominally equivalent low-level tests for high-level tests may be less effective due to the reduced amount of overall functionality tested. Hence the applicability of performing low level test should be documented in the SVP.

The following shall be considered when writing tests to verify low level requirements:

1. Does the low level requirement that you are defining a test case for meet the software requirement?

   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Trace back to the software requirements.

2. Is the accuracy correct, if applicable?

   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Trace back to the software requirements.

3.c) Is the requirement consistent with others? i.e. no conflicts or contradictions.

   To verify this:

- Check the results from the review of the Software Requirements (SRD Checklist).
- Trace back to the system requirements.

4. Has compatibility with the target system been considered?

   This will typically be verified during the execution of informal tests.

5. Does this requirement conform to the Software Design Standards?

   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Verify against the Software Design Standards.

6. Does the requirement trace to the relevant software requirement?

   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Verify the low-level requirement traces to the software requirement from the SDD.

7. Are any algorithms associated with this requirement accurate and have correct behaviour?

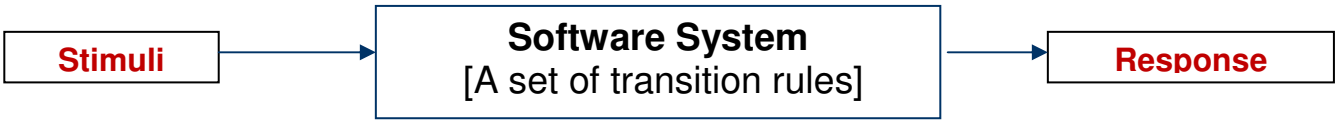
   To verify this:

   - Check the results from the review of the Software Design (SDD Checklist).
   - Run an informal test to verify algorithm.

6.7 INPUTS AND OUTPUTS FOR UT

INPUTS	OUTPUTS
FOR TEST CASE GENERATION: ➢ Software Design Description (SDD). ➢ Low Level Requirement (this is usually part of SDD document). ➢ Data Dictionary. ➢ Testing Standards, Software Verification Plan. ➢ Any tool used for the traceability management (Test Case ⇔ LLR) example DOORS as applicable. **FOR TEST PROCEDURE GENERATION:** ➢ Test Case ➢ Tool and its supporting environment for writing the test script if any (Example, Rational Test Real Time with Target Deployment Port on specific target) ➢ Testing Standards, Software Verification Plan.	➢ Software Verification Cases and Procedure Document. ➢ Test Case. ➢ Test Procedure or, Scripts ➢ Traceability Matrix, LLR ⇔ Test Case, Test Case ⇔ Test Procedure. ➢ Test Report. ➢ Test Report (Pass/Fail status). ➢ Coverage Report. ➢ Review Record/Checklist. ➢ Configuration records.

7.0 SYSTEM TESTING

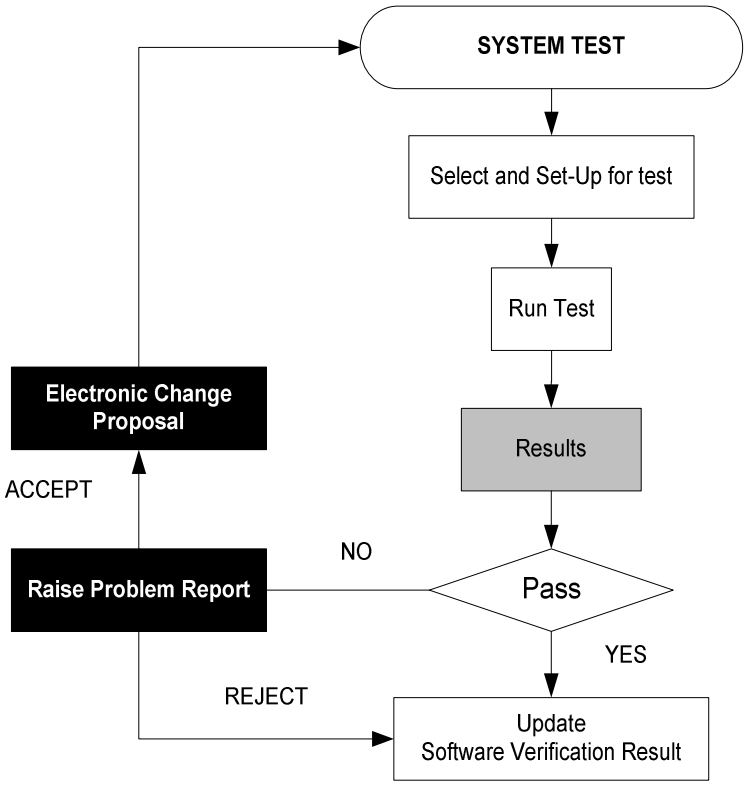


A typical scenario of the system is as shown above. Any system accepts *stimuli* and gives a *response*. A stimulus is nothing but user inputs. A software system which consists of a set of transition rules is nothing but implementation of functionality in a methodical manner following stringent process.

System testing is the process of testing carried out in an integrated hardware and software environment to verify the behavior of the complete system i.e. System Testing is the verification performed by integrating the Hardware and Software of a system simulating the actual use environment along with other system elements like target hardware, field data etc.

System testing basically involves 3 basic steps and they are

- Selecting the inputs,
- Applying it to the system and
- Verifying the outputs.



System testing deals with the verification of the high level requirements specified in the System Requirements (Segment) Specification/Data. The verification tests involved in System Testing are conducted on the target hardware.

The testing methodology used is Black Box testing (Module under test is treated as a black box (without seeing the code), to which input is applied and corresponding output is verified i.e. only functionality is checked).

The test cases are defined based on the system and high level requirements only.

7.1 ETVX Criteria for System Testing

Entry Criteria:

- Completion of Software Integration.

Inputs:

- System Requirements Document.
- Software Requirements Data.
- Software Design Document.
- Software Verification Plan.
- Software Integration Documents.

Activities:

- Define test cases and procedures based on the system level requirements.
- Execute the tests on the defined hardware-software environment and obtain the results

Exit Criteria:

- Successful completion of the integration of the Software and the target Hardware.
- Correct performance of the system according to the requirements specified.

Outputs:

- System Testing Reports.
- Software Test Cases and Procedures [SVCP] and Software Verification Results [SVR].

7.2 System Testing Objectives:

General:

- Scaling and range of data as expected from hardware to software,
- Correct output of data from software to hardware,
- Data within specifications (normal range),
- Data outside specifications (abnormal range/robustness),
- Boundary data,
- Correct acquisition of all data by the software,
- Timing,
- Interrupt processing,
- Correct memory usage (addressing, overlaps, etc.),
- State transitions.

Cross channel communications

All cross channel communication are verified. The considerations for this are:

- Correct transmitting and receiving of data,
- Correct composition of data packets,
- Correct timing.

Timing

The timing constraints imposed on the system as defined in the System Specification are checked.

Interrupt processing

All interrupts are verified independently from the initial request through full servicing and onto completion. Test cases are specifically designed in order to adequately test interrupts and the following considerations are taken into account:

- Conformity of the performance of each interrupt handling procedure to the requirements,
- Correct assignment and handling of priorities for each interrupt,
- Correct processing of the interrupts.

Analysing whether a high volume of interrupt arrivals at critical junctures cause problems in system functioning or performance.

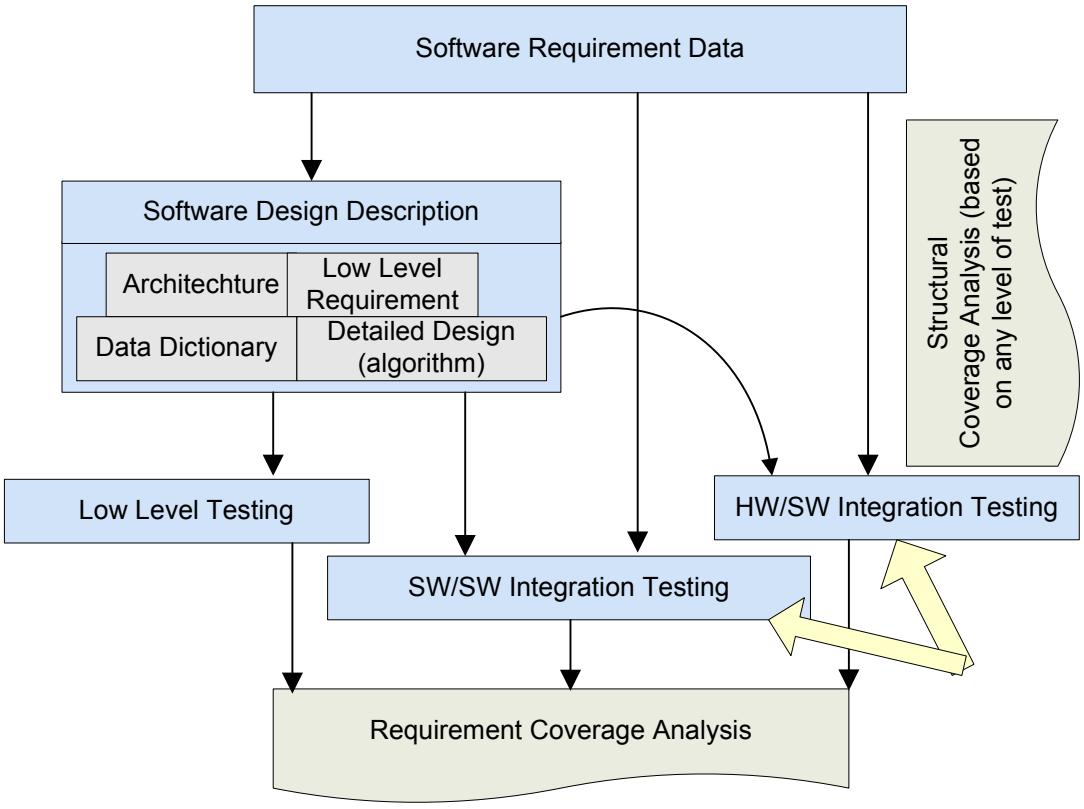
All *system tests* and *acceptance tests* have to be executed in the target environment. This may be facilitated by developing and executing system tests in the host environment, then porting and repeating them in the target environment. Target dependencies may prevent system tests developed in a host environment from being repeated without modification in a target environment.

Careful structuring of software and tests will minimize the effort involved in porting system tests between the host and target environments. However, fewer developers will be involved in system testing, so it may be more convenient to forgo execution of system tests in the host environment. The final stage of system testing, acceptance testing, has to be conducted in the target environment. Acceptance of a system must be based on tests of the actual system, and not a simulation in the host environment.

8.0 INTEGRATION TESTING

Integration testing is the process of testing in which, software and/or hardware components are combined and tested progressively until the entire system has been integrated. Integration is performed to verify the interactions between the modules of a software system. It deals with the verification of the high and low level software requirements specified in the Software Requirements Specification/Data and the Software Design Document.

Integration Testing refers to the test cases that are made against the High Level Requirement i.e., Software Requirement Data (SRD). In some case the test cases are also based on the Software architecture. Integration Tests maps to DO-178B Hardware/Software Integration Testing and Software-Software Integration Testing as depicted in the diagram below.

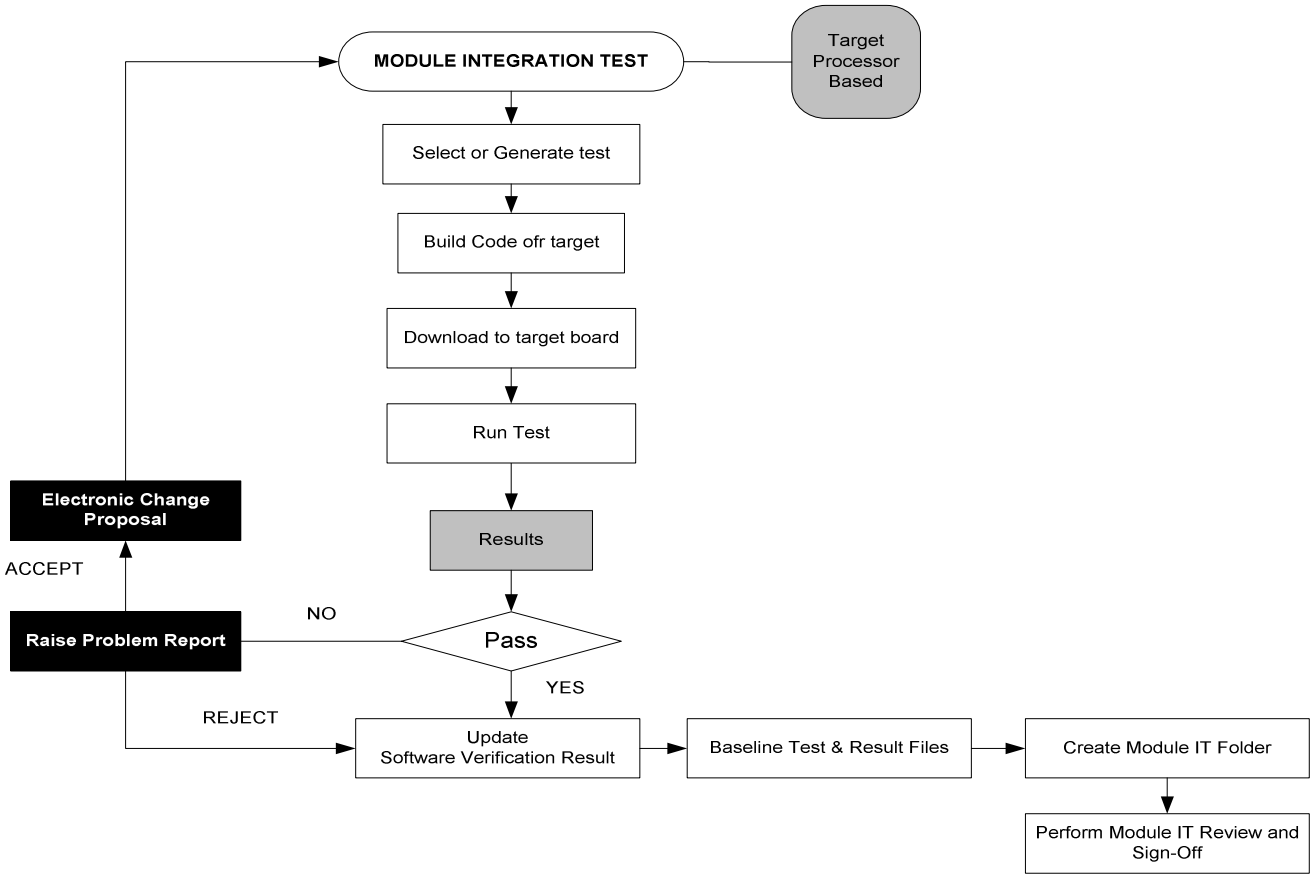


It's a systematic technique for constructing the program structure while conducting tests to uncover errors associated with interfacing. The objective is to take the unit tested modules and build a program structure that has been dictated by design. There are often tendencies to attempt non-incremented integration i.e. to construct the program using a big-bang approach. All modules are combined in advance and the entire program is tested as a whole and chaos usually results. A set of errors is encountered. Correction is difficult because isolation causes is complicated by the vast expansion of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small segments, where errors are easier to isolate and correct. Interfaces are more likely to be tested completely and a systematic test approach may be applied. There are some incremental methods like The integration tests are conducted on a system based on the target processor. The methodology used is Black Box testing. Either bottom-up or top-down integration can be used. Test cases are defined using the high level software requirements only.

Software integration may also be achieved largely in the host environment, with units specific to the target environment continuing to be simulated in the host. Repeating tests in the target environment for confirmation will again be necessary. Confirmation tests at this level will identify environment specific problems, such as errors in memory allocation and de-allocation. The practicality of conducting software integration in the host environment will depend on how much target specific functionality there is. For some embedded systems the coupling with the target environment will be very strong, making it impractical to conduct software integration in the host environment. Large software developments will divide software integration into a number of levels. The lower levels software integration could be based predominantly in the host environment, with later levels of software integration becoming more dependent on the target environment.

If software only is being tested then it is called Software *Software Integration Testing [SSIT]* and if both hardware and software are being tested then it is called Hardware *Software Integration Testing [HSIT]*.



8.1 ETVX Criteria for Integration Testing

Entry Criteria:

- Completion of Unit Testing

Inputs:

- Software Requirements Data
- Software Design Document
- Software Verification Plan
- Software Integration Documents

Activities:

- Define test cases and procedures based on the High and Low level requirements
- Combine low-level modules builds that implement a common functionality
- Develop a test harness
- Test the build
- Once the test is passed, the build is combined with other builds and tested until the system is integrated as a whole.
- Re-execute all the tests on the target processor based platform, and obtain the results

Exit Criteria:

- Successful completion of the integration of the Software module on the target Hardware
- Correct performance of the software according to the requirements specified

Outputs:

- Integration test reports
- Software Test Cases and Procedures [SVCP].

8.2 I/O for Integration Testing

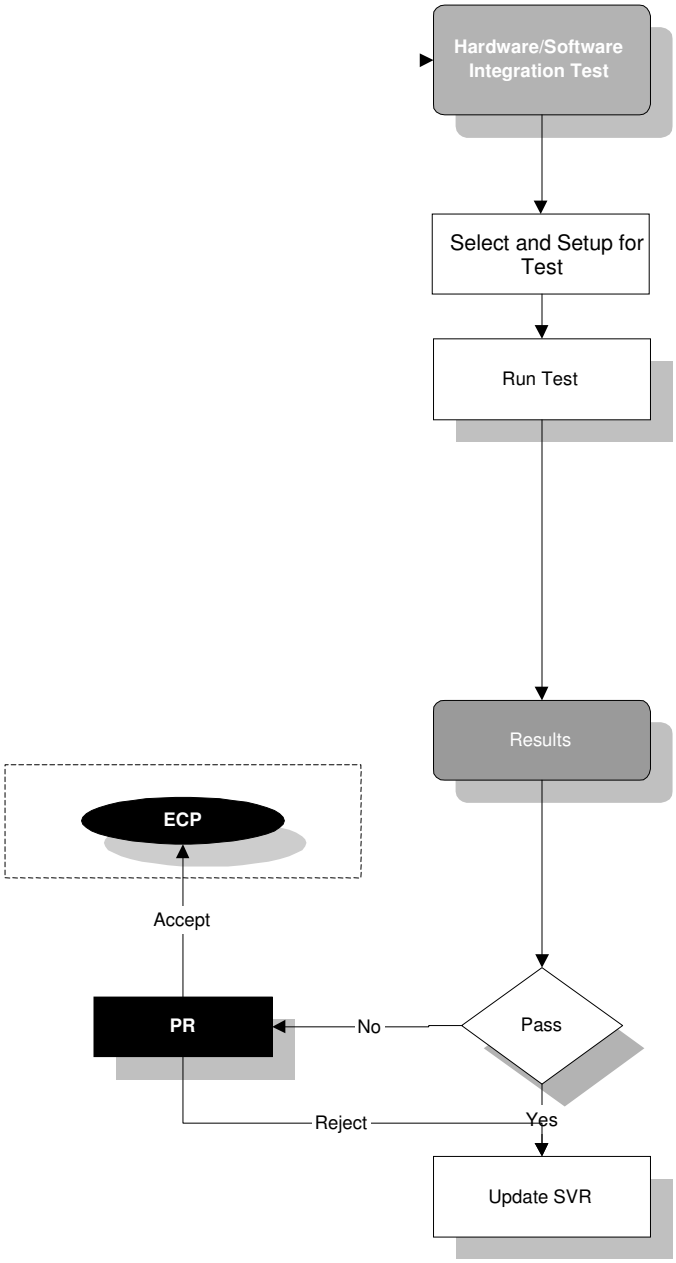
INPUTS	OUTPUTS
➢ Software Requirements Data ➢ Software Design Data including Software Architecture ➢ Testing Standards, Software Verification Plan ➢ Hardware Interface document as applicable in project ➢ Communication Interface document example, ARINC, CAN etc as applicable in project ➢ Data Dictionary – Can be present as part of REQR.. / ICD etc	➢ Software Verification Cases and Procedure Document ➢ Test Case, Test Procedure ➢ Traceability Matrix, HLR ⇔ Test Case, Test Case ⇔ Test Procedure. In some cases the test traces to LLR also ➢ Test Report ➢ Review Record/Checklist ➢ Configuration records

8.3 Integration Testing Objectives

- Correct with software requirements
- Correct control flow
- Correct memory usage.
- Correct data flow
- Correct timing

8.4 Hardware-Software Integration Testing (HSIT)

It is the testing of the Computer Software Components [CSC] operating within the target computer environment, and on the high-level functionality. It concentrates on the behavior of the integrated software developed on the target environment.



Hardware software integration deals with the verification of the high level requirements. All tests at this level are conducted on the target hardware.

1. Black box testing is the primary testing methodology used at this level of testing.
2. Define test cases from the high level requirements only
3. Test must be executed on production standard hardware (on target)

Items to consider when designing test cases for HW/SW Integration:

1. Correct acquisition of all data by the software.
2. Scaling and range of data as expected from hardware to software.
3. Correct output of data from software to hardware.
4. Data within specifications (normal range).
5. Data outside specifications (abnormal range).
6. Boundary data.
7. Interrupts processing.
8. Timing.
9. Correct memory usage (addressing, overlaps etc.).
10. State transitions.

8.4.1 Errors Revealed by HSIT

This testing method should concentrate on error sources associated with the software operating within the target computer environment, and on the high-level functionality. The objective of requirements-based hardware/software integration testing is to ensure that the software in the target computer will satisfy the high-level requirements.

Typical errors revealed by this testing method include:

- Incorrect interrupt handling.
- Failure to satisfy execution time requirements.
- Incorrect software response to hardware transients or hardware failures, for example, start-up sequencing, transient input loads and input power transients.
- Data bus and other resource contention problems, for example, memory mapping.
- Inability of built-in test to detect failures.
- Errors in hardware/software interfaces.
- Incorrect behavior of feedback loops.
- Incorrect control of memory management hardware or other hardware devices under software control.
- Stack overflow.
- Incorrect operation of mechanism(s) used to confirm the correctness and compatibility of field-loadable software.
- Violations of software partitioning.

8.4.2 Interrupt Processing

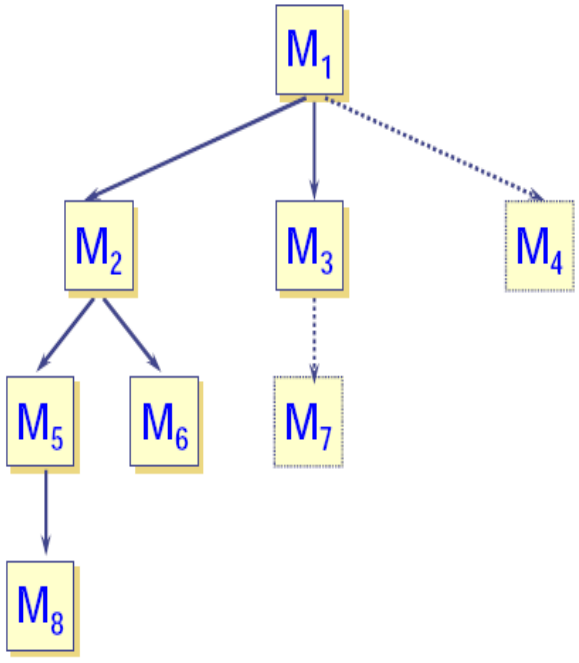
All interrupts will be verified independently from initial request through full servicing and onto completion. Test cases will be specifically designed in order to adequately test interrupts. The following will be considered when designing test cases for interrupts:

1. Does performance of each interrupt handling procedure conform to requirements?
2. Are interrupt priorities correctly assigned and handled?
3. Is processing for each interrupt correctly handled?
4. Does a high volume of interrupts arriving at critical times create problems in function or performance?

8.5 Software-Software Integration Testing (SSIT)

It is the testing of the Computer Software Component operating within the host / target computer environment while simulating the entire system [other CSC's], and on the high-level functionality. It concentrates on the behavior of a CSC in a simulated host / target environment. The approach used for Software Integration can be a top-down, a bottom-up approach or a combination of both.

8.5.1 Top-Down Approach



It's an incremental approach to construction of the program structure. Modules are integrated by moving downward through the control hierarchy beginning with the main control module. Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Depth-first integration integrates all modules on a major control path of the structure as displayed in the diagram:

Selection of a major control path is application specific. For example, selecting the left hand path, modules M1, M2 and M5 would be integrated first. Next would be M8 or (if necessary for the proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. Breadth-first integration incorporates all modules directly subordinate at each level, moving across the structure horizontally. In the diagram above, modules M2, M3 and M4 would be integrated first. The next control level, M5, M6 and so on, follows.

In this approach the module integration process is performed in a series of five steps:

1. The main control module is used as a test driver and the stubs are substituted for all modules directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e. depth-or breadth first) the subordinate stubs are replaced one at a time with actual modules.
3. Tests are conducted as each module is integrated.
4. On completion of each set of tests, another stub is replaced with real module.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step2 until the entire program structure is built. The top-down strategy sounds relatively uncomplicated, but in practice, logistical problems arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing and therefore no significant data can flow upward in the program structure.

The tester is left with these choices

1. Delay many tests until stubs are replaced with actual modules.
2. Develop stubs that perform limited functions that simulate the actual module.
3. Integrate the software from the bottom of the hierarchy upward.

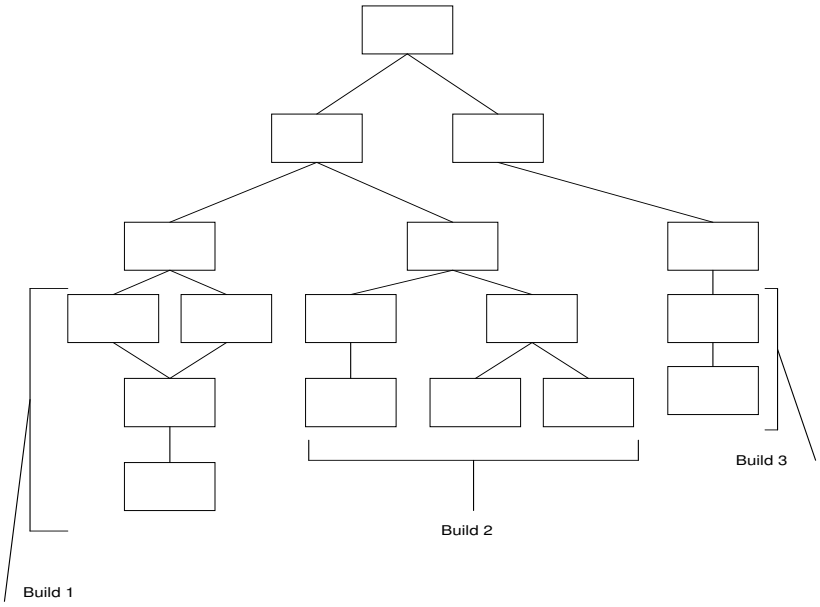
The first approach causes us to lose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors which, tends to violate the highly constrained nature of the top down approach. The second approach is workable but can lead to significant overhead, as stubs become increasingly complex.

8.5.2 Bottom-Approach Approach

Bottom-up integration begins construction and testing with modules at the lowest level in the program structure. In this process the modules are integrated from the bottom to the top. In this approach processing required for the modules subordinate to a given level is always available and the need for the stubs is eliminated.

This integration test process is performed in a series of four steps

1. Low-level modules are combined into clusters that perform a specific software sub-function.
2. A driver is written to co-ordinate test case input and output.
3. The cluster or build is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.



As integration moves upward, the need for separate test drivers lessens. In the fact, if the top two levels of program structure are integrated top-down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified. Integration follows the pattern illustrated below. Modules are combined to form builds 1, 2 and 3. Each of the builds is tested using a driver. As integration moves upward, the need for separate test drivers lessens.

Note: If the top two levels of program structure are integrated Top-down, the number of drivers can be reduced substantially and integration of builds is greatly simplified.

8.5.3 Requirements based Software-Software Integration Testing

This testing method should concentrate on the inter-relationships between the software requirements, and on the implementation of requirements by the software architecture. The objective of requirements-based software integration testing is to ensure that the software components interact correctly with each other and satisfy the software requirements and software architecture. This method may be performed by expanding the scope of requirements through successive integration of code components with a corresponding expansion of the scope of the test cases.

8.5.4 Errors Revealed by HSIT

Typical errors revealed by this testing method include:

- Incorrect initialization of variables and constants.
- Parameter passing errors.
- Data corruption, especially global data.
- Inadequate end-to-end numerical resolution.
- Incorrect sequencing of events and operations.

8.6 Integration Testing Guidelines

8.6.1 Test Case Template

The test case template should contain:

- Test Case Identifier: Unique ID
- Test Case Author/Version/Date:
- Test Type: Normal/Robustness
- Test Description
- Input(s)
- Initial Condition: Defines the initial state of test setup & environment
- Expected Output(s)
- Pass/Fail Criteria
- Requirement(s) Traced

Sample format:

TC Identifier	Test Type	Description	Input	Output	Initial Condition(s)	Expected Result(s)	Pass/Fail Criteria	Requirements Traced
Case 1	Normal	<To mention the high level description / objective of test without repeating the input/output values>	Input 1 Input 2	Output 1 Output 2		See Output Signals	actual results equal expected result	Req-1
Case 2	Normal	Example, Tests the timer reset condition Tests for FT condition for MC/DC combination on Input 1, Input 2				See Output Signals	actual results equal expected result	Req-1
Case 3	Normal					See Output Signals	actual results equal expected result	Req-2
Case 4	Robustness					See Output Signals	actual results equal expected result	Req-3

Note: The Author/Version/Date can be maintained by CM tool usage and need not be part of template itself)

8.6.2 Test Procedure Template

The test procedure in terms of environment setup is usually documented in Software Cases and Procedure Document based on test environment. The test procedure itself is usually implemented as script based on the test case. It is acceptable to have the test procedure implementing multiple test cases. The traceability of test procedure to test case and vice-versa should be maintained. The test procedure template should contain:

- Test Procedure Name (or ID)
- Test Procedure Author/Version/Date: (Note the Author/Version/Date can be maintained by CM tool usage and need not be part of template itself)
- Test Environment
- Test Case(s) Traced

8.6.3 Common Testing Guidelines

01. The initial condition of the output should be of inverted than the expected result in the test case.
02. When checking the output, ensure that output values are toggled in different test case to prove the affect of input
03. For a float input, values should be given greater by 1 LSB (or possible constraint due to HW) at the operator condition. The floating point precision should be 0.000001 or, as defined by project plan
04. The float comparison on the target computer should be given as 95% of expected value
05. For data input from external analog sensor simulator, LSB should be equal to the minimum step possible
06. Tolerances for analog values should be noted either in requirement or, software architecture as this will be used for Pass/Fail criteria for analog signals
07. For certain requirement, it is acceptable to modify the software, example for worst case timing requirement, it may be required to toggle any output pins, but this should be documented and made part of test procedure
08. The requirement based test should cover the MC/DC combination of requirement itself. For the cases not possible to achieve requirement coverage at MC/DC level, justification should be given. The MC/DC coverage of requirement should be irrespective of Software Level. Note that this requirement should be agreed in the project plan.
09. For the test covering the boundary value of the input should be categorized as "Normal Range" test. For the test covering the outside the boundary value of the data dictionary of the input, should be categorized as "Robustness Range" test.
10. For the Software Requirement based test, data type should not be based on source code and design but based on requirement and/or Interface control document. For Low Level requirement based testing, data type should be based on design and not to the source code.
11. The testing on the relational operators should be tested on the boundary above or below the relation based on the accuracy of data type (usually 1 LSB for integer and least precision supported on target for float)
12. Additional robustness cases can include:
    i. Setting of invalid data like data being stale, drop-outs, data outside the range and/or tolerance
    ii. Periodic warm start cases, and warm/cold starts, exceptions
    iii. Transition of various software mode of operation intermittently including invalid/valid transitions
    iv. Creating scenarios of arithmetic exceptions
    v. Forcing overrun cases on memory, timing
    vi. Memory corruption cases, example power shutdown while forcing SW to write data to NVM

8.7 Test Allocation Strategy

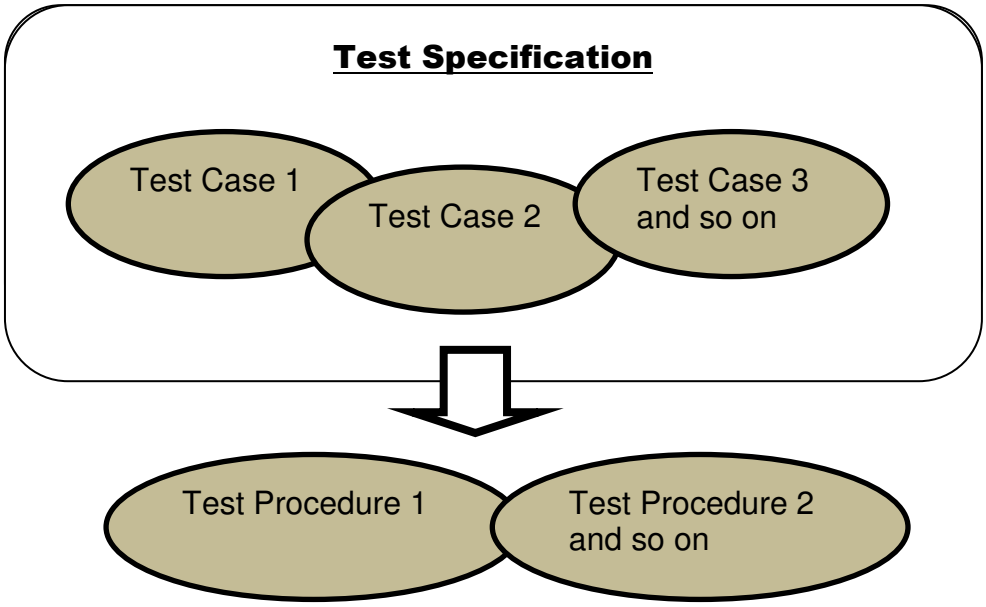
Test allocation plays an important role in terms of overall testing. The test allocation should meet at least the following requirement:

1. Functional/Logical grouping of requirement
2. Human Resource – Allocation based on skill
3. HW/SW Resource – Test related to usage of console Vs standalone for optimal usage of hardware
4. Prioritization of testing the functionality – Test the input interface first then communication then system functionalities and so on, again should be assessed on the schedule, example Safety of Flight Requirement to be completed first and so on
5. Regression test allocation
6. Resource deployment plan – Number of resources Vs Timeline

Point 2 through 6 are subjective to the program need, more focus in the further section is made on approach of Functional/Logical grouping of requirement

8.7.1 Functional/Logical grouping of requirement

First and foremost, a thorough review of the complete requirement is required, with presence of stakeholders – V&V Lead, Developer with System experience to create the test specification. A single test specification may contain multiple test cases. The test procedure, usually in form of script can be created to trace single or, multiple test cases. The test specification to test case to test procedure relationship is depicted below.



A sample of test specification (also called as grouping or scenarios in some cases) is defined as follows:

<ID>	Test Specification	Brief approach	Test Environment	Test Method
TI-001	1. Power Up Test		Eg. Target Board with Test Console Target board with Emulator etc	SW Integration Test, HW/SW Integration Test, Inspection, or Analysis.
TI-002	2. Discrete Input Test			
TI-003	3. Communication Test			
	3.1 Bus Selection			
	3.2 ARINC 429 Input Bus1			
	3.3 ARINC 429 Input Bus2			
	3.4 ARINC 429 Output Bus1			
	3.5 ARINC 429 Output Bus2			
TI-004	4. System Mode and State Test			
	And so on			

8.7.2 Regression Test

The need of regression test is used to provide confidence of deliverable or, validate any 'change' impact on the overall functionality.

- To check the basic functionalities ('basic' needs to be decided) whenever there is any change in SW
- Any change in development or, test environment etc

8.8 RBT Categories

8.8.1 Testing Requirement with Operator '>'

Requirement:

➢ Software ***shall*** set TO1 = TRUE when [(TIN1 > 30.0)]

Test Approach:

- Assume that the data range of TIN1 is [-10.0 to 200.0]
- Note the toggling status of TO1 in terms of arranging the input

Test Type	Description	TIN1	TO1
Normal	Test for TIN1 with 1 LSB* greater than operator condition value	30.000001	TRUE
Normal	Test for TIN1 at operator condition value	30.0	FALSE
Normal	Test for TIN1 for maximum limit	200.0	TRUE
Normal	Test for TIN1 for minimum limit	-10.0	FALSE
Robustness	Test for TIN1 for more than maximum limit	200.000001	TRUE
Robustness	Test for TIN1 for less than minimum limit	-9.999999	FALSE

* Note that 1 LSB should be equal to 1 for integer data type and float precision in case of float data type

8.8.2 Testing Requirement with Operator '>='

Requirement:

- Software shall set TO1 = TRUE when [(TIN1 >= 30.0)]

Test Approach:

- Assume that the data range of TIN1 is [-10.0 to 200.0]

Test Type	Description	TIN1	TO1
Normal	Test for TIN1 with 1 LSB greater than operator condition value	30.000001	TRUE
Normal	Test for TIN1 1LSB less than operator condition value	29.999999	FALSE
Normal	Test for TIN1 at operator condition value	30.0	TRUE
Normal	Test for TIN1 for minimum limit	-10.0	FALSE
Normal	Test for TIN1 for maximum limit	200.0	TRUE
Robustness	Test for TIN1 for less than minimum limit	-9.999999	FALSE
Robustness	Test for TIN1 for more than maximum limit	200.000001	TRUE

8.8.3 Testing Requirement with Operator '<'

Requirement:

➢ Software shall set TO1 = TRUE when [(TIN1 < 30.0)]

Test Approach:

- Assume that the data range of TIN1 is [-10.0 to 200.0]
- Note the toggling status of TO1 in terms of arranging the input

Test Type	Description	TIN1	TO1
Normal	Test for TIN1 with 1 LSB less than operator condition value	29.999999	TRUE
Normal	Test for TIN1 at operator condition value	30.0	FALSE
Normal	Test for TIN1 for minimum limit	-10.0	TRUE
Normal	Test for TIN1 for maximum limit	200.0	FALSE
Robustness	Test for TIN1 for less than minimum limit	-9.999999	TRUE
Robustness	Test for TIN1 for more than maximum limit	200.000001	FALSE

8.8.4 Testing Requirement with Operator '<='

Requirement:

➢ Software shall set TO1 = TRUE when [(TIN1 <= 30.0)]

Test Approach:

- Assume that the data range of TIN1 is [-10.0 to 200.0]

Test Type	Description	TIN1	TO1
Normal	Test for TIN1 with 1 LSB less than operator condition value	29.999999	TRUE
Normal	Test for TIN1 1LSB more than operator condition value	30.000001	FALSE
Normal	Test for TIN1 at operator condition value	30.0	TRUE
Normal	Test for TIN1 for maximum limit	200.0	FALSE

Test Type	Description	TIN1	TO1
Normal	Test for TIN1 for minimum limit	-10.0	TRUE
Robustness	Test for TIN1 for more than maximum limit	200.000001	FALSE
Robustness	Test for TIN1 for less than minimum limit	-9.999999	TRUE

8.8.5 Testing MC/DC Requirement

Note that this requirement is irrespective of Level of software and subjected to agreement in software plan.

Requirement:

- Software shall set TM1 = TRUE when [(TX1 = TRUE) AND (TY1 > 30.0)]

Test Approach:

- Ensure that TM1 = FALSE as initial condition
- When checking the output, TM1, ensure that o/p values are toggled in different test case
- Since TY1 is a float i/p, values should be checked for greater by 1 LSB (or possible constraint due to HW).

Test Type	Description	(TX1 = TRUE)	(TY1 > 30.0)	TM1
Normal	Verifies the independence of TY1 on the output	TX1 = **TRUE**	**FALSE**, TY1 = 30.0	**FALSE**
Normal	Verifies that the output signal values are produced given the input values	TX1 = **TRUE**	**TRUE**, TY1 = 30.0 + 1 LSB	**TRUE**
Normal	Verifies the independence of TX1 on the output	TX1 = **FALSE**	**TRUE**, TY1 = 30.0 + 1 LSB	**FALSE**

8.8.6 Testing Timer Requirement

Requirement:

- Software shall set TY1 = TRUE when [TX1 = TRUE] for 300 ms.

Test Approach:

- Check for output with the shortest timer/counter pulse for inactive state
- Check for output with the timer/counters:
  - Make TX1 = TRUE for 'half-time duration' i.e, 150 ms, then make TX1 = FALSE for 150 duration
  - Timer is active for about half the timer/counter value,
  - Inactive for the duration of the timer/counter value and then
  - Active for the duration of the timer/counter value
  - Active for the duration + some tolerance (eg. 10 ms), of the timer/counter value

This is ensure that timer reset is properly taken care in the software before testing for full duration

Test Type	Description	TX1	TY1
Normal	Test the output with timer for shortest pulse for inactive state	FALSE for 1 ms	FALSE
Normal	Timer is active for about half the timer/counter value, inactive for the duration of the timer/counter value and then active for the duration of the timer/counter value, and then active for the duration of timer/counter + additional count	TRUE for 150 ms FALSE for 300 ms TRUE for 300 ms TRUE for 310 ms	FALSE FALSE TRUE TRUE

8.8.7 Testing Latching Requirement

Requirement:

- Software shall set 'TX Failed' = TRUE when 'RX Invalid' = TRUE for 100 ms.
- Software shall latch 'TX Failed' in RAM.

Test Approach:

- First part of requirement should be verified as per 'Testing Timer Requirement'
- For the latching of data in RAM, it should be verified as:
  - Test that latched data is set when the input condition is set for setting the latch
  - Test that latched data is set even when the input condition is not-set, for duration of timer/counter
  - Power Cycle the target board (or reset on other simulated environment)
  - Check that latch data is reset

Test Type	Description	RX Invalid	TX Invalid
Normal	Test the output with timer for shortest pulse for inactive state	FALSE for 1 ms	FALSE
Normal	Timer is active for about half the timer/counter value, inactive for the duration of the timer/counter value and then active for the duration of the timer/counter value, and then active for the duration of timer/counter + additional count	TRUE for 50 ms FALSE for 100 ms TRUE for 100 ms TRUE for 110 ms	FALSE FALSE TRUE TRUE
Normal	Check that 'TX Failed' is latched as TRUE	FALSE for 100 ms FALSE for 110 ms	TRUE TRUE
Normal	Verifies that latched signal 'TX Failed' is cleared upon power cycle	FALSE	FALSE

8.8.8 Testing Interpolation Requirement

Requirement:

- Software shall calculate the velocity schedule from the Airspeed using the linear interpolation as shown below:

Airspeed (knots)	Velocity Schedule (mm/sec)
<0	5.451
0	9.681
160	9.681
400	2.766
>400	0.545

Test Approach:

- Check that output in terms of float can be checked as part of any message could be checked, else can check this as part of interpolation module tests
- For every point on Airspeed, check for +/-1 LSB of input (Airspeed)
- Assumed LSB = 1 for Airspeed, needs to seen as per requirement
- Assumed that Airspeed range as per the data dictionary (or ICD) range = 0 to 600 knots

Test Type	Description	Input - Airspeed(knots)	Output - Velocity Schedule (mm/sec)
Robustness	Verifies the Velocity Schedule with Airspeed <0 (MIN Limit - 1)	-1	5.431
Normal	Verifies the Velocity Schedule with Airspeed = 0	0	9.681
Normal	Verifies the Velocity Schedule with Airspeed = 159	159	9.681
Normal	Verifies the Velocity Schedule with Airspeed = 160	160	9.681
Normal	Verifies the Velocity Schedule with Airspeed = 399	399	9.681
Normal	Verifies the Velocity Schedule with Airspeed = 400	400	9.681
Normal	Verifies the Velocity Schedule with Airspeed = 401	401	0.545
Normal	Verifies the Velocity Schedule with Airspeed = Max limit	600	0.545
Robustness	Verifies the Velocity Schedule with Airspeed = 601, Max limit + 1	601	0.545

8.8.9 Testing Hardware Interface

8.8.9.1 Testing Analog Requirement

Requirement:

- Software shall interface 28V DC power input via 12 bit ADC sampled at 1 ms.

Test Approach:

- Assume that DC power range = 0 to 28V, hence:
  - 0V = 0
  - 28V = 0xFFF on 12 bit ADC
  - 1 LSB = 28/0xFFF V
- Assume that SW scheduler runs @ 1ms

Test Type	Description	Voltage Input at source	Expected voltage
Normal	Test for minimum voltage at input	0V for 1ms	0V +/- 1LSB

Test Type	Description	Voltage Input at source	Expected voltage
Normal	Test for intermediate voltage at input	14V for 1ms	14V +/- 1LSB x 14
Normal	Test for maximum voltage at input	28V for 1ms	28V +/- 1LSB x 28

8.8.9.2 Testing NVRAM Requirement

Requirement:

- When commanded, software shall set store the LVDT-1 offset in NVRAM if it is within the range 0-10 mm.
- Software shall send a message on ARINC Label 110 with:
- Bit 10 => Status offset validity (data range between 0-10 mm) and,
- Software shall send a message on ARINC Label 120 with the value of LVDT-1 offset, if data is written properly in NVRAM
- Unless commanded, software shall set use the LVDT-1 offset from NVRAM after power-up

Test Approach:

Note: Focus of the section is to make the test case related to NVRAM being written – [A], verified from an output regarding the data write status – [B], and retrieved properly next time after power-up – [C].

Typical issues of an NVRAM implementation:

- If for any reason, SW does not able to write, it times-out without indicating properly
- Usually NVRAM implementation are done on walk-through, i.e, every time the new data is getting written in NVRAM, previous location is erased, and then data gets written to next available location.
- Robustness -> On 100-200 power-up, data missed to be retrieved. This typically reveals the protocol issue on NVRAM read at power up.
- Robustness ->The number of times the data should be forced to written should be based on forcing the data to be written 'at least once' throughout the NVRAM to prove wrap-around design for choosing new location is properly implemented

Test Type	Description	Input - LVDT-1 Offset	Expected Output
Normal	Verifies the LVDT-1 offset out of-range write operation of NVRAM	0mm	ARINC Label 110, Bit 10 = 0 Label 120 data = 0
Normal	Power up the board	NA	ARINC Label 110, Bit 10 = 0

Test Type	Description	Input - LVDT-1 Offset	Expected Output
			Label 120 data = 0
Normal	Verifies the LVDT-1 offset in-range write operation of NVRAM	1mm	ARINC Label 110, Bit 10 = 1 Label 120 data = 1
Normal	Power up the board	NA	ARINC Label 110, Bit 10 = 1 Label 120 data = 1
	Repeat the above two step for 2 – 9 and out-of-range case for 10		
Robustness	Tests 200 power up to verify the NVRAM data retrieval		
Robustness	Based on the NVRAM design architecture repeat the in-range case of NVRAM write to cover the complete range of sector being written and wrap-backed.		

8.8.9.3 Testing ARINC 429 Requirement (Typical)

Requirement:

- Software shall receive the ARINC Label 314, XY Position as defined below.

Message Name	Label (Bits 1-8)	Direction	Speed (kbps)	Update interval (ms)	Type	Eng Unit	Eng Unit Min	Eng Unit Max
XY Position	314	Input to the Board from ARINC Source	100	20	BNR	Deg	-15	+15

BIT	MEANING	BIT	MEANING	BIT	MEANING	BIT	MEANING
1-8	Label	15	Not used	22	0.1171875	29	0 = "+"
9	SDI	16	Not used	23	0.234375		1 = "-"
10		17	0.003662109	24	0.46875	30	SSM
11	Not used	18	0.007324219	25	0.9375	31	
12	Not used	19	0.014648438	26	1.875	32	Parity
13	Not used	20	0.029296875	27	3.75		
14	Not used	21	0.05859375	28	7.5		

Test Approach:

- This requirement, related to communication, should be tested for:
    - Data validity in terms of valid SSM, P, SDI
    - Data value, normal and robustness
    - Data receive rate

Test Type	Description	Input	Expected output
Normal	Test for label 314, with normal range data, valid SSM, valid SDI, **invalid** Parity	Input ARINC Word = 0xTBD	Data Invalid
Normal	Test for label 314, with normal range data, valid SSM, **valid** SDI, valid Parity	Input ARINC Word = 0xTBD	Data valid
Normal	Test for label 314, with normal range data, valid SSM, **invalid** SDI, valid Parity	Input ARINC Word = 0xTBD	Data Invalid
Normal	Test for label 314, with normal range data, valid SSM, valid SDI, valid Parity	Input ARINC Word = 0xTBD	Data valid
Normal	Test for label 314, with normal range data, **invalid** SSM, valid SDI, valid Parity	Input ARINC Word = 0xTBD	Data invalid
Normal	Test for label 314, with normal range data, **valid** SSM, valid SDI, valid Parity	Input ARINC Word = 0xTBD	Data valid
Normal	Test for label 314, with **min range data**, valid SSM, valid SDI, valid Parity	Input ARINC Word = 0xTBD	Data valid
Normal	Test for label 314, **with max ICD range data**, valid SSM, valid SDI, valid Parity	Input ARINC Word = 0xTBD	Data valid
Robustness	Test for label 314, **with max data range data**, valid SSM, valid SDI, valid Parity	Input ARINC Word = 0xTBD	Data valid
Robustness	Test for label 314, with max data range data, valid SSM, valid SDI, valid Parity, and **unused bit set (Bit 11-16)**	Input ARINC Word = 0xTBD	Data valid, no change
Normal	Test for the receive rate at 20 ms		

8.8.9.4 Testing Data Bus Communication Requirement (Typical)

Requirement:

➢ Software shall transmit the following ARINC Label on the Avionics bus.

- Label L1, Rate 100ms, Message Format: BNR, Data Format: ___
- Label L2, Rate 100ms, Format: Discrete, Data Format: ___
- Label L3, Rate 100ms, Format: BNR, Data Format: ___
- Label L4, Rate 100ms, Format: BNR, Data Format: ___

Test Approach:

- Prior to coming to this requirement, it is assumed that all the labels have been tested for Message Format, Rate, and Data Format for normal/robustness cases. The approach here is testing for completeness.
- Test that all the messages listed in the requirement are transmitted
- Test that no other message apart from the one listed is transmitted

8.8.9.5 Testing Discrete Interface Requirement

Requirement:

➢ Software shall interface the Pin programming discrete from the input Pin 1

Test Approach:

- Make the Pin 1 high (either via discrete simulator or by giving the required voltage at the pin)
- Check that Software receives the discrete status as high
- Make the Pin 1 low
- Check that Software receives the discrete status as low

8.8.9.6 Testing Watchdog Requirement

Requirement:

➢ Software shall configure the external watchdog for 5 ms timeout

Test Approach:

- Prove that SW has configured the watchdog for 5 ms, and DSP remains alive when configured
- Induce scenario that SW is unable to serve the watchdog, thus expecting the DSP reset within 5 ms

8.8.9.7 Testing Stack Requirement

Requirement:

➢ Software shall monitor the stack boundary when 70% of the stack limit is utilized by raising 'STACK FAIL' bit at specified memory location

Test Approach:

- Important part of test is to verify that SW can handle the failure condition when stack overflows in a predictable manner. Usually this is done as part of worst case timing testing.
- Assume (for approach purpose) that stack memory ranges 0x200 to 0x2000, hence 70% = 0x1700
- Assume (for approach purpose) 16 bit memory access

Test Type	Description	Input	Expected output
NA	Setup the stack memory with a known pattern	Using emulator, write 0x0 from 0x1700 to 0x20000, reset the CPU and run the SW previously loaded in Flash	Check that 0x1700 to 0x2000 is written by 0xAAAA Check that 'STACK FAIL' bit = FALSE
Normal	Verify the stack memory utilization	Perform the test scenario that causes the maximum stack utilization (maximum nesting as per call tree) If 0x1700 != 0xAAAA, Manually set 0x1700 with 0xAAAA	Memory location from 0x1700 to 0x2000 is 0xAAAA. If 0x1700 != 0xAAAA, check that 'STACK FAIL' bit is TRUE Check that 'STACK FAIL' bit is TRUE

8.8.9.8 Testing CRC Requirement

Requirement:

- A CRC check shall be performed on the contents of the program ROM during CBIT.
- If the test detects a bad CRC, software shall stop further execution after attempting to report the failure over the ARINC bus, Label 310, Bit 2.

Test Approach:

- Check that no CRC error is reported on the ARINC Label 310, Bit 2
- Corrupt the CRC either by:
  - If the CRC is stored in RAM, halt the program, and corrupt the CRC using emulator
  - If there is no access to alter the CRC, rebuild the program with wrong CRC, power up
- Check that CRC error is reported on the ARINC Label 310, Bit 2
- Check any expected 'refresh or computed' data is no more refreshed

8.8.9.9 Testing Timing Margin Requirement

Requirement:

- The software shall have 30% spare timing margin at the time of certification

Test Approach:

Note: Timer Utilization = Obtained Timing / Expected Timing where,

Expected Timing = Rate at which the main scheduler operates and,

Obtained Timing = Worst case time obtained upon driving the system to maximum utilization based on inputs/scenarios

The guideline for maximum utilization for worst case timing should be:

- To the extent possible, SW should acquire all the inputs
- To the extent possible, SW should produce the output
- Call-tree should be analyzed to be invoked to maximum. However it should be noted that call tree can give the deepest nesting path on synchronous event, but asynchronous event should also be considered for worst timing analysis. This needs to be assessed based on the knowledge of software architecture apart from high level requirement.
- The timing should be usually measured by the clock different than the one used for implementing the main scheduler should be used. Example – timing measured via scope.
- Generally preferred method is to monitor an output line (or output signal) that will be OFF in the start of scheduler and ON in the end of schedule. Measured time should be OFF-OFF period in a scope.

Test Type	Description	Input	Expected output
Normal	<Worst case timing scenario>	<inputs>	Spare Timer margin >= 30% of <expected rate>

8.8.9.10 Testing Power On Built-In-Test Requirements

Requirement:

- The software shall perform the ARINC Transreceiver test during Power-up test
- The software shall transition to 'Failed mode' if the ARINC test fails
- The software shall transition to 'Normal mode' if the ARINC test passes

Test Approach:

Note: Usually all the power-up test are via software-controlled, example making the routing of RX->Tx->Rx based on control register, and hence test case may not have provision to have any control while the DSP/controller power's up.

Hence possible options to test these requirements are:

- On any output communication message from SW
- Storage of ARINC status on the memory (like RAM, NVRAM etc) else,
- Go for debug based testing if allowed on the software using debugger Or,
- Modify the code to have the status of test reflected in communication output message or, memory storage. The modified code should be justified in terms of documentation and need or,
- Check any output that is outcome only when the SW is in normal mode under 'Pass' state of ARINC Test, and similarly check any output that is outcome only when the SW is in failed mode under 'Fail' state of ARINC Test,

For any approach of the test being selected, test case should be written for 'Pass' and 'Fail' outcome of ARINC Test.

Test Type	Description	Input	Expected output
Normal	Initial SW state = any state other than 'normal mode' Test for expected failure of test	Power up, inject error on the ARINC Test (using HW, SW, or by modification)	SW state = 'Failed mode' ARINC Test Status = Fail
Normal	Initial SW state = any state other than 'normal mode' Test for expected pass of test	Power up, make the state of HW test to make the ARINC Test (using HW, SW, or by modification) to Pass	SW state = 'Normal mode' ARINC Test Status = Pass

8.8.9.11 Testing Software Partitioning Requirement

Requirement:

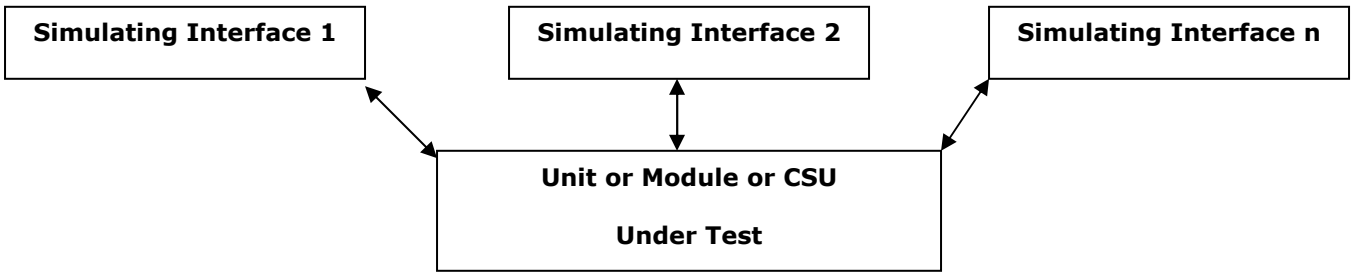
- The software shall reside in Partition A.
- The software shall interact with Partition B using communication bus/protocol xxx.
- The software shall receive the following input from Partition B ….
- The software shall transmit the following output to the Partition B ….

Test Approach:

- Review of the executable load procedure on the Partition A
- Test Case should be executed on the HSIT environment
- Test Partition to replicate the Partition B should be made to provide the data/communication interface to Partition A

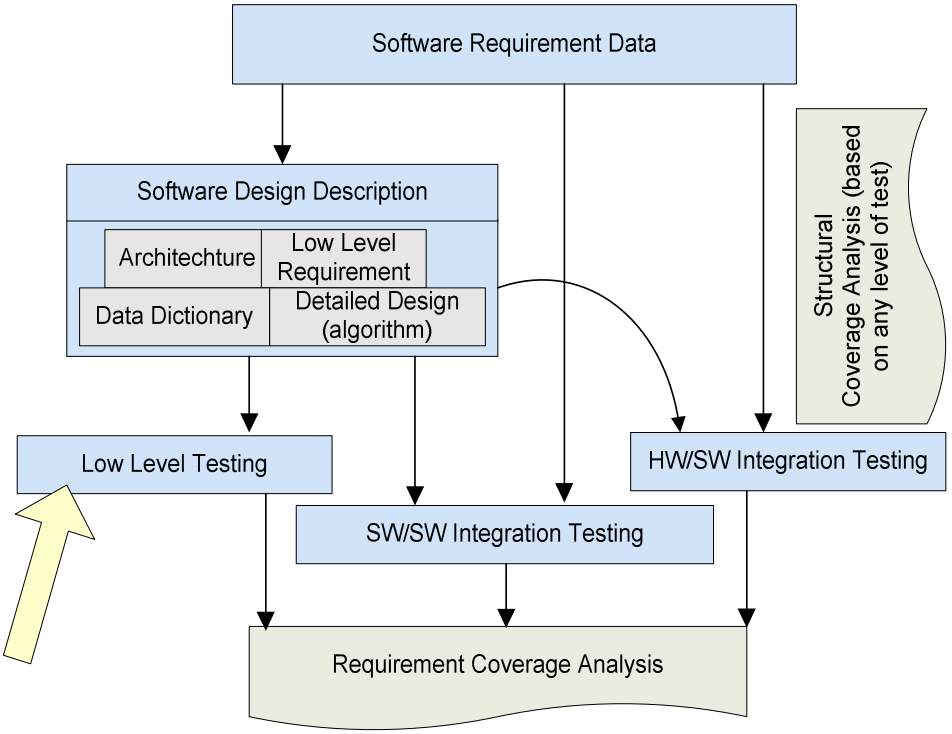
9.0 UNIT TESTING

Unit testing is the process of testing the functionality of the lowest entity with respect to the design, isolating entity from the system by means of *simulating all other external interfaces* of the entity. It is also called as white box or glass box or component or module testing. The unit under test has to undergo structural coverage other than functionality.



Any software unit or module or component must have a software design specification in order to be tested. Given any initial state of the component, in a defined environment, for any fully-defined sequence of inputs and any observed outcome, it shall be possible to establish whether or not the unit or module or component conforms to the specification.

Unit testing is usually conducted as part of a combined code and unit test phase of the software lifecycle, although it is not uncommon for coding and unit testing to be conducted as two distinct phases. The basic units of design and code are individual subprograms (procedures, functions, member functions).



9.1 ETVX Criteria for Unit Testing

Entry criteria:

Batch of source and header files to be tested.

Inputs:

- Source files,
- Header files,
- Software Requirements Data,
- Software Design Document,
- Software Coding Standard,
- Template of Code Review checklist

Activity:

The Project Manager assigns the modules to Team Members. The following activities are performed by the Team Member:

- The source code is checked for conformance with the requirements through the traceability matrix.
- The source code is checked for conformance with the coding standards.

A Problem Report is raised for any discrepancies mentioned in the Code Review checklist and sent to the customer. The Unit Testing phase commences after successful completion of Code Review phase with no discrepancies mentioned in the code review checklist.

Outputs:

Software Verification Cases and Procedures, Code Review checklist and Problem report (if any).

9.2 Unit Testing Objectives

The principal objectives of Unit Testing are to verify the functionalities of the module under test,

- To verify compliance of each component with respect to its low -level requirements
- To verify the response of each component to normal as well as abnormal conditions
- To generate the measure of structural coverage as per DO-178B applicable level. It should be noted that the structural coverage can be attained at any level of test and not necessarily by low level test.

But in addition to the above we can have the following too.

- To verify its structural coverage,
- To check code for functional match with the design and data transformation,
- To find undiscovered errors,
- To ensure the quality of software.
- Testing the lowest level entity (Most independent i.e. CSU) within its boundaries.
- The functionality of the module and at the same time tests its robustness.

As per DO-178B Section 6.4, If a test case and its corresponding test procedure are developed and executed for hardware/software integration tests or software integration testing and satisfy the requirements-based coverage and structural coverage, it is ***not necessary to duplicate the test for low-level testing***. Substituting nominally equivalent low-level tests for high-level tests may be less effective due to the reduced amount of overall functionality tested. Hence the applicability of performing low level test should be documented in the SVP.

Unit test is simplified when a module with high cohesion is designed. When a module addresses only one function, the number of test cases is reduced and errors can be more easily predicted and uncovered. Unit Testing is also called white box testing. White box is a testing technique that takes into account the internal structure of the system or the component. The entire source code of the system must be available. This technique is known a white-box testing because the complete internal structure and working of the code is available.

Unit helps to derive test cases to ensure the following

- All independent paths are exercised at least once.
- All logical decisions are exercised for both true and false paths.
- All loops are executed at their boundaries and within operational bounds.
- All internal data structures are exercised to ensure validity.

9.3 INPUTS AND OUTPUTS FOR UT

INPUTS	OUTPUTS
FOR TEST CASE GENERATION: ➢ Software Design Description (SDD) ➢ Low Level Requirement (this is usually part of SDD document) ➢ Data Dictionary ➢ Testing Standards, Software Verification Plan ➢ Any tool used for the traceability management (Test Case ⇔ LLR) example DOORS as applicable **FOR TEST PROCEDURE GENERATION:** ➢ Test Case ➢ Tool and its supporting environment for writing the test script if any (Example, Rational Test Real Time with Target Deployment Port on specific target) ➢ Testing Standards, Software Verification Plan	➢ Software Verification Cases and Procedure Document ➢ Test Case ➢ Test Procedure ➢ Traceability Matrix, LLR ⇔ Test Case, Test Case ⇔ Test Procedure ➢ Test Report ➢ Test Report (Pass/Fail status) ➢ Coverage Report ➢ Problem Report if applicable ➢ Review Record/Checklist ➢ Configuration records ➢ Additionally the following data should be completed prior starting the formal execution of low level test if applicable: ➢ Source to object verification – required only for Level A and if structural coverage is obtained based on source code (and not on object code), Refer Appendix C for detail on the guideline for performing this activity ➢ Tool Qualification data if applicable

9.4 Unit Testing Guidelines

9.4.1 Common Testing Guideline

- The initial condition of the output should be of different than the expected result in the test case.
- When checking the output, ensure that output values are toggled in different test case to prove affect of input.
- For the expected value of the float on the target computer, delta value for tolerance should be as stated in SVP or standard or, design document as applicable
- It is acceptable to modify the software; however this should be documented and made part of test procedure or, SVCP document. Typical example of code modification for Low Level Test
    1. Infinite loops
    2. Tool related constraints related modification

       Example: Usage of interrupt for the function prototype. Since the testing is done on low level, ISR will not be present. Hence modification can be done and justification to be recorded in SVCP
    3. Usage of asm keyword
- For the test covering the till the boundary value of the input should be categorized as "Normal Range" test. For the test covering the outside the boundary value of the data dictionary of the input, should be categorized as "Robustness Range" test.
- Test Case description should state the intention or objective or highlight of test instead of repeating the input and expected data

Example:

- ✓ Tests the nominal range of input 1
- ✓ Tests the boundary value for input 1
- ✓ Tests the out of range for input 1
- ✓ Tests the FT condition for MC/DC combination on Input 1, Input 2
- ✓ Tests the divide by zero condition for input 1 / input 2
- ✓ Test for Input 1 with 1 LSB (1 count) greater than operator condition value
- ✓ Test for Input 1 at operator condition value
- ✓ Test for Input 1 for maximum limit
- ✓ Test for Input 1 for minimum limit
- ✓ Test for Input 1 for more than maximum limit
- ✓ Test for Input 1 for less than minimum limit
- ✓ Tests the switch case X, value Y
- ✓ Tests the default case switch case X

- For Low Level requirement based testing, data type should be based on design and not to the source code.
- Test Case and Procedure should follow the accepted template

9.4.2 Test Case Guideline

- Function under test should be public function of .c (or .ada etc as applicable)
- The variables to be tested should be:
    - ✓ Function parameter(s)
    - ✓ Function return variable (if applicable)
    - ✓ Global variable(s)
    - ✓ Memory mapped local variables or, registers
    - ✓ Stub parameter(s)
    - ✓ Number of times of calling the Stub
    - ✓ Stub return variable (if applicable)

  Note: The stubs are the functions that are external to the current module.
- All output data(s),
    - ✓ Parameter(s) – Function or Stubs
    - ✓ Global variables
    - ✓ Memory mapped local variables

  should have an expected value different from initial values
- The individual test case should test only the variables or fields of structure or specific array index updated in the design

  Note: In general, entire variable (structure, array, parameter etc) should be kept in environment block (or Initialization block) with known input and expected output, and specific field of structure or, array should be tested in specific test case. This will be applicable to function parameters, local variables, and global variables. This block will be default executed all the cases with the same input, expected value unless the specific case overrides the input and expected value.
- The mode (IN, OUT or INOUT) used in stub should be same as in the actual function prototype
- 'Named values' of enumerated types, preprocessor definitions should be used instead of its numerical values as specified in data dictionary
- The expected value should not computation, and should state the explicit value
- The tolerance used for expected floating data should be as agreed in the plans
- Test case should cover the nominal value of the input data as per data dictionary range
- Test case should cover the boundary value of the input data as per data dictionary range
- Test case should cover the out of boundary range of the input data as per data dictionary range

Relational operators should be tested on the relational boundary +/- LSB as count of 1. Apart from these values, minimum, maximum and out of the boundary value and any other point of interest should be tested.

- ✓ "a>10": Test, a = 10, a = 11
- ✓ "a>=10": Test, a = 9, a = 10, a = 11
- ✓ "a<10": Test, a = 9, a = 10
- ✓ "a<=10": Test, a = 9, a = 10, a = 11
- ✓ "a != 20", Test a = 20 and any other value
- ✓ "a == 20", Test a = 20 and any other value

- Static functions should be tested through the calls of the public function only
- For floating point the test cases should be same as integer except the LSB should be changed to "Delta" stated in plan
- Interrupt Service Routine, ISR, functions will be tested as normal function (without ISR invocated) at low level test
- Use a variety of inputs and expected results rather than constantly testing with the same sets of values
- The traceability of low level requirement to test case(s) should be complete i.e, at least one test case should exists for low-level requirement
- The traceability of test procedure(s) to test procedure should be complete i.e, every test case should have the associated test procedure.

9.4.3 Test Procedure (or Script) Guideline

- The test procedure should be traceable to the test case(s)
- No test procedure should be written without the traceability to the test case(s)
- For uniform indentation, common editor should be used, with a defined tab spaces, and no tab usage

9.4.4 Test Report Guideline

Test execution should yield Pass/Fail status and structural coverage report if applicable.

- The tests should be executed with and without instrumentation (only for instrumentation based tool). Only if the test result has the pass status, the coverage credit can be taken. In case of the difference of the test result, problem report should be raised and analyzed
- For any failure of the test, problem report should be raised
- Problem Report should be raised for any "Failed" test status or insufficient coverage

9.4.5 Typical Issues/Errors Found In Design during Low Level Test

Typical Issues/Errors, not an exhaustive list, found during Low Level Test against design are stated below.

- Source Code does not implement the algorithm
- Dead code
    - ✓ Due to incorrect coding practices
    - ✓ Additional code without being associated to requirement
- Data overflow/underflow on assignments due to type-casting problems: eg. output (signed 16) = input 1 (signed 32)
- Un-initialized data leading to inconsistent expected value
- Mathematical computation issues
    - ✓ Divide by zero
    - ✓ Data overflow/underflow on computation, example output (signed 16) = input 1 (signed 16) * 2
    - ✓ Accuracy issues on floating computation expected values
- Incorrect data access modes (IN, OUT, INOUT). Example global data defined as IN, is actually INOUT etc
- Memory access violation, example indexing an out-of bound array
- Incorrect logic decision, example brackets not proper, usage of ">" instead of ">=", usage of "=" instead of "==" etc
- Incorrect response to corrupted input
- Incorrect handling of validity status of data prior to usage. Example, a global data is assigned to any physical memory address by an call to the external function, and the same global data is referenced in the current function under test
- Design lists the components that are never called for.
- Lack of information in design. This should ideally be mitigated by performing the design and code reviews on the baseline prior performing the low level test
    - ✓ Does not contain the detail design for all component
    - ✓ Does not contain the data dictionary details, data/range/type/defines etc

9.4.6 Typical Issues/Errors Found within Low Level Test Itself

Typical Issues/Errors, not an exhaustive list, found within Low Level Test are stated below. The issues are listed in terms of volume of the issues in ascending order, based on project case studies. Most of the issues are related to configuration, followed by test case selection criteria followed by reviews.

- Test done on component in distributed baseline of components. Problems surfaces during the formal run
- Tool related files required for re-creating the environment (example, .rtp, .inf in case of RTRT) not checked in or, not the latest
- Incorrect description of test case objectives, test repeating the test data rather than the rationale
- Due to lack of any automated traceability management tool, lack or, incorrect, bi-directional trace between LLR ⇔ Test Case and Test Case ⇔ Test Procedure (or Script)
- Test does not verify all the data elements, either due to lack of initialization setup of complete data structure or, missing to test each data elements individually
- Test calls the function under test with value instead of variables
- Function/Stub Parameter mode not tested as per design
- Modification on the source code not documented, and found to have issue on test execution
- Incorrect handling of memory mapped register – not repeatable if not set properly
- Inability of tester to set the linker memory section leading to unpredicted result
- Issue with the setting of coverage setting, this getting debated on late state of project

9.5 Unit Test Case Designing

Unit testing should be done only with the Software Design Document as the input but not the code

Because

WITH CODE AS INPUT, THE REQUIREMENTS ARE NOT TESTED AND

MISMATCH BETWEEN DETAILED DESIGN AND CODE CANNOT BE DETECTED

Unit testing focuses verification efforts on the smallest unit of software design the module. Using the procedural design description as guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of the tests and uncovered errors are limited by the constraint scope established for the unit testing. The unit test is normally white-box oriented and the step can be conducted in parallel for the multiple modules.

Example 1 : With Detailed Design as Input

Detailed Design says:

Swap the values of 2 integer variables A and B (where A, B holds half the magnitude of its type).

A = A + B **B = A - B** **A = A - B**	**A = A - B** **B = B + A** **A = B - A**	**Temp = A** **A = B** **B = Temp**

Irrespective of the kinds of algorithm used, the function it does is swapping two values of variables. One of the inputs and expected output of this requirement is

I/p **-> A=10, B=5** and O/p -> **A=5, B=10** (Testing Within Boundaries)

Then we can say that the functionality is correctly implemented.

Example 1: With Code as Input

Detailed Design says:

Swap the values of 2 integer variables A and B (where A, B holds half the magnitude of its type).

A = A * B **B = A / B** **A = A / B**	**A = A / B** **B = A * B** **A = B / A**	**Temp = A** **B = A** **A = Temp**

Consider the swap design using "Temp". Since the design is not referred, the tester doesn't have the knowledge that this is a code for swapping the values of 2 variables. So tester executes his test with

I/p -> *A=10, B=5* and expects O/p -> *A=10, B=10* Result is OK

From tester's point of view, the code is behaving correctly and there is absolutely no error with the functionality. Do you really prove any point in this kind of testing…?

NO is the only answer

Now let us consider the below design

A = A * B **B = A / B** **A = A / B**	**A = A / B** **B = A * B** **A = B / A**	**Temp = A** **B = A** **A = Temp**

Consider code in first 2 designs without using "Temp". Given I/p -> A=10, B=5 and O/p is A=5, B=10… great!!! It works, isn't it…? So the functionality is correct, you would say…

Now if we give the following *I/p -> A=10, b-> 0 O/p -> ??* What went wrong? Analyze the code. Step 2 resulted in divide by zero error. So the code executed is *not ROBUST* -> functionality is not implemented for all the possible values of inputs

Unit Testing is normally considered as an adjunct to the coding step. After source-level code has been developed, reviewed and verified for correct syntax unit test case design begins. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed above. Each test case should be coupled with a set of expected results. Since the module is not a standalone program, driver and or the stub must be developed for each unit test. The driver is nothing more than a "main program" that accepts test cases data, passes such data to the test module and prints relevant results. Stubs serve to replace the modules that are subordinate to the module that is to be tested. A "STUB" or "DUMMY SUB PROGRAM" uses the subordinate module's interface may do minimal data manipulation and reports verification of entry and returns.

9.5.1 Objective

The objectives of Low level testing are:

- To verify compliance of each component with respect to its low -level requirements
- To verify the response of each component to normal as well as abnormal conditions
- To generate the measure of structural coverage as per DO-178B applicable level. It should be noted that the structural coverage can be attained at any level of test and not necessarily by low level test.

As per DO-178B Section 6.4, If a test case and its corresponding test procedure are developed and executed for hardware/software integration tests or software integration testing and satisfy the requirements-based coverage and structural coverage, it is not necessary to duplicate the test for low-level testing. Substituting nominally equivalent low-level tests for high-level tests may be less effective due to the reduced amount of overall functionality tested.

Hence the applicability of performing low level test should be documented in the SVP.

9.5.2 Low Level Test Inputs / Outputs

INPUTS	OUTPUTS
FOR TEST CASE GENERATION: ➢ Software Design Description (SDD) ➢ Low Level Requirement (this is usually part of SDD document) ➢ Data Dictionary ➢ Testing Standards, Software Verification Plan ➢ Any tool used for the traceability management (Test Case ⇔ LLR) example DOORS as applicable **FOR TEST PROCEDURE GENERATION:** ➢ Test Case ➢ Tool and its supporting environment for writing the test script if any (Example, Rational Test Real Time with Target Deployment Port on specific target) ➢ Testing Standards, Software Verification Plan.	➢ Software Verification Cases and Procedure Document ➢ Test Case ➢ Test Procedure ➢ Traceability Matrix, LLR ⇔ Test Case, Test Case ⇔ Test Procedure ➢ Test Report ➢ Test Report (Pass/Fail status) ➢ Coverage Report ➢ Problem Report if applicable ➢ Review Record/Checklist ➢ Configuration records Additionally the following data should be completed prior starting the formal execution of low level test if applicable: ➢ Source to object verification – required only for Level A and if structural coverage is obtained based on source code (and not on object code), Refer Appendix C for detail on the guideline for performing this activity ➢ Tool Qualification data if applicable

9.5.3 Test Case/Procedure Format

As per DO-178B, a test case is defined as set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

The test case template should contain:

TC Identifier	Test Type	Description	Input	Output	Initial Condition(s)	Expected Result(s)	Pass/Fail Criteria	Requirements Traced
Case 1	Normal	<To mention the high level description / objective of test without repeating the input/output values>	Input 1 Input 2	Output 1 Output 2		See Output Signals	actual results equal expected result	Req-1
Case 2	Normal	Example, Tests the timer reset condition Tests for FT condition for MC/DC combination on Input 1, Input 2				See Output Signals	actual results equal expected result	Req-1
Case 3	Normal					See Output Signals	actual results equal expected result	Req-2
Case 4	Robustness					See Output Signals	actual results equal expected result	Req-3

As a good practice, it is advisable to trace only one requirement per test case row for easing the review process of requirement coverage to test case. In case if the same test case (example Case 1) traces to more than one requirement, test case can be added as a new row with reference to Case 1 in test case description. This may look redundant, but found to be very effective while reviewing the completeness of a requirement. As per DO-178B, a test procedure is defined as detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.

The procedure should also include the procedure to retrieve the test script from the CM. The source code, executable retrieval procedure can be referenced to Build and Load Procedure document, typically maintained in Software Environment Life Cycle Configuration Index document. In case the software builds needs to be made specifically for testing, the specific build procedure should be detailed.

The test setup in terms of environment setup is usually documented in SVCP, Software Cases and Procedure document based on test environment. The test procedure itself is usually implemented as script, based on the test case. It is acceptable to have the test procedure implementing multiple test cases. The traceability of test procedure to test case and vice-versa should be maintained. The test procedure template should contain:

- Test Procedure Name (or Identifier)
- Test Procedure Author/Version/Date: (Note the Author/Version/Date can be maintained by CM tool usage and need not be part of template itself)

- Test Environment: This can be part of each script or, as a common note in SVCP document
- Test Case(s) Traced

For cases where the source code debug method is utilized for requirement coverage, test case should still be written. Only the format of test procedure should mention the execution steps. In certain cases, the requirements are directly verified via inspection of implementation, example code inspection. In such cases, test case, test procedure format will not be applicable.

9.5.4 Data Dictionary

Data Dictionary is defined during the design phase, which defines the detailed description of data, parameters, variables, and constants used by the system, both internally and externally throughout the software architecture. It can be part of SDD itself or separate document.

Data dictionary is a vital input for performing the low level test for performing the data range tests. A data dictionary should contain the following details at minimum:

Data Dictionary for global, constant data: (content filled for sample)

Element Name	Data Type	Description	Initial Value	Range	Unit
gAirSpeed	INT32	Stores the airspeed value	0	[0..300]	Knots

gVTable	VTStruct	Constant structure for voltage rate	{
			0, 5
			10, 15
			20, 22
			30, 25
			}
				NA – Constant	Volts

Data Dictionary for function parameters: (content filled for sample)

Element Name	Data Type	Description	Initial Value	Range	Unit	Function Name	Mode
pFaultCode	FaultCodeEnum	Fault Code	NA	[0..30]	NA	FaultCompute	Input
*pDest	INT32	Destination Pointer	NA	NA	NA	memCopy	Output
*pNVM_Stat	NVMCodeEnum	Stores the NVM write status	NA	[0..5]	NA	NVMWrite	Input/Output

Data Dictionary for macro definition:

Data dictionary should define the macros used for the design algorithm as applicable.

Type Dictionary:

Data dictionary should type details for the data type used in design. Additionally if there is any tolerance on the float comparison, the same should be defined as part of design document.

9.5.5 Test Case Selection Criteria

Test case selection criteria should include the following:

9.5.5.1 Requirement Based Test Cases

The objectives covered by requirement-based test cases are:

- To test that the algorithms satisfies the low-level requirement(s)
- To verify that the component does not execute un-specified requirement
- To verify the computation accuracy
- To verify that all interfaces input data have been exercised as per the range, and obtained value as expected
- To verify that expected values of all extern function interfaces are obtained
- To test the limit values as per the ranges of input(s), global variable(s) as defined in data dictionary.

9.5.5.2 Robustness Test Cases

Robustness test cases should be written to verify that erroneous input data does not disturb the component execution.

The objectives covered by robustness test cases are:

- To verify the response to the inputs outside the range specified in low-level requirement
- To verify the response to missing or corrupted input data
- To verify the violations to data access such as array limits if possible
- To verify the external interface data mapping
- To verify erroneous possibility of mathematical computations

Example of robustness cases:

- Divide by zero possibility
- Exponentials or powers with negative numbers
- Square root of negative number

- Data Overflow, Underflow, eg. multiply 2 big 16 bit values to make sure result is larger than 16 bits and does not cause overflow; storage overflow/underflow on a bit-field structure element
- Testing with the value not a member of a defined enumerated type
- Testing with the value of switch case value that is not handled
- Using invalid values
- Using null pointers
- For floating point values
  - Use infinity
  - Negative infinity
  - Not a number (NaN)

9.5.6 Test Environment

All the tests should be executed on the target computer. Hence the test environment build steps should be same as required to make the formal executable including build option.

Any deviation (like changes in compiler options, usage of simulator etc) in the test environment should be approved of environment equivalency with respect to the target computer.

9.5.7 Tool Qualification

Tool qualification should be assessed on two folds:

1. Whether tool requires qualification?
2. If yes, what category, development or verification, should the tool be qualified?

A tool is categorized as verification tool if the tool output is not part of airborne software and, the tool can fail to detect the error. The tool requires qualification, as verification tool, if the output of the tool is not reviewed, or it eliminates, reduces or automates any DO-178B process.

Example, Tool qualification would be required if the verification tool provides any one of the following feature:

- Automated generation of test cases or script
- Automated generation of test report – Pass/Fail status
- Automated generation of coverage measure
- Automated review of test cases/procedure per standards etc

Tool operational requirements should be written describing the DO-178B credit sought for use of the tool, tool installation and operating instructions.

Based upon the tool operational requirements, project usage of language constructs, test cases should be prepared. The test should be executed in the project specific test environment generating the test reports. The test reports should be manually reviewed.

In general, for any certification credit taken by tool, should be manually reviewed as part of tool qualification process.

9.5.8 Data and Control Coupling

Data coupling is defined as the dependence of a software component on data not exclusively under the control of that software component.

Control coupling is defined as the manner or degree by which one software component influences the execution of another software component.

Following paragraph provides guideline for data and control coupling.

9.5.8.1 Data Coupling

The data flow analysis will be done starting with the review of software requirements data, design document and code review using their respective review checklists.

The test coverage of the interface data flow at high level requirements will be attained by associating these data flow to the high level requirement based test, and producing an analysis report as part of software verification results for completeness.

The test coverage of the data flow at low level requirements will be attained by associating the data from data dictionary which includes all global data (linker map may also be used), to the low level requirement based test, and producing an analysis report as part of software verification results for completeness.

9.5.8.2 Control Coupling

The calling tree will be analyzed to identify all of the functions in the software. This will be documented as part of software design document and reviewed using design review checklist.

As a part of control coupling analysis, it will be manually reviewed that there are no unused function in the calling tree. Also, the structural coverage report generated by low level test (or as applicable to other level) will be analyzed to check that the entire calling trees have been executed.

Both data and control coupling analysis should be stated in software verification results.

9.5.9 Structure Coverage Analysis

It is reiterated that structural coverage can be attained at any level of test (SW/HW, SW/SW or Low Level Test).

The structural coverage objective per level is:

- Level C: 100% statement coverage
- Level B: Level C + 100% decision coverage
- Level A: Level B + 100% MC/DC coverage
- Every point of entry and exit in the program has been invoked at least once

Post test execution, the structural coverage measure should obtained. If any instrumentation based tool is used, the coverage credit can be taken only if the test passed in with/without instrumentation.

For the uncovered structural coverage portion, it should be analyzed if the following are required (or leads to):

- Changes to:
  - Low-level requirements
  - Low level tests
  - Software
- Justify the presence of non-coverage (List not exhaustive)
  - Defensive Code, example having a default case may be a coding standard requirement
  - Hardware constraints, example RAM test failure logic cannot be covered unless the memory is corrupted
  - Cannot be possible to cover under the input scenarios but a good coding practice. Example of a deficient code coverage for Level B, else part cannot be covered in the case below

    Example:

    array_index = array_index + 1;

    If (array_index > 100)

    {

    array_index = 100;

    }
  - Static function always a limited data range, and the calling function checks the range upon receiving the return value. In the example below, false part of if condition cannot be covered.

    Example:

    retVar = LimitAdc (input_var); /* Call to a static function which always limits the value 2047 after some scaling */

```
If (retVar <= 2047)
{
  …
}
```

- Code checking for some location that cannot be accessed with low level test, example a Flash location being accessed when the test is done from RAM.

- Supplement the coverage by manual inspection

Suggested format for structural coverage analysis is:

File Name / Version	Function Name	Coverage Report Name / Version	Code Extract with deficient coverage (in red)	Code Coverage Analysis	Action / Conclusion
1.c	Fun()	1.c.html	IN_AIR: Status = True; break; default: status = False; break;	Defensive Code as required by coding standard	No action required on requirement, code or test. 100% coverage is justified based on analysis

9.5.10 Requirement Coverage Analysis

The first objective of requirement coverage analysis is to verify:

- Each software requirement has one or more test cases verifying the requirement and,
- There is no test case that is not associated with any requirement

The second objective of requirement coverage analysis is to verify each requirement has test cases for normal range and equivalence class values, boundary values, and robustness tests. The review of the test cases and the traceability matrix accomplishes the requirement for test coverage analysis.

9.5.11 Formal Test Execution

The formal execution should be performed under the conformed environment and controlled inputs. Usually the environment conformity is done by Quality Assurance.

Following guideline are listed below to be followed for the assessing the readiness of formal execution. Usually the project calls for the test readiness milestone as the gate (or, transition criteria) prior conducting the formal execution.

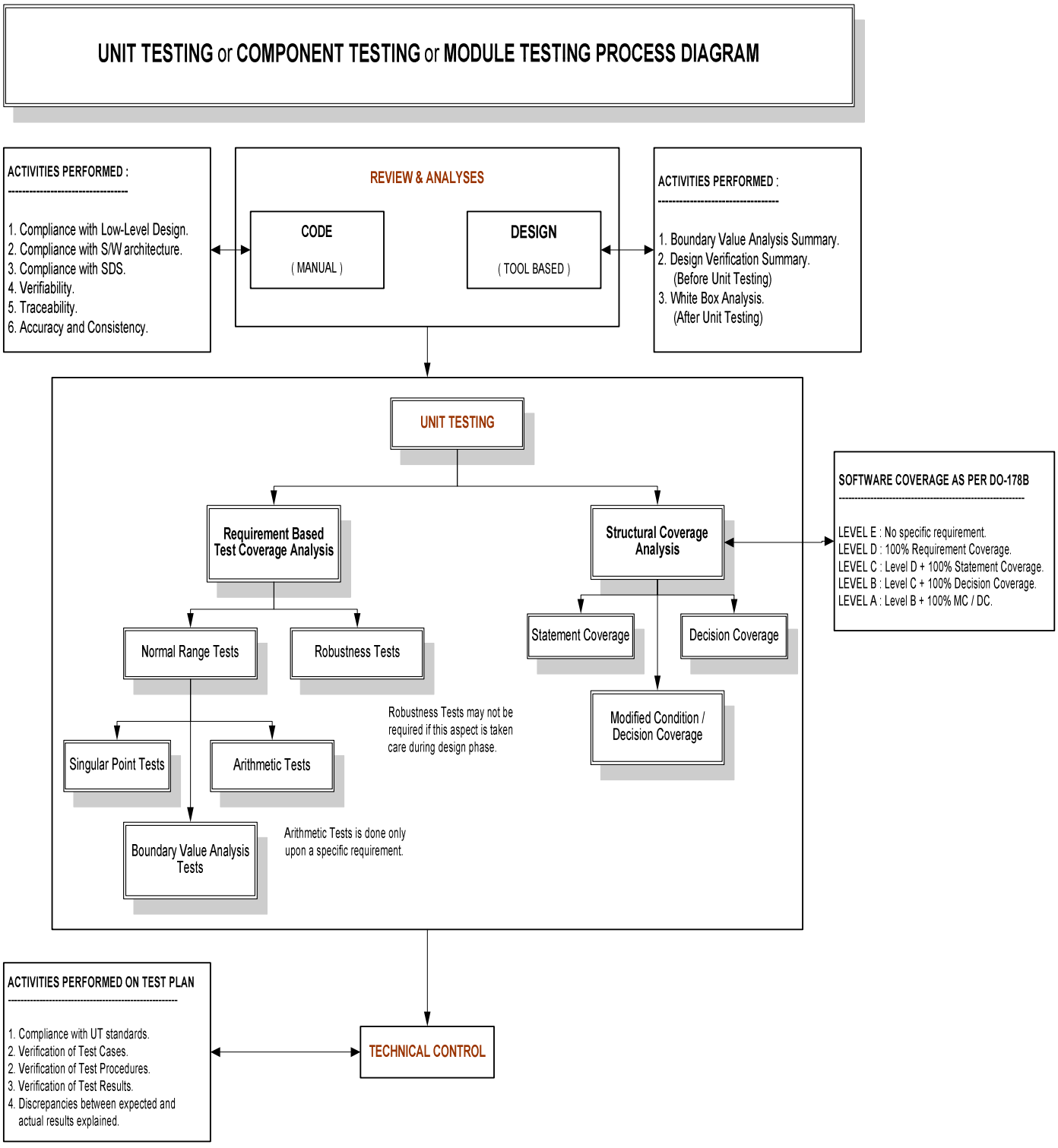
- Requirements document are controlled (checked in CM)
- Test cases and procedures are controlled (checked in CM)

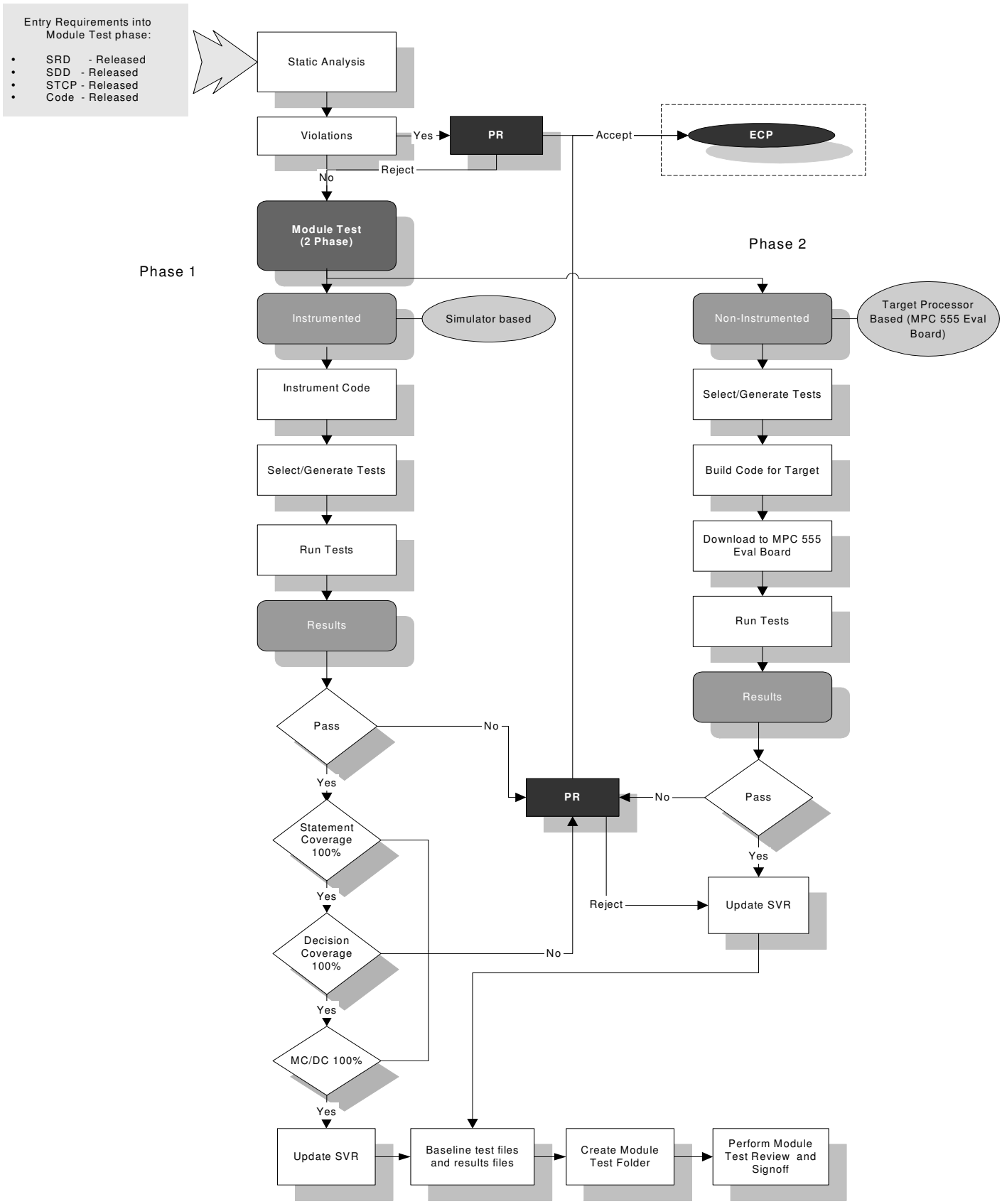
- Source code are controlled (checked in CM)
- Review/QA/CM record are available on the baseline undergoing formal run
- Known problem reports are available and agreed
- Informal metrics on structural coverage are available and agreed
- Informal metrics on test result are available and agreed
- Any deviations if applicable are agreed
- Test environment available as specified in test procedure
- Tool qualification if applicable is completed
- If structural coverage tool is used, the coverage settings in the tool is correct as per the applicable DO-178B level
- Compiler/Assemble should be same as used in development. If there are any difference, it should be documented prior and agreed
- Any test constraint (e.g. modification of source code due to infinite loop, h/w memory mapping, usage of debugger for polling logic) should be justified and documented
- Retrieval of source code, test procedure from CM system is defined and documented to be followed for formal run.
- For Level A and B, tests should be executed by person different from author of test.

9.6 Unit Testing Process

The Unit Test process of a module typically consists of the following phases:

1. Review and Analyses Phase
2. Unit Testing Activity
3. Technical Control / Peer Review of the unit testing activity.


UNIT TESTING or COMPONENT TESTING or MODULE TESTING PROCESS DIAGRAM
ACTIVITIES PERFORMED :
1. Compliance with Low-Level Design.
2. Compliance with S/W architecture.
3. Compliance with SDS.
4. Verifiability.
5. Traceability.
6. Accuracy and Consistency.
REVIEW & ANALYSES
CODE
(MANUAL)
DESIGN
(TOOL BASED)
ACTIVITIES PERFORMED :
1. Boundary Value Analysis Summary.
2. Design Verification Summary.
(Before Unit Testing)
3. White Box Analysis.
(After Unit Testing)
UNIT TESTING
Requirement Based Test Coverage Analysis
Structural Coverage Analysis
SOFTWARE COVERAGE AS PER DO-178B
LEVEL E : No specific requirement.
LEVEL D : 100% Requirement Coverage.
LEVEL C : Level D + 100% Statement Coverage.
LEVEL B : Level C + 100% Decision Coverage.
LEVEL A : Level B + 100% MC / DC.
Normal Range Tests
Robustness Tests
Statement Coverage
Decision Coverage
Robustness Tests may not be required if this aspect is taken care during design phase.
Modified Condition / Decision Coverage
Singular Point Tests
Arithmetic Tests
Boundary Value Analysis Tests
Arithmetic Tests is done only upon a specific requirement.
ACTIVITIES PERFORMED ON TEST PLAN
1. Compliance with UT standards.
2. Verification of Test Cases.
2. Verification of Test Procedures.
3. Verification of Test Results.
4. Discrepancies between expected and actual results explained.
TECHNICAL CONTROL


Entry Requirements into Module Test phase:
SRD - Released
SDD - Released
STCP - Released
Code - Released
Static Analysis
Violations
Yes
PR
Accept
ECP
Reject
No
Module Test (2 Phase)
Phase 1
Phase 2
Instrumented
Simulator based
Non-Instrumented
Target Processor Based (MPC 555 Eval Board)
Instrument Code
Select/Generate Tests
Run Tests
Results
Select/Generate Tests
Build Code for Target
Download to MPC 555 Eval Board
Run Tests
Results
Pass
No
PR
No
Pass
Yes
Yes
Statement Coverage 100%
Yes
Reject
Update SVR
Decision Coverage 100%
No
Yes
MC/DC 100%
Yes
Update SVR
Baseline test files and results files
Create Module Test Folder
Perform Module Test Review and Signoff

9.6.1 Review and Analyses Phase

The objective is to detect and report errors that may have been introduced during the software coding process. These reviews and analyses confirm that the outputs of the software coding process are accurate, complete and can be verified. Primary concerns include correctness of the code with respect to the software requirements and the software architecture, and conformance to the Software Code Standards. These reviews and analyses are usually confined to the Source Code.

Its objectives are:

- **Compliance with the low-level requirements:**
  The objective is to ensure that the Source Code is accurate and complete with respect to the software low-level requirements, and that no Source Code implements an undocumented function.

- **Compliance with the software architecture:**
  The objective is to ensure that the Source Code matches the data flow and control flow defined in the software architecture.

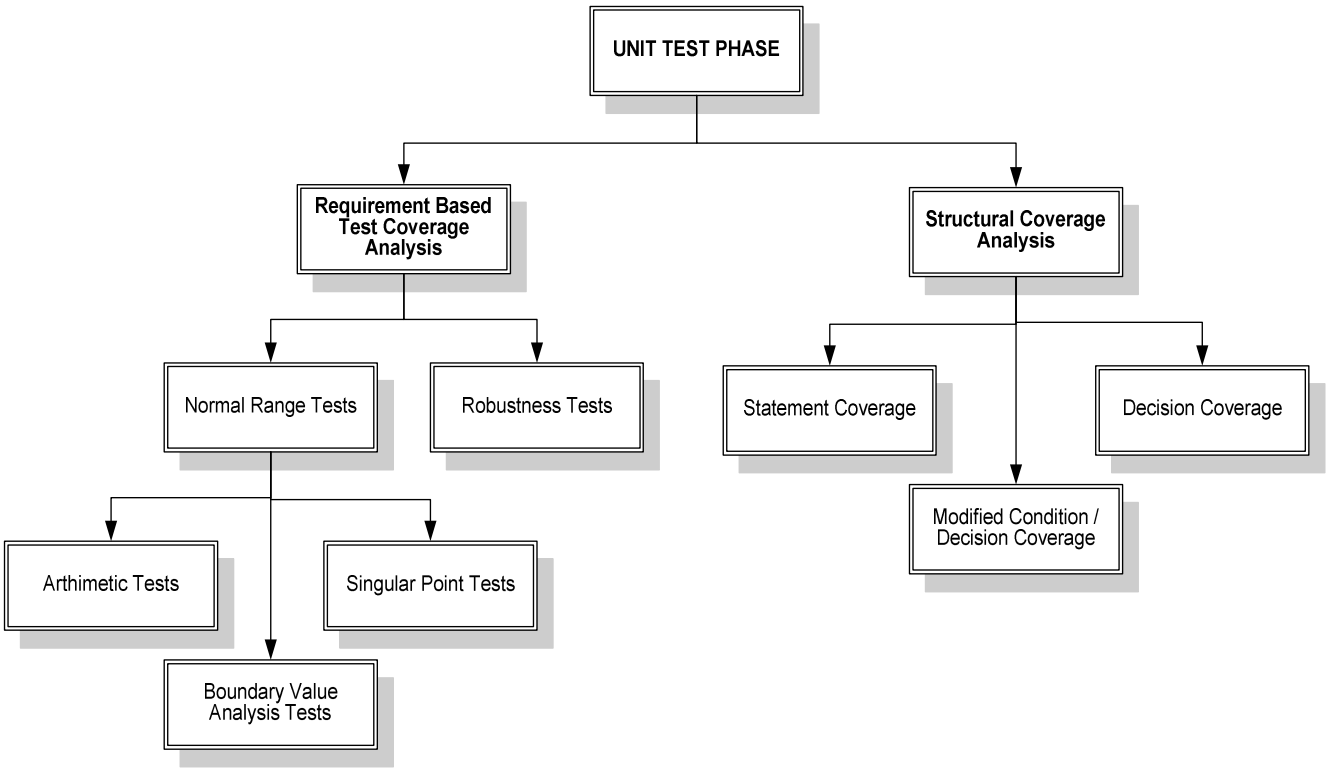
- **Verifiability:**
  The objective is to ensure the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it.

- **Conformance to standards:**
  The objective is to ensure that the Software Code Standards were followed during the development of the code, especially complexity restrictions and code constraints that would be consistent with the system safety objectives. Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions. This analysis also ensures that deviations to the standards are justified.

- **Traceability:**
  The objective is to ensure that the software low-level requirements were developed into Source Code.

- **Accuracy and consistency:**
  The objective is to determine the correctness and consistency of the Source Code, including stack usage, fixed point arithmetic overflow and resolution, resource contention, worst-case execution timing, exception handling, use of un-initialized variables or constants, unused variables or constants, and data corruption due to task or interrupt conflicts.

9.6.2 Unit Testing Phase



9.6.2.1 Requirements Based Test Coverage Analysis

This is done to verify whether the implementation matches the requirements and only the requirements. Software requirements should contain a finite list of behaviors and features, and each requirement should be written to be verifiable. Testing based on requirements is appealing because it is done from the perspective of the user (thus providing a demonstration of intended function), and allows for development of test plans and cases concurrently with development of the requirements. Given a finite list of requirements and a set of completion criteria, requirements-based testing becomes a feasible process, unlike exhaustive testing.

The software verification test cases are to be created based on the software requirements specification. The first step is to develop functional and robustness tests to completely test and verify the implementation of the software requirements. The second step is to measure coverage (functional and structural). The measure of structural coverage will help provide an indication of the software verification campaign completion status. The tests stop criteria is not limited to a specific step but rather applied for all tests. For example, some high level requirements can be covered by integration tests, i.e., structural coverage are measured on all tests levels.

The objectives of Requirements Based Testing are as follows:

1. At least one test case should be associated with each software requirement.
2. The test cases should satisfy the criteria for normal and abnormal range of inputs.

The objective of this analysis is to determine how well the requirements-based testing verified the implementation of the software requirements. This analysis may reveal the need for additional requirements-based test cases. The requirements-based test coverage analysis should show that:

- Test cases exist for each software requirement.
- Test cases satisfy the criteria of normal and robustness testing

Requirements-based tests are subdivided into two categories:

- Normal Range Tests.
- Robustness Tests.

9.6.2.1.1 Robustness or Abnormal Tests

The objective of robustness test cases is to demonstrate the ability of the software to respond to abnormal inputs and conditions. This is done to verify robustness of the code under test. In robustness testing, the tester should verify the behavior of the module under test when subjected to abnormal operational values. Robustness test cases enable visibility of how the software responds to a range of abnormal inputs and conditions. Robustness tests will be incorporated into functional tests if specifications exist for handling invalid input data. The Software Design Standard **[SDS]** should mandatory require that handling of invalid input data be specified in the Software Design Document **[SDD]**.

Robustness test cases include:

- Exercise real and integer inputs using equivalence class of invalid boundary values.
- System initialization exercised with abnormal conditions.
- Determine the possible failure modes of incoming data especially complex, digital data strings from an external system.
- Compute out of range loop counts as appropriate.
- Check for arithmetic overflow for time related functions.
- Exercise transitions that are not allowed by the software requirements for state transitions.

9.6.2.1.2 Normal Range Tests

The objective of normal range test cases is to demonstrate the ability of the software to respond to normal inputs and conditions. Normal range test cases enable visibility of how the software responds to a range of normal inputs and conditions. Normal range test cases include:

- Real and integer input variables should be exercised using valid equivalence classes and valid boundary values.
- For time-related functions, such as filters, integrators and delays, multiple iterations of the code should be performed to check the characteristics of the function in context.

- For state transitions, test cases should be developed to exercise the transitions possible during normal operation.
- For software requirements expressed by logic equations, the normal range test cases should verify the variable usage and the Boolean operators.

The following sub-categories of tests are defined to achieve the above mentioned objectives:

- Arithmetic Tests.
- Singular Point Tests.
- Boundary Value Analysis Tests.

In this type of testing tester should ensure each function entry and exit is encountered at least once and with the inputs subjected to normal operational values. Nodes are mathematical operations (+, -, *, /) within both control flow and signal flow diagrams.

- Floating-point nodes shall be tested to show that the operations are performed correctly.
- Integer or fixed-point nodes shall be stressed to a minimum and a maximum value.
  - Integer counters are a special case. If they are incremented only within a single unit and they are limited, they need only be tested at the limits. If they are incremented in multiple units they shall be tested at -32768 and 32767.
- The following restrictions shall be applied:
  - Addition and Subtraction: A test case must exist where all inputs are not 0.0
  - Multiplication: A test must exist where all inputs are not 1.0
  - Division: A test case must exist where the numerator and denominator are not 0.0
- During test development, the tester shall examine the logic for the following conditions:
  - divide by zero condition for a divider
  - a negative input to a square root
  - a negative input to the LOG or LN function
- Bullet proofing will be used for these conditions. Any unit that does not have this bullet proofing will be rejected out as not testable.

9.6.2.1.2.1 Arithmetic Tests

The goal of arithmetic tests is to verify all computations and their precision using random input values. This is achieved by fixing a random value (avoid round numbers) to all variables within the arithmetic expressions and calculated manually. The result is verified by executing the unit code. The accuracy of the computation will be calculated and expressed in terms of number of deltas.

In this type of testing each arithmetic expression is evaluated for its outcome when the resolution or DELTA of its output value while DELTA for all input values is specified a priori.

If Z = X +/- Y then **DELTA** of Z = (**DELTA** of X + **DELTA** of Y)

If Z = X * Y then **DELTA** of Z = (X * **DELTA** of Y + Y * **DELTA** of X)

If Z = X / Y then **DELTA** of Z = (Y * **DELTA** of X + X * **DELTA** of Y) / Square (Y)

9.6.2.1.2.2 Singular Point Tests

In this type of testing each comparison in the logical condition is evaluated with the values representing all possibilities of the condition along with a resolution called DELTA. The purpose of this test is to implement conditions such as comparisons that shall be verified by making minor variations (called delta) to the operands involved. For each inequality (<, <=, >, >=) we will have two possible test cases:

- Case where the condition is verified
- Case where the condition is not verified

The singular point tests will be performed about the same point or value. Comparisons shall be tested as follows:

Floating-point except for equal to (=) and not equal to (!=) : Within 10% above and within 10% below (see note)

Floating-point equal to (=) and not equal (!=) to : Within 10% above, equal to, within 10% below (see note)

Signed and Unsigned Integer : 1 count above, equal to, 1 count below

Discrete Word : Equal to and not equal to

Boolean : Equal to and not equal to

Note: For the comparison **"X < Y"**, there must be one test case where **Y < X < (1.1 * Y)** and another test case where **(0.9 * Y) < X < Y**, where 1.1 and 0.9 are DELTA. X and Y may be reversed. If the value is 0.0, use +1.0 and -1.0 instead of 10% above and below.

- During test development, the tester shall examine the logic for conditions other than equal to and not equal to on discrete words and Boolean. Any unit that has one of these conditions shall be kicked out as not testable.
- If there is an equal to or not equal to comparison on floating-point values, consult with the unit test coordinator. The unit may not work.
- Some examples of conditional testing:

Floating Point	***Integer***
Condition: X < 55.0	Condition: X > 100
10% above test case X = 60.5	+1 test case X = 101
= test case not required	= test case X = 100
10% below test case X = 49.5	-1 test case X = 99

The following will be the test cases for the different inequalities:

Case A>B

- A=B (the condition is not verified, hence false)
- A=B+delta (the condition is verified, hence true)

Case A>=B

- A=B (the condition is verified)
- A=B-delta (the condition is not verified)

Case A<B

- A=B (condition is not verified)
- A=B-delta (condition is verified)

Case A<=B

- A=B (condition is verified)
- A=B+delta (condition is not verified)

9.6.2.1.2.3 Boundary Value Tests

This is done to verify whether the test inputs are tested at the minimum, nominal and median ((minimum + maximum)/2) values. In this testing inputs are made to represent their boundary limits of the range. The minimum, maximum and median value of each variable in an expression and computation must be verified to make sure that there is no overflow.

For all numerical variables as input, we have to perform three tests:

- A test for the maximum value of the type of that variable,
- A test for the minimum value of the type of that variable and
- A test for the median value of the type of that variable.

Also the maximum, minimum and median for every calculation in the modules will be manually calculated and compared to the results obtained using the testing tool. Moreover, some variables may be restricted to a range of variation smaller than the range of its type; in this case two additional tests must be realized, each testing the limit of the restricted range.

Boundary Value Analysis is required when inputs or initial values for states are used in mathematical operations and shall be applied as shown below:

Floating-point	: not required	
Integer	: -32768 to 32767 (16-bit)	or -2147483648 to 2147483647 (32-bit)
Unsigned Integer	: 0 to 65535 (16-bit)	or 0 to 4294967295 (32-bit)
Discrete Word	: 0000H to FFFFH (16-bit)	or 00000000H to FFFFFFFFH (32-bit)
Boolean	: FALSE to TRUE	

For floating-point values, if tools require ranges, values should be selected to functionally test the unit. In general, -100,000.0 to 100,000.0 will adequately test the unit

- For non floating-point adjustments, the min and max adjustment range values shall be used instead of the values described above. For floating-point adjustments, if tools require ranges, the adjustment range values should be used.
- For counters, see the exception under Nodal coverage.

9.6.2.1.2.4 Basis Path Tests

It is a testing mechanism is proposed by McCabe. The aim of this is derive a logical complexity measure of a procedural design and use this as a guide for defining a basic set of execution paths. The test cases, which exercise the basic set, will execute every statement at least once.

The virtues of the basis path testing are defined by Cyclomatic Complexity. The Cyclomatic Complexity gives a quantitative measure of the logical complexity. This value gives the number of independent paths in the basis set and an upper bound for the number of tests to ensure that each statement is excused at least once. An independent path is any path through a program that introduces at least one new set of processing statements or a new condition.

The objective of independent path testing is to exercise all independent execution paths through a code component. If all of the independent paths are executed then all statements in the code component must have been executed at least once. Also, all conditional statements are tested for both true and false.

- These testing techniques shall only be used at the module/function level because as a program increases in size during integration then the number of paths grows quickly, thus making it infeasible to use these techniques.
- Independent path testing does not test all possible combinations of all paths through the program.

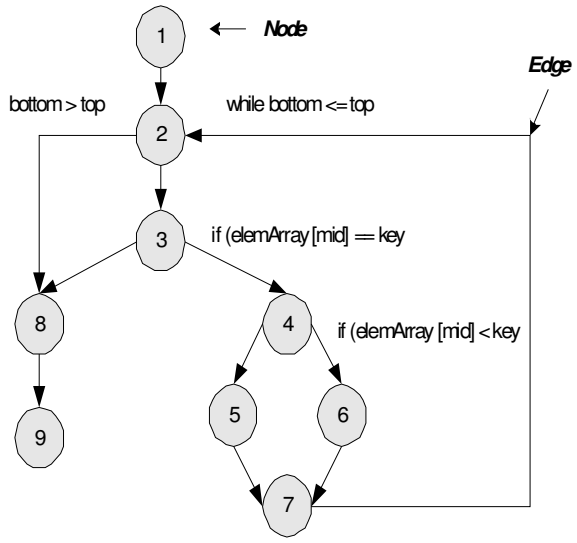
The following steps shall be followed for independent path testing:

1. Create a flow graph for the code component.

   This flow graph can be obtained from LDRA Testbed (refer to relevant LDRA Testbed documentation) or created by hand.

A flow graph consists of nodes that represent decisions and edges that show the control flow. Refer to Figure 5.3.4.1-1

Example: Flow graph for the binary search routine.



FLOW GRAPH

- An independent path is one that traverses at least one new edge in the flow graph (i.e. exercising one or more new program conditions)
- Both the true and false branches of all conditions must also be executed.

2. Derive the independent paths.

   From the example flow graph defined in Figure 5.3.4.1-1 we can derive the following independent paths:

   1, 2, 3, 8, 9

   1, 2, 3, 4, 6, 7, 2

   1, 2, 3, 4, 5, 7, 2

   1, 2, 3, 4, 6, 7, 2, 8, 9

   If all of these paths are executed then:

   a) every statement in the code has be executed at least once.

   b) every branch has been exercised for true and false conditions.

   The number of independent paths can be derived by calculating the Cyclomatic Complexity of the flow graph. This is achieved with the following formula:

   CC(G) = Number (edges) – Number (nodes) + 2

Thus the CC of the flow graph in Figure 5.3.4.1-1 is:

11 – 9 + 2

CC = 4

The Cyclomatic Complexity can also be obtained from LDRA Testbed (Refer to relevant LDRA Testbed documentation).

1. Design Test Cases

   The minimum number of test cases required to test all program paths is equal the Cyclomatic Complexity

9.6.2.2 Structural Coverage (Code Coverage) Analysis

Test coverage analysis is a two step process, involving requirements-based coverage analysis and structural coverage analysis. The first step analyzes the test cases in relation to the software requirements to confirm that the selected test cases satisfy the specified criteria. The second step confirms that the requirements-based test procedures exercised the code structure. Structural coverage analysis may not satisfy the specified criteria. Additional guidelines are provided for resolution of such situations as dead code.

Code coverage analysis is the process of:

- Finding areas of a program not exercised by a set of test cases,
- Creating additional test cases to increase coverage, and
- Determining a quantitative measure of code coverage, which is an indirect measure of quality.

An optional aspect of code coverage analysis is:

- Identifying redundant test cases that do not increase coverage.

Code coverage analysis is a structural testing technique (AKA glass box testing and white box testing). **Structural testing compares test program behavior against the apparent intention of the source code.** This contrasts with functional testing (AKA black-box testing), which compares test program behavior against a requirements specification. Structural testing examines how the program works, taking into account possible pitfalls in the structure and logic. **Functional testing examines what the program accomplishes, without regard to how it works internally.**

The objective of this analysis is to determine which code structure was not exercised by the requirements-based test procedures. The requirements-based test cases may not have completely exercised the code structure, so structural coverage analysis is performed and additional verification produced to provide structural coverage. Guidance includes:

- The analysis should confirm the degree of structural coverage appropriate to the software level.
- The structural coverage analysis may be performed on the Source Code, unless the software level is A and the compiler generates object code that is not directly traceable to Source Code statements. Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences. A compiler-generated array-bound check in the object code is an example of object code that is not directly traceable to the Source Code.
- The analysis should confirm the data coupling and control coupling between the code components.

The purpose of structural coverage analysis with the associated structural coverage analysis resolution is to complement requirements-based testing as follows:

- Provide evidence that the code structure is verified to the degree required for the applicable software level
- Provide a means to support demonstration of absence of unintended functions
- Establish the thoroughness of requirements-based testing.

The following sub-categories of tests are defined to achieve the above mentioned objectives:

1. Statement Coverage
2. Decision Coverage
3. Modified Condition / Decision Coverage (MC / DC)

9.6.2.2.1 Statement Coverage

This measure reports whether each executable statement is encountered. To achieve statement coverage, every executable statement in the program is invoked at least once during software testing. Achieving statement coverage shows that all code statements are reachable (in the context of DO-178B, reachable based on test cases developed from the requirements). In summary, this measure is affected more by computational statements than by decisions.

Consider the following code segment:

```
if ((x > 1) && (y = 0))
{
    z = z / x;
}
if ((z = 2) || (y > 1))
{
    z = z + 1;
}
```

By choosing x = 2, y = 0, and z = 4 as input to this code segment, every statement is executed at least once.

Statement coverage by default also include

- Condition Coverage and
- Multiple Conditions Coverage.
- Loop Coverage

Statement coverage is calculated as follows

$$\text{Statement Coverage} = \frac{\text{Number of Executable Statements Executed}}{\text{Total Number of Executable Statements}} * 100\ \%$$

Advantages :

- The chief advantage of this measure is that it can be applied directly to object code and does not require processing source code.

Limitations :

- Statement coverage does not report whether loops reach their termination condition - only whether the loop body was executed. With C, C++, and Java, this limitation affects loops that contain break statements.
- Since do-while loops always execute at least once, statement coverage considers them the same rank as non-branching statements.
- Statement coverage is completely insensitive to the logical operators (|| and &&).
- Statement coverage cannot distinguish consecutive switch labels.

9.6.2.2.2 Condition Coverage.

Condition coverage reports the true or false outcome of each boolean sub-expression, separated by logical-and and logical-or if they occur. Condition coverage measures the sub-expressions independently of each other. This measure is similar to decision coverage but has better sensitivity to the control flow. Condition coverage means that every condition has been made to take true and false.

The branch condition coverage is calculated as follows

$$\text{Branch Condition Coverage} = \frac{\text{Number of Boolean Operand Values Executed}}{\text{Total Number of Boolean Operand Values}} * 100\ \%$$

9.6.2.2.3 Multiple Condition Coverage

Multiple condition coverage reports whether every possible combination of boolean sub-expressions occurs. As with condition coverage, the sub-expressions are separated by logical-and and logical-or, when present. The test cases required for full multiple condition coverage of a condition are given by the logical operator truth table for the condition. Multiple conditions coverage means that every possible combination of the logical condition must be executed during at least once and every possible condition must evaluated to both true and false. However, full condition coverage does not guarantee full decision coverage

Limitations :

- A disadvantage of this measure is that it can be tedious to determine the minimum set of test cases required, especially for very complex boolean expressions.
- An additional disadvantage of this measure is that the number of test cases required could vary substantially among conditions that have similar complexity.

The multiple condition combination coverage is calculated as follows

Multiple Condition Combination Coverage =

$$\frac{\text{Number of Boolean Operand Value Combinations Executed}}{\text{Total Number of Boolean Operand Value Combinations}} * 100\ \%$$

9.6.2.2.4 Loop Coverage

This measure reports whether you executed each loop body zero times, exactly once, and more than once (consecutively). For do-while loops loop coverage reports whether you executed the body exactly once, and more than once. The valuable aspect of this measure is determining whether while-loops and for-loops execute more than once, information not reported by others measure.

Loops shall be tested to verify that the loop executes appropriately.

- If the loop executes for a fixed number of iterations this will be automatically achieved as long as the loop is executed.
- If the loop executes for a variable number of cycles, test cases shall be included that test the loop at the minimum and maximum number of cycles.

- Where loops set array values, test cases shall be included that use the lowest and highest array indices that the loop can affect. Every array value that can be affected by the test case shall be examined and the number of array values shall be identically equal to the number that are set by the loop.
  - If this cannot be done, then the unit shall be rejected as not testable. (For example, a unit with a loop is not testable if during the execution of the loop a single array value is set more than once.)

In case of a static loop (Number of iterations in the loop is controlled by a constant) like
for (I=0; I<10; I++) → this loop would not have 100% loop coverage since it cannot undergo 0 iteration and 1 iteration. At any instant only 2 or more iterations are possible. So the loop coverage will be *1/3* ➔ *33.33%.*

9.6.2.2.5 Decision Coverage

This measure reports whether Boolean expressions tested in control structures (such as the if-statement and while-statement) evaluated to both true and false. The entire Boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators. Additionally, this measure includes coverage of switch-statement cases, exception handlers, and interrupts handlers. Also known as: branch coverage, all-edges coverage, basis path coverage, and decision-decision-path testing. "Basis path" testing selects paths that achieve decision coverage.

Advantages :

- Simplicity without the problems of statement coverage.

Limitations :

- A disadvantage of this measure ignores branches within boolean expressions which occur due to short-circuit operators.

Decision coverage requires two test cases: one for a true outcome and another for a false outcome. For simple decisions (i.e., decisions with a single condition), decision coverage ensures complete testing of control constructs. For the decision (A or B), test cases (TF) and (FF) will toggle the decision outcome between true and false. However, the effect of B is not tested; that is, those test cases cannot distinguish between the decision (A or B) and the decision A.

The decision coverage is calculated as follows

$$\text{Decision Coverage} = \frac{\text{Number of Executed Decision Outcomes}}{\text{Total Number of Decision Outcomes}} * 100\ \%$$

9.6.2.2.5 Logical Combinatory or Modified Condition or Decision Coverage [MC/DC]

This measure requires enough test cases to verify every condition can affect the result of its encompassing decision. This measure was created at Boeing and is required for aviation software by **RCTA/DO-178B**. Modified Condition Decision Coverage (MCDC) is a pragmatic compromise which requires fewer test cases than Branch Condition Combination Coverage. It is widely used in the development of avionics software, as required by **RTCA/DO-178B**. Modified Condition Decision Coverage requires test cases to show that each Boolean can independently affect the outcome of the decision. This is less than all the combinations (as required by Branch Condition Combination Coverage).

Definition of Modified Condition Decision Coverage [MC/DC]:

Every decision has taken all possible outcomes at least once and every condition in a decision has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying only that condition while holding all other conditions constant.

- A decision is a single IF statement, a decision block in a flow chart, a multi-way branch, or a switch on a signal flow diagram.
- A condition is a single Boolean-valued expression that cannot be broken down into simpler Boolean expressions. Complex Boolean expressions within a decision can be made up of multiple conditions.
- MC/DC applies to:
  - Boolean expressions within an IF statement
  - Boolean expressions within a decision block on a flow chart
  - Boolean expressions in the input column of a truth table
  - Boolean expressions on the right side of an assignment statement
  - Collections of Boolean symbols on a signal flow diagram, the output of which is an output of the module or an input to a nonlinear symbol like a switch
- Conditions (AND, OR, NAND, NOR, XOR, XNOR) shall be tested in accordance with the RTCA/DO178B for Modified Conditions / Decision Coverage (MC/DC) as show in the table below:

Condition	Case (T=True, F=False)			
	TT	TF	FT	FF
AND	Y	Y	Y	NA
OR	NA	Y	Y	Y
NAND	Y	Y	Y	NA
NOR	NA	Y	Y	Y
XOR	Y	Y	Y	Y
XNOR	Y	Y	Y	Y

- For conditions with more than two inputs, this requirement shall be extended by one additional case per input. For example, an AND condition with three inputs requires TTT, TTF, TFT, and FTT.
- Combinations of different types of conditions shall be tested so that each condition is verified according to the rules described above.

1. Conditions that cannot be verified as described because of coupling shall be documented in the test documentation.
   - Strong coupling exists when conditions within a decision cannot be changed independently of each other
   - Weak coupling exists when conditions overlap so that sometimes a change to one condition affects another

In this type of testing all possibilities of logical combination, multiple conditions and modified conditions are tested. Also a test case shall be written to verify whether a condition is shown to independently affect a decision's outcome by varying only that condition while holding all other conditions constant which, is nothing but MC/DC.

If there are "***N***" inputs

Then minimum number of test cases required to perform MC/DC is

"***N+1***"

Consider the below example

A	**or**	**B**	**and**	**C**	**RESULT**	**A**	**and**	**B**	**or**	**C**	**NUM**
True		False		True	True	True		False		True	1
False		False		True	False	True		False		*False*	2
False		*True*		True	True	True		*True*		False	3
False		True		*False*	False	*False*		True		False	4

In summary for A or (B and C):

A is shown to independently affect the outcome of the decision condition by test cases 1 and 2;

B is shown to independently affect the outcome of the decision condition by test cases 2 and 3;

C is shown to independently affect the outcome of the decision condition by test cases 3 and 4.

In summary for A and (B or C):

C is shown to independently affect the outcome of the decision condition by test cases 1 and 2;

B is shown to independently affect the outcome of the decision condition by test cases 2 and 3;

A is shown to independently affect the outcome of the decision condition by test cases 3 and 4.

The Modified Condition / Decision Coverage [MCDC] is calculated as follows

$$MC/DC = \frac{\text{Number of Boolean Operand Values shown to independently affect the decision}}{\text{Total Number of Boolean Operands}} * 100\%$$

9.6.2.2.6 Path Coverage

This measure reports whether each of the possible paths in each function have been followed. A path is a unique sequence of branches from the function entry to the exit. Also known as predicate coverage. Predicate coverage views paths as possible combinations of logical conditions. Since loops introduce an unbounded number of paths, this measure considers only a limited number of looping possibilities. A large number of variations of this measure exist to cope with loops. Boundary-interior path testing considers two possibilities for loops: zero repetitions and more than zero repetitions [Ntafos1988]. For do-while loops, the two possibilities are one iteration and more than one iteration. Path coverage has the advantage of requiring very thorough testing. Path coverage has two severe disadvantages. The first is that the number of paths is exponential to the number of branches. For example, a function containing 10 if-statements has 1024 paths to test. Adding just one more if-

statement doubles the count to 2048. The second disadvantage is that many paths are impossible to exercise due to relationships of data. For example, consider the following C/C++ code fragment:

```
if (success)
   statement1;
statement2;
if (success)
   statement3;
```

Path coverage considers this fragment to contain 4 paths. In fact, only two are feasible: success=false and success=true.

9.6.2.2.7 Other Type of Coverage's

- **CALL COVERAGE**

  This measure reports whether you executed each function call. The hypothesis is that faults commonly occur in interfaces between modules. It is Also known as call pair coverage.

- **DATA FLOW COVERAGE**

  This variation of path coverage considers only the sub-paths from variable assignments to subsequent references of the variables. The advantage of this measure is the paths reported have direct relevance to the way the program handles data. One disadvantage is that this measure does not include decision coverage. Another disadvantage is complexity.

  Researchers have proposed numerous variations, all of which increase the complexity of this measure. For example, variations distinguish between the use of a variable in a computation versus a use in a decision, and between local and global variables. As with data flow analysis for code optimization, pointers also present problems.

- **LINEAR CODE SEQUENCE AND JUMP (LCSAJ) COVERAGE**

  This variation of path coverage considers only sub-paths that can easily be represented in the program source code, without requiring a flow graph [Woodward1980]. . An LCSAJ is a sequence of source code lines executed in sequence. This "linear" sequence can contain decisions as long as the control flow actually continues from one line to the next at run-time. Sub-paths are constructed by concatenating LCSAJs. Researchers refer to the coverage ratio of paths of length n LCSAJs as the test effectiveness ratio (TER) n+2. The advantage of this measure is that it is more thorough than decision coverage yet avoids the exponential difficulty of path coverage. The disadvantage is that it does not avoid infeasible paths.

❖ **OBJECT CODE BRANCH COVERAGE**

This measure reports whether each machine language conditional branch instruction both took the branch and fell through. This measure gives results that depend on the compiler rather than on the program structure since compiler code generation and optimization techniques can create object code that bears little similarity to the original source code structure.

Since branches disrupt the instruction pipeline, compilers sometimes avoid generating a branch and instead generate an equivalent sequence of non-branching instructions. Compilers often expand the body of a function inline to save the cost of a function call. If such functions contain branches, the number of machine language branches increases dramatically relative to the original source code. It's better off testing the original source code since it relates to program requirements better than the object code.

❖ **RELATIONAL OPERATOR COVERAGE**

This measure reports whether boundary situations occur with relational operators (<, <=, >, >=). The hypothesis is that boundary test cases find off-by-one errors and mistaken uses of wrong relational operators such as < instead of <=. For example, consider the following C/C++ code fragment:

```
if (a < b)
   statement;
```

Relational operator coverage reports whether the situation a==b occurs. If a==b occurs and the program behaves correctly, you can assume the relational operator is not suppose to be <=.

❖ **WEAK MUTATION COVERAGE**

This measure is similar to relational operator coverage but much more general [Howden1982]. It reports whether test cases occur which would expose the use of wrong operators and also wrong operands. It works by reporting coverage of conditions derived by substituting (mutating) the program's expressions with alternate operators, such as "-" substituted for "+", and with alternate variables substituted. This measure interests the academic world mainly. Caveats are many; programs must meet special requirements to enable measurement.

9.6.3 Technical Control or Peer Review Phase

The objective of these reviews and analyses is to ensure that the testing of the code was developed and performed accurately and completely. The topics include:

a. Test cases:

The verification of test cases is presented in paragraph.

b. Test procedures:

The objective is to verify that the test cases were accurately developed into test procedures and expected results.

c. Test results:

The objective is to ensure that the test results are correct and that discrepancies between actual and expected results are explained.

In case of any problem found during the unit testing phase, problem reporting mechanism is followed as shown below.

9.6.4 Unit Test Procedure or Test Script Content

Typical content of test script is mentioned below. The comments, definition to different files can be done as required by the project based on tool selection.

```
%% ------------------------------------------------------------------------
%% -- Include Files
%% ------------------------------------------------------------------------
%% Include the header files


%% ------------------------------------------------------------------------
%%  Data declarations
%% ------------------------------------------------------------------------
%% Declare all the extern data required for test
%% Declare any test data required for test


%% ------------------------------------------------------------------------
%%  Stub
%% ------------------------------------------------------------------------
%% Define the stubs simulating the external function than
%% module under test


%% ------------------------------------------------------------------------
%% Environment Block
%% ------------------------------------------------------------------------
%% This block  the will have the initialization of all data under
%% test. This block will be tested in all test case unless the
%% test case overrides the input or, expected value
        Variable 1, initial value = ?, Expected value = ?
        Variable 2, initial value = ?, Expected value = ?
```

Variable 3, initial value = ?, Expected value = ?

%% ---

%% Test Procedure

%% ---

%% Test Procedure ID:

%% Upward Trace: Test Case Identifier

Variable 1, initial value = ?, Expected value = ?

Variable 2, initial value = ?, Expected value = ?

#<Call to unit under test>

%% Test Procedure ID:

%% Upward Trace: Test Case Identifier

Variable 1, initial value = ?, Expected value = ?

Variable 2, initial value = ?, Expected value = ?

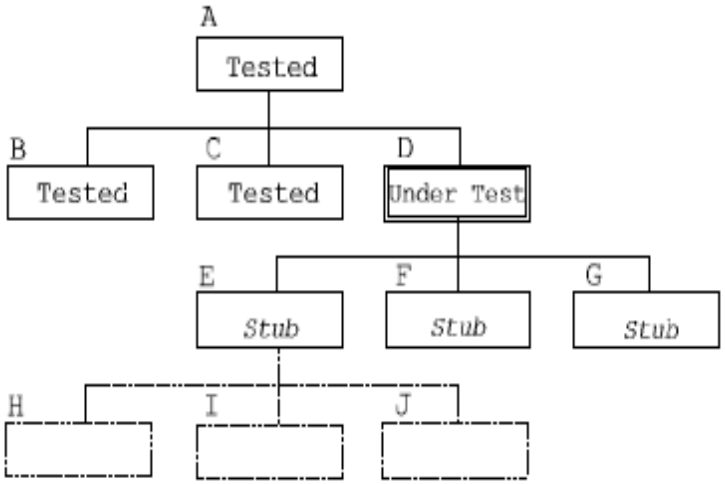
Variable 3, initial value = ?, Expected value = ?

#<Call to unit under test>

9.7 Organizational Approach to Unit Testing

When developing a strategy for unit testing, there are three basic organizational approaches that can be taken. These are *top down*, *bottom up* and *isolation*.

9.7.1 Top-Down Approach to Unit Testing



In top down unit testing, individual units are tested by using them from the units which call them, but in isolation from the units called. The unit at the top of a hierarchy is tested first, with all called units replaced by stubs. Testing continues by replacing the stubs with the actual called units, with lower level units being stubbed. This process is repeated until the lowest level units have been tested. Top down testing requires test stubs, but not test drivers.

The above figure illustrates the test stubs and tested units needed to test unit D, assuming that units A, B and C have already been tested in a top down approach. A unit test plan for the program shown in below figure, using a strategy based on the top down organizational approach, could read as follows:

Step (1)

Test unit A, using stubs for units B, C and D.

Step (2)

Test unit B, by calling it from tested unit A, using stubs for units C and D.

Step (3)

Test unit C, by calling it from tested unit A, using tested units B and a stub for unit D.

Step (4)

Test unit D, by calling it from tested unit A, using tested unit B and C, and stubs for units E, F and G.

Step (5)

Test unit E, by calling it from tested unit D, which is called from tested unit A, using tested units B and C, and stubs for units F, G, H, I and J.

Step (6)

Test unit F, by calling it from tested unit D, which is called from tested unit A, using tested units B, C and E, and stubs for units G, H, I and J.

Step (7)

Test unit G, by calling it from tested unit D, which is called from tested unit A, using tested units B, C, E and F, and stubs for units H, I and J.

Step (8)

Test unit H, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F and G, and stubs for units I and J.

Step (9)

Test unit I, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F, G and H, and a stub for units J.

Step (10)

Test unit J, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F, G, H and I.

9.7.1.1 Advantages

Top down unit testing provides an early integration of units before the software integration phase. In fact, top down unit testing is really a combined unit test and software integration strategy.

The detailed design of units is top down, and top down unit testing implements tests in the sequence units are designed, so development time can be shortened by overlapping unit testing with the detailed design and code phases of the software lifecycle. In a conventionally structured design, where units at the top of the hierarchy provide high level functions, with units at the bottom of the hierarchy implementing details, top down unit testing will provide an early integration of 'visible' functionality. This gives a very requirements oriented approach to unit testing. Redundant functionality in lower level units will be identified by top down unit testing, because there will be no route to test it. (However, there can be some difficulty in distinguishing between redundant functionality and untested functionality).

9.7.1.2 Limitations

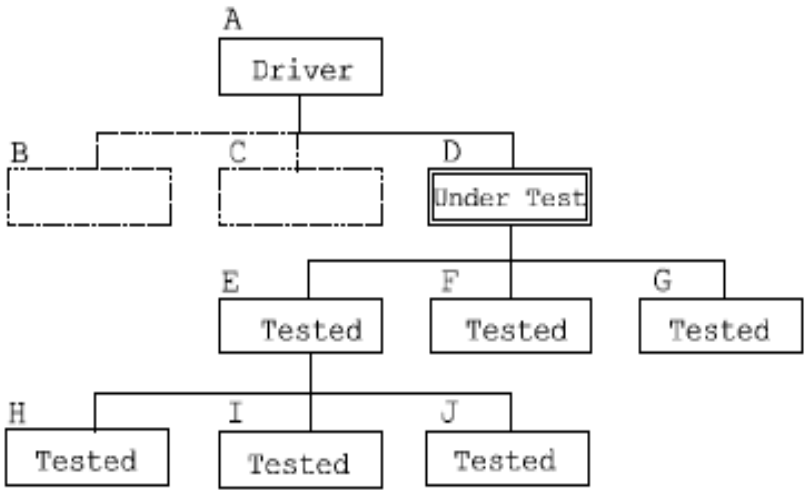
Top down unit testing is controlled by stubs, with test cases often spread across many stubs. With each unit tested, testing becomes more complicated, and consequently more expensive to develop and maintain. As testing progresses down the unit hierarchy, it also becomes more difficult to achieve the good structural coverage which is essential for high integrity and safety critical applications, and which are required by many standards. Difficulty in achieving structural coverage can also lead to confusion between genuinely redundant functionality and untested functionality. Testing some low level functionality, especially error handling code, can be totally impractical.

Changes to a unit often impact the testing of sibling units and units below it in the hierarchy. For example, consider a change to unit D. obviously, the unit test for unit D would have to change and be repeated. In addition, unit tests for units E, F, G, H, I and J, which use the tested unit D, would also have to be repeated. These tests may also have to change themselves, as a consequence of the change to unit D, even though units E, F, G, H, I and J had not actually changed. This leads to a high cost associated with retesting when changes are made, and a high maintenance and overall lifecycle cost.

The design of test cases for top down unit testing requires structural knowledge of when the unit under test calls other units. The sequence in which units can be tested is constrained by the hierarchy of units, with lower units having to wait for higher units to be tested, forcing a 'long and thin' unit test phase. (However, this can overlap substantially with the detailed design and code phases of the software lifecycle).

The relationships between units in the example program in above figure is much simpler than would be encountered in a real program, where units could be referenced from more than one other unit in the hierarchy. All of the disadvantages of a top down approach to unit testing are compounded by a unit being referenced from more than one other unit.

9.7.2 Bottom-Up Approach to Unit Testing



In bottom up unit testing, units are tested in isolation from the units which call them, but using the actual units called as part of the test. The lowest level units are tested first, then used to facilitate the testing of higher level units. Other units are then tested, using previously tested called units. The process is repeated until the unit at the top of the hierarchy has been tested.

Bottom up testing requires test drivers, but does not require test stubs. The below figure illustrates the test driver and tested units needed to test unit D, assuming that units E, F, G, H, I and J have already been tested in a bottom up approach. A unit test plan for the program shown in above figure, using a strategy based on the bottom up organizational approach, could read as follows:

Step (1)

(Note that the sequence of tests within this step is unimportant, all tests within step 1 could be executed in parallel.)

Test unit H, using a driver to call it in place of unit E;

Test unit I, using a driver to call it in place of unit E;

Test unit J, using a driver to call it in place of unit E;

Test unit F, using a driver to call it in place of unit D;

Test unit G, using a driver to call it in place of unit D;

Test unit B, using a driver to call it in place of unit A;

Test unit C, using a driver to call it in place of unit A.

Step (2)

Test unit E, using a driver to call it in place of unit D and tested units H, I and J.

Step (3)

Test unit D, using a driver to call it in place of unit A and tested units E, F, G, H, I and J.

Step (4)

Test unit A, using tested units B, C, D, E, F, G, H, I and J.

9.7.2.1 Advantages

Like top down unit testing, bottom up unit testing provides an early integration of units before the software integration phase. Bottom up unit testing is also really a combined unit test and software integration strategy. All test cases are controlled solely by the test driver, with no stubs required. This can make unit tests near the bottom of the unit hierarchy relatively simple. (However, higher level unit tests can be very complicated). Test cases for bottom up testing may be designed solely from functional design information, requiring no structural design information (although structural design information may be useful in achieving full coverage). This makes the bottom up approach to unit testing useful when the detailed design documentation lacks structural detail. Bottom up unit testing provides an early integration of low level functionality, with higher level functionality being added in layers as unit testing progresses up the unit hierarchy. This makes bottom up unit testing readily compatible with the testing of objects.

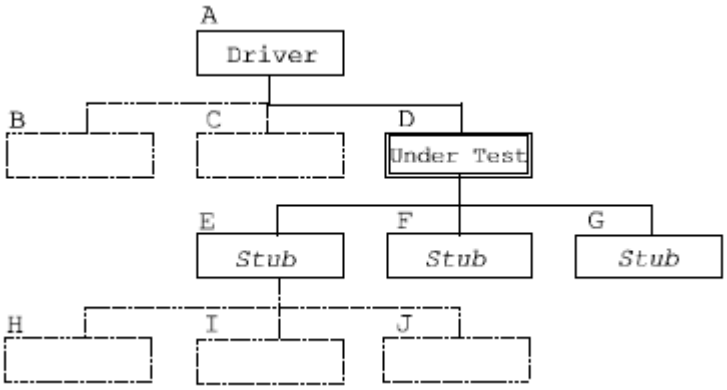
9.7.2.2 Limitations

As testing progresses up the unit hierarchy, bottom up unit testing becomes more complicated, and consequently more expensive to develop and maintain. As testing progresses up the unit hierarchy, it also becomes more difficult to achieve good structural coverage.

Changes to a unit often impact the testing of units above it in the hierarchy. For example, consider a change to unit H. Obviously, the unit test for unit H would have to change and be repeated. In addition, unit tests for units A, D and E, which use the tested unit H, would also have to be repeated. These tests may also have to change themselves, as a consequence of the change to unit H, even though units A, D and E had not actually changed. This leads to a high cost associated with retesting when changes are made, and a high maintenance and overall lifecycle cost.

The sequence in which units can be tested is constrained by the hierarchy of units, with higher units having to wait for lower units to be tested, forcing a 'long and thin' unit test phase. The first units to be tested are the last units to be designed, so unit testing cannot overlap with the detailed design phase of the software lifecycle. The relationships between units in the example program in figure 2.2 is much simpler than would be encountered in a real program, where units could be referenced from more than one other unit in the hierarchy. As for top down unit testing, the disadvantages of a bottom up approach to unit testing are compounded by a unit being referenced from more than one other unit.

9.7.3 Isolation Approach to Unit Testing



Isolation testing tests each unit in isolation from the units which call it and the units it calls. Units can be tested in any sequence, because no unit test requires any other unit to have been tested. Each unit test requires a test driver and all called units are replaced by stubs. The below figure illustrates the test driver and tested stubs needed to test unit D.

A unit test plan for the program shown in above figure, using a strategy based on the isolation organizational approach, need contain only one step, as follows:

Step (1)

(Note that there is only one step to the test plan. The sequence of tests is unimportant; all tests could be executed in parallel.)

Test unit A, using a driver to start the test and stubs in place of units B, C and D;

Test unit B, using a driver to call it in place of unit A;

Test unit C, using a driver to call it in place of unit A;

Test unit D, using a driver to call it in place of unit A and stubs in place of units E, F and G,

(Shown in figure of bottom up approach);

Test unit E, using a driver to call it in place of unit D and stubs in place of units H, I and J;

Test unit F, using a driver to call it in place of unit D;

Test unit G, using a driver to call it in place of unit D;

Test unit H, using a driver to call it in place of unit E;

Test unit I, using a driver to call it in place of unit E;

Test unit J, using a driver to call it in place of unit E.

9.7.3.1 Advantages

It is easier to test an isolated unit thoroughly, where the unit test is removed from the complexity of other units. Isolation testing is the easiest way to achieve good structural coverage, and the difficulty of achieving good structural coverage does not vary with the position of a unit in the unit hierarchy. Because only one unit is being tested at a time, the test drivers tend to be simpler than for bottom up testing, while the stubs tend to be simpler than for top down testing.

With an isolation approach to unit testing, there are no dependencies between the unit tests, so the unit test phase can overlap the detailed design and code phases of the software lifecycle. Any number of units can be

tested in parallel, to give a 'short and fat' unit test phase. This is a useful way of using an increase in team size to shorten the overall time of a software development. A further advantage of the removal of interdependency between unit tests is that changes to a unit only require changes to the unit test for that unit, with no impact on other unit tests. This results in a lower cost than the bottom up or top down organizational approaches, especially when changes are made.

An isolation approach provides a distinct separation of unit testing from integration testing, allowing developers to focus on unit testing during the unit test phase of the software lifecycle, and on integration testing during the integration phase of the software lifecycle. Isolation testing is the only pure approach to unit testing, both top down testing and bottom up testing result in a hybrid of the unit test and integration phases. Unlike the top down and bottom up approaches, the isolation approach to unit testing is not affected by a unit being referenced from more than one other unit.

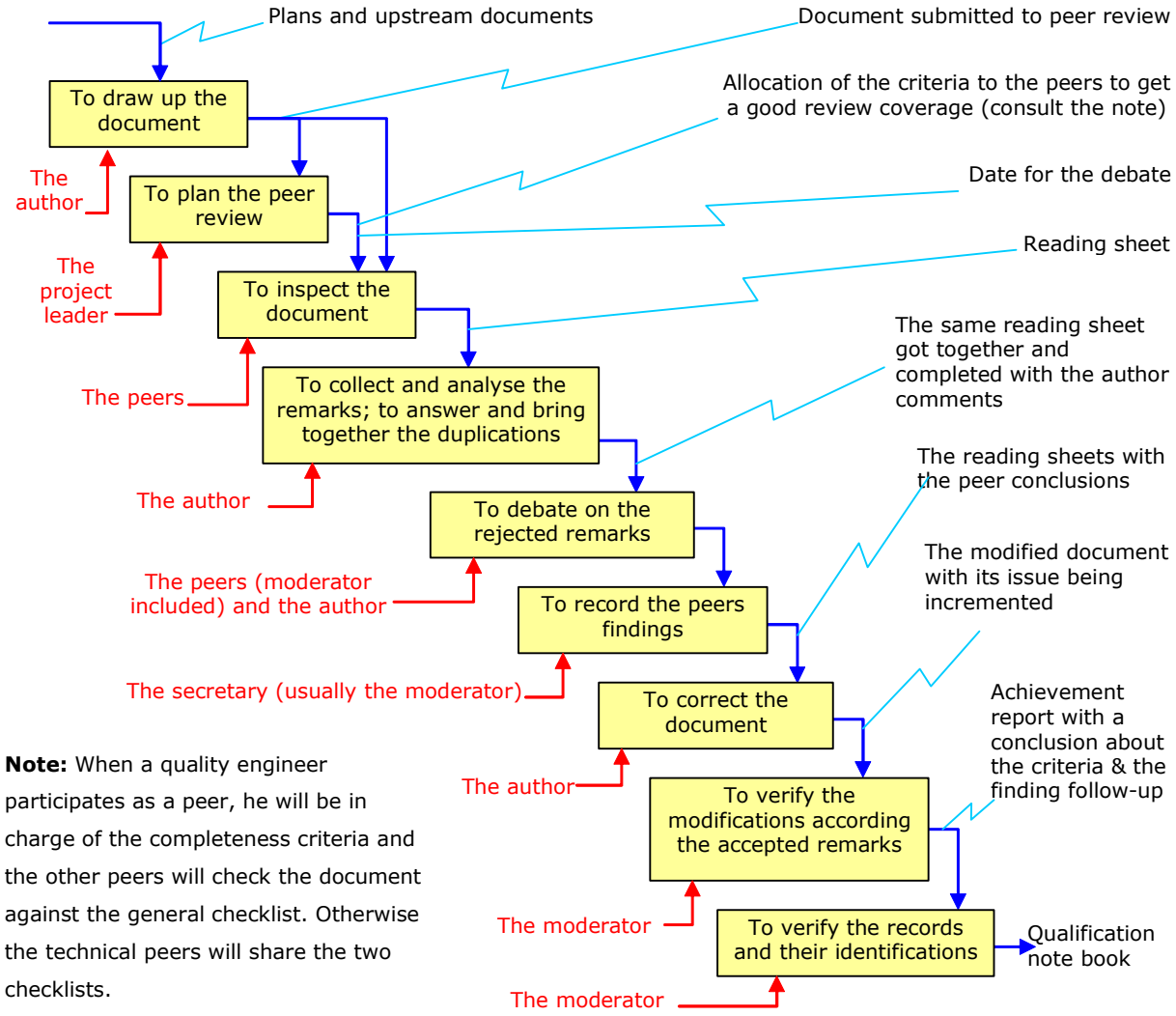
9.7.3.2 Limitations

The main disadvantage of an isolation approach to unit testing is that it does not provide any early integration of units. Integration has to wait for the integration phase of the software lifecycle. (Is this really a disadvantage?). An isolation approach to unit testing requires structural design information and the use of both stubs and drivers. This can lead to higher costs than bottom up testing for units near the bottom of the unit hierarchy. However, this will be compensated by simplified testing for units higher in the unit hierarchy, together with lower costs each time a unit is changed.

10.0 Inspection of the outputs of Software Development Life Cycle (SDLC)

10.1 Inspection Process

The inspection process lifecycle should be defined prior to development process and each output of SDLC will undergo this process. This inspection process lifecycle can be defined using electronic change management software. A detailed inspection process life cycle is shown below:



Note: When a quality engineer participates as a peer, he will be in charge of the completeness criteria and the other peers will check the document against the general checklist. Otherwise the technical peers will share the two checklists.

Each of the activities of the above figure is detailed in the following table:

Activity	Inputs	Outputs	Description / comment
1. Draw up the document	Plan and upstream documents	Peer review dossier	The main objective of the author is to help the peer. So he will make up a dossier as complete as possible. This dossier will include the applicable issues of the upstream documents, the documents under review, a pre-filled review-form, the known problems on the document (e.g. uncovered requirements or uncorrected problem reports). The author shall check himself the automated verifications, such as spelling and traceability matrix establishment. Note 1: The peers will review the traceability foundations and results Note 2: The documents include the source code and the test data.
2. Plan the peer review	Availability date	Allocation of the criteria to the peers	Some review may be hold at one third of the phase to assess the direction. In this case a second review shall be organized at the document ending. The author should declare himself the availability of the document. If the author says that the document is uncompleted and it is not ready to be profitably reviewed, it is probably right. The software project manager will select the moderator and the peers. A reading duration objective should be estimated in the project plans and communicated to the peers. Some of the criteria may be allocated to one of the peers (such as the standard conformity check, for example). When the quality engineer participates as a peer, he will be in charge of the completeness criteria and the other peers will check the document against the general checklist. Otherwise the technical peers will share the two checklists. A meeting could be hold to present the review dossier, the peer review process and the criteria allocation. This meeting is optional.
3. Inspect the document	Peer review dossier Allocation of the criteria to the peers	Individual reading sheets	Each peer will fill up his reading form. Take care to completely fill the referenced document boxes, including upstream documents He must fill the criteria table, in order to check whether the coverage of all the criteria added cover the DO-178B objectives. The functional analysis and the coverage analysis are performed during this inspection. The rational (if any) will be included in a specific section of the reading sheets. The conclusion will be included in the criteria table. The reading sheet shall be returned to the author one or two days before the debate meeting.

Activity	Inputs	Outputs	Description / comment
4. Collect and analyze the remarks; answer and solve the duplications	Individual reading sheets	Merged reading sheets	The author will define the set of remarks that will be debated during the meeting. The moderator will supervise this selection if three or more peers take part to the review. The author will accept some remarks; so there is no need for a debate about those accepted ones. The points that need explanations, the major remarks and the remarks that the author would like to reject will be debated during the meeting. The author will merge the individual reading sheet in a single document that will support the meeting. In this document the remark should be prioritized; the more important the first. The author's answers will be recorded in the same document that will support the debate.
5. Debate on the rejected remarks	Merged reading sheets	Peer findings	The moderator leads the debate (not the author). The moderator is responsible for the understanding of all the participants, and for that all the major points are debated. The conclusion of the debate between the peers and the author should be aligned on the most critical peer. It means that the document should be modified if only one of the peers state so. The debate meeting may convince this peer that a modification is unnecessary, but this decision cannot be a majority voting and cannot be the author's decision. This basic rule is taken order to improve the general quality
6. Records the peer findings	Peer findings	Peer review report Problem reports	A secretary will record the conclusion. He cannot be the author: He may be the moderator but a third people would be better. A pleasant way to get an agreement upon the finding wording is to show the peer review report with a video equipment during the debate. The problem identified in upstream documents will be recorded in Problem reports.
7. Correct the document	Peer review report	Modified document	The objective of the peers is to identify the problems. The way to correct the problems is left to the author, and when the project plans are impacted, to the software project manager.
8. Check the modifications according the accepted remarks	Peer review report Modified document	Achievement report	The moderator will check that all the actions defined in the Peer review report are correctly handled. The moderator will record his conclusions in an achievement report, with the modified document as reference (and its issue incremented). This report will include a conclusion about the peer finding follow-up and the DO-178B criteria compliance.

Activity	Inputs	Outputs	Description / comment
9. Check the records and their identifications	Peer review report Modified document Achievement report	Verification note book	The moderator will check whether the peer review outputs are correctly archived. The moderator will add a reference to the Achievement report in the Verification notebook.

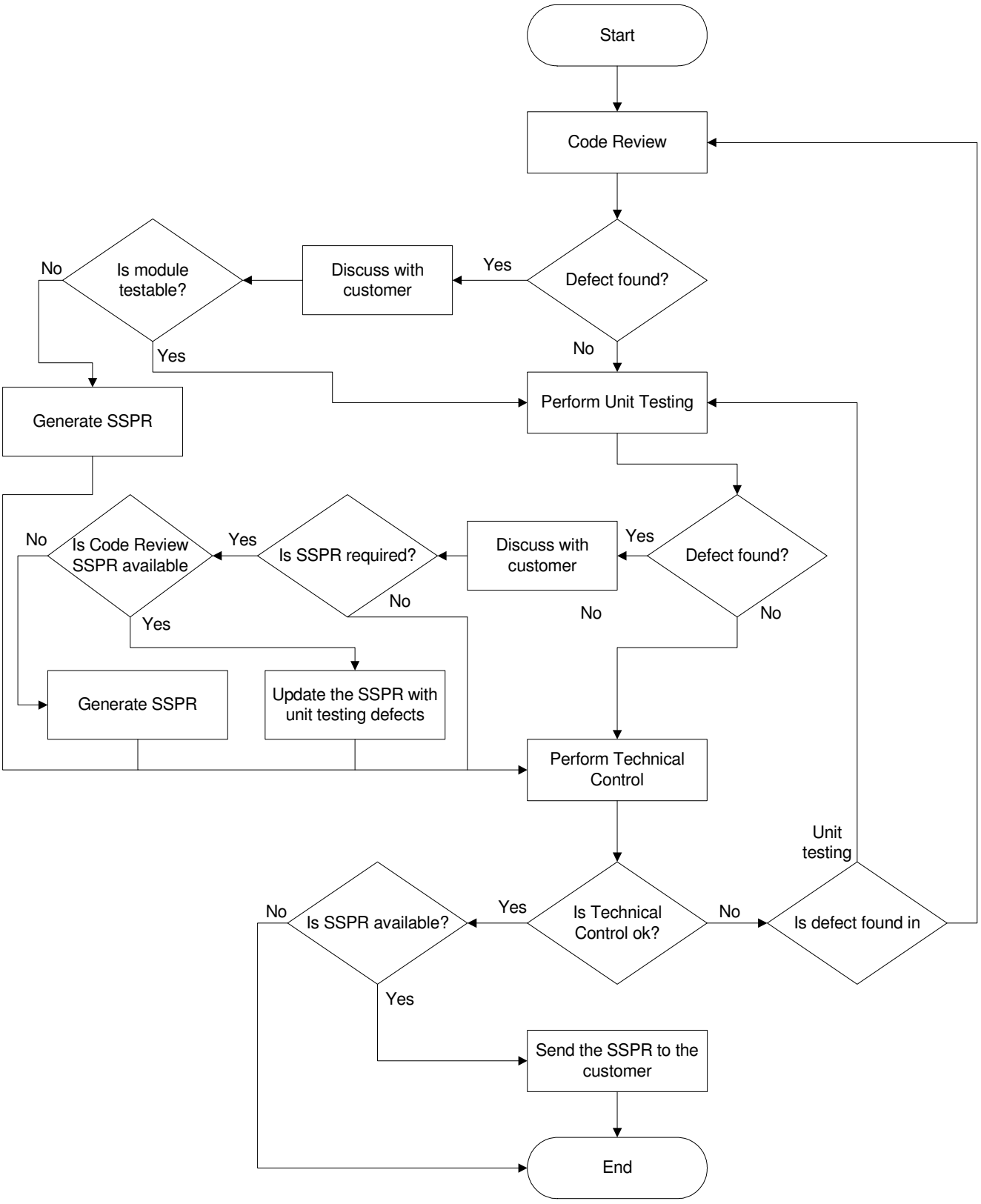
10.2 Inspection Process Guidelines

No.	DO-178B Objectives		Inspection Guidelines
	Description	Ref.	
1.	Test procedures are correct	6.3.6.b	The test cases/procedure exists and is written in accordance with test guidelines. The review should check that all low-level requirements are exercised by the test case(s) and the test procedure implements the test cases. No test case should be written without any associated low level requirement No test procedure should be written without any associated test case
2.	Tests results are correct and discrepancies explained	6.3.6.c	The tests results are obtained. Any discrepancies, if applicable are detailed in problem reports. The test execution should use the required version of the tested source code and low-level requirement. Additionally the tool coverage and build options should be verified. For level A, source to object traceability should be done
3.	Test coverage of software structure is achieved (modified condition/decision)	6.4.4.2	Structural coverage report exists to depicts 100% MC/DC coverage. Note that the coverage should be 100% structural coverage (as per DO-178B level) by combining the coverage attained from the tool supported by static coverage analysis or justification if required.
4.	Test coverage of software structure (decision coverage) is achieved	6.4.4.2.a 6.4.4.2.b	Structural coverage report exists to depicts 100% decision coverage.

No.	DO-178B Objectives		Inspection Guidelines
	Description	Ref.	
5.	Test coverage of software structure (statement coverage) is achieved	6.4.4.2.a 6.4.4.2.b	Structural coverage report exists to depict 100% statement coverage.
6.	Test coverage of software structure (data coupling and control coupling) is achieved	6.4.4.2.c	Data Coupling: The test coverage of the data flow at low level requirements is attained by associating the data from data dictionary which includes all global data (linker map may also be used), to the low level requirement based test case. Control Coupling: The calling tree, as documented in Software Design Description, is analyzed to identify all of the functions in the software and is reviewed using software design review checklist. Review of any unused function than one listed in the calling tree is available. Also, the structural coverage report generated by low level requirement based test execution will be analyzed to check that the entire calling trees have been executed. The data and control coupling analysis report is included as part of software verification results for completeness

10.3 Problem Reporting Mechanism in Unit Testing

During the unit testing process a tester can confront with different types of problems. A predefined problem reporting has to be defined before the start of this campaign. A detailed problem reporting mechanism is as shown in below diagram.


Start
Code Review
Defect found?
Yes
Discuss with customer
No
Is module testable?
No
Yes
Generate SSPR
Perform Unit Testing
Defect found?
Yes
Discuss with customer
No
Is SSPR required?
Yes
No
Is Code Review SSPR available
No
Yes
Generate SSPR
Update the SSPR with unit testing defects
Perform Technical Control
Unit testing
Is defect found in
No
Is Technical Control ok?
Yes
Is SSPR available?
No
Yes
Send the SSPR to the customer
End

11.0 MISCELLEANEOUS TOPICS

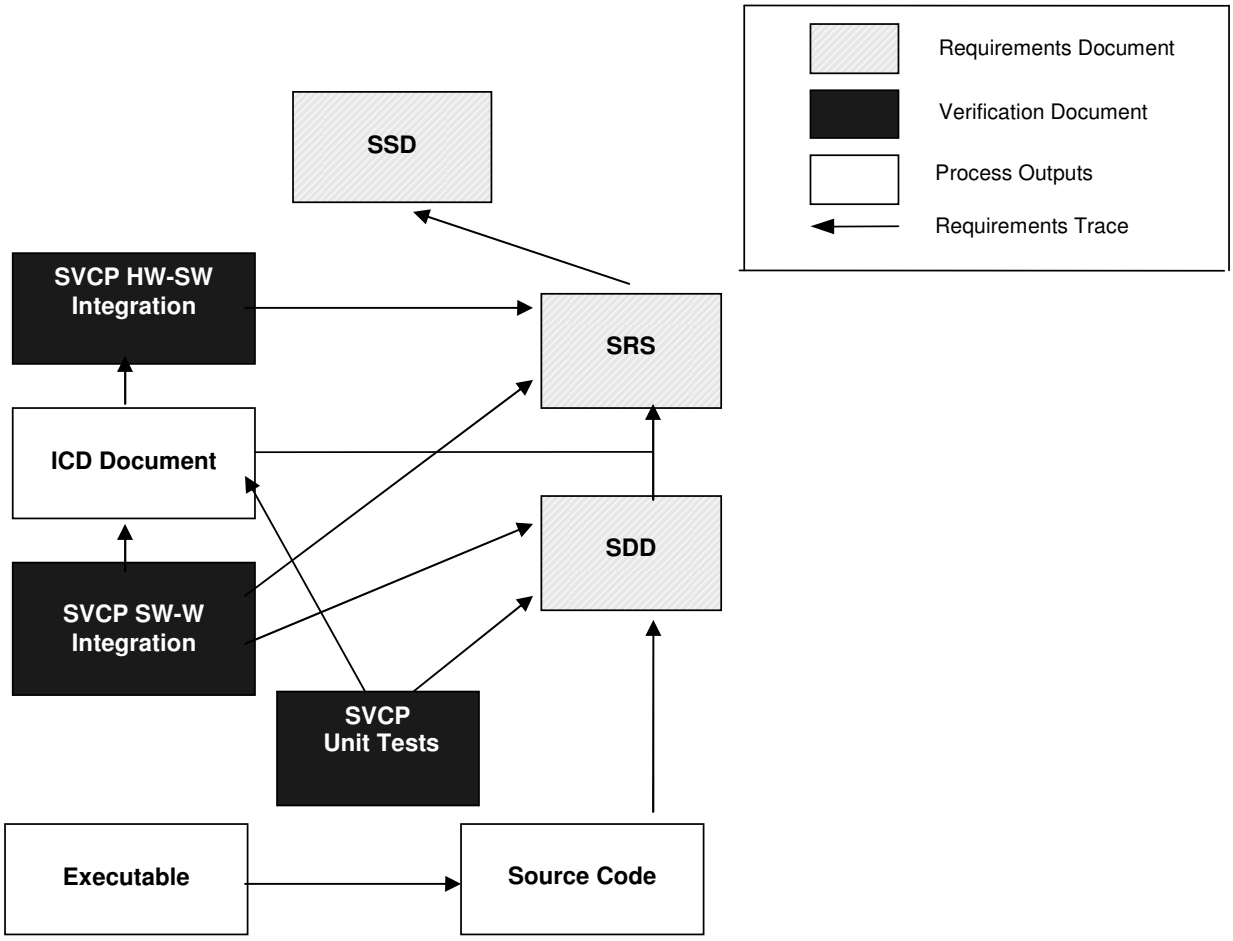
11.1 REQUIREMENT TRACEABILITY

To demonstrate that the requirements have been satisfied, a trace matrix should be used to demonstrate traceability between system requirements and software design data. Traceability will be verified in both directions.

The following documents will be referenced by the trace matrix to provide test coverage traceability:

- Software Verification Cases & Procedures (SVCP), Hardware (HW)/Software (SW) Integration,
- Software Verification Cases & Procedures (SVCP), Software Integration,
- Software Verification Cases & Procedures (SVCP), Module Test,
- Software Verification Cases & Procedures (SVCP), Module Test.

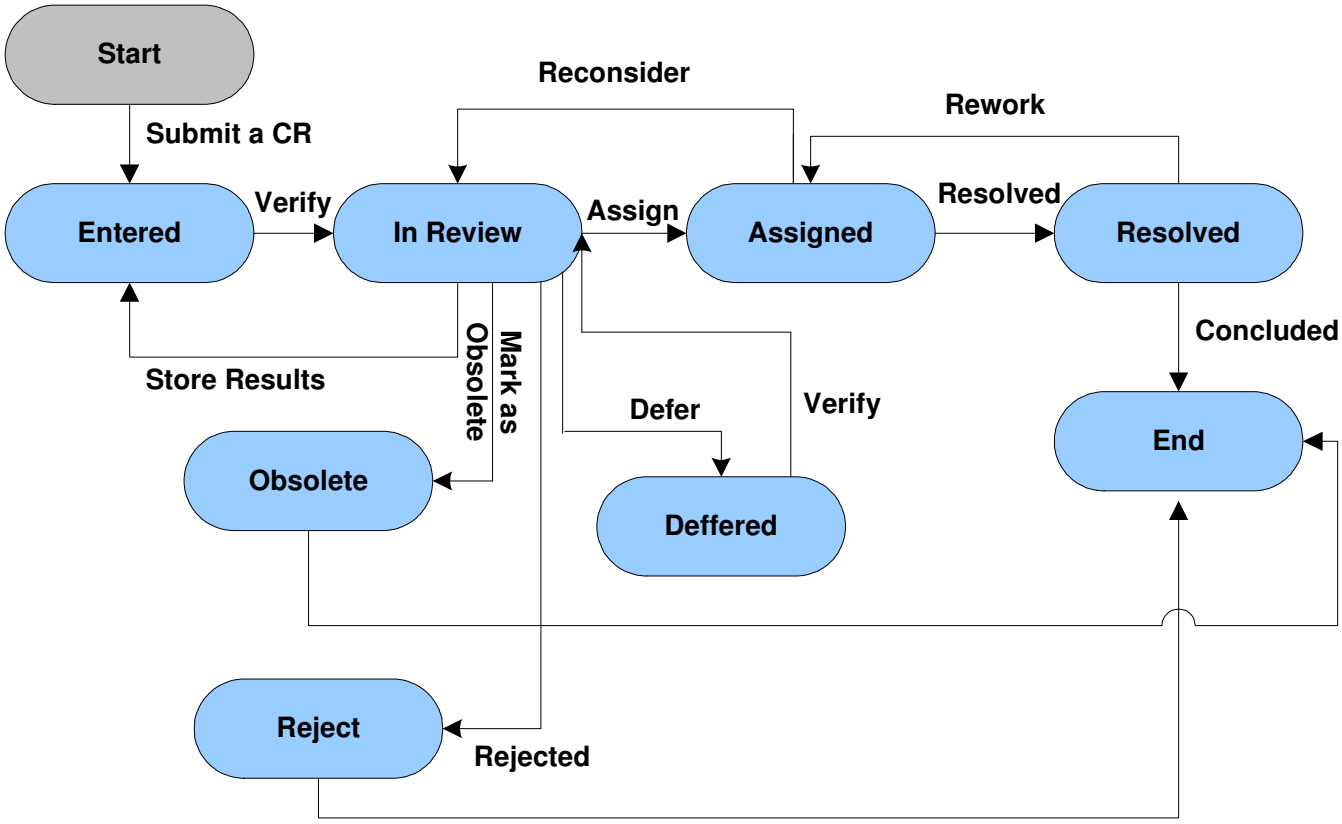
Figure 6.2.1-1 Details the requirements trace flow between the relevant documents.



11.2 SOFTWARE CHANGE REQUEST LIFE CYCLE

The Software Change Request [SCR] process lifecycle should be defined prior to launching the Software Verification Process [SVP]. The software change request process lifecycle is usually defined using a finite state machine diagram. The finite state machine diagram shall consist of states and state transitions.

The software change request process lifecycle can be defined using electronic change management software. A generic software change request process lifecycle is shown below:



11.3 ASSEMBLY TESTING

- Perform review if the assembly sources are implicitly or, explicitly covered by high/low level tests for requirement coverage. If yes, then perform hand analysis on the structural coverage measure per test case flow in case no tool exists to automatically produce the structural coverage
- Add additional tests if the requirements pertaining to assembly source are not covered by high/low level tests
- Add additional test cases if there is a coverage gap and associate with existing/new requirement
- If no test are available, perform an low-level tests as per the low-level requirement, structural coverage measure analysis can be performed manually

Alternatively, the coverage measure could be aided using debug whilst execution informally.

Sample manual coverage analysis format:

Line #	Test Case #	Test Case #	Test Case #	Test Case #	Test Case #	ASM Code Extract
1	NA	NA	NA	NA	NA	; comment
2	NA	NA	NA	NA	NA	; comment
3	NA	NA	NA	NA	NA	; comment
4	NA	NA	NA	NA	NA	; comment
5	1	2	3	4		assembly statement 1
6	1	2	3	4		assembly statement 2
7		2				assembly decision 1
8	1	2	3	4		assembly statement 3
9			3			assembly decision 2
10			3			assembly statement 4

Conclusion :

All lines are covered [Yes/No]:

Decision Coverage Percentage: ___ (Covered Count / Total Count)

Statement Coverage Percentage: ___ (Covered Count / Total Count)

Following decision(s) are not covered:

Following statement(s) are not covered:

Note: In assembly, MC/DC cannot be applicable, as in the assembly source code there cannot be more than two operands. Hence only decision and/or statement coverage analysis should be done.

11.4 SOURCE TO OBJECT ANALYSIS (FOR LEVEL A ONLY)

Objective:

- To verify the functional correctness of the generated assembly code to source code.
- There is no additional functionality that is generated in the assembly code with the pre-defined set of compiler options.

Usually two approaches are used for source to object analysis. The agreement of the approach should be documented in the plan.

Approach 1: Make the representative code of the project, assemble and list, and manually review line to line of high level language (example C) and the assembled/listed code for functional correctness.

Approach 2: Use actual project source code instead of representative code, remaining same as approach 1

Hence the approach should be agreed prior commencing this activity. In most of the project, representative code is acceptable.

The detail of the approach 1 is stated below. Same can be used for approach 2, but with only difference of actual code instead of representative code:

- Constructs and Keyword to the specific language should be defined in plan or, standard
- Generate the representative code that covers the various combination of keyword/construct usage
- Substantiate the representation of code, with the actual code on keyword/construct

Example:

C Construct/ Keyword	Description	Used in Project, Yes/No	Project Source Code File Name	Representative C File
-	subtraction operator	Yes	File1.c	C-TST1.c
--	decrement value by one	Yes		
!	not operation	Yes		
!=	not equal relational operator	Yes		
%	modulus operator	Yes		
&	address operator	Yes		
&&	logical AND operator	Yes		
&=	logical AND assignment operator.	Yes		
*	multiplication operator (or pointer)	Yes		
*=	multiplication assignment operator	Not used		

C Construct/ Keyword	Description	Used in Project, Yes/No	Project Source Code File Name	Representative C File
int	integer type	Yes		
interrupt	signifies the function is an interrupt routine.	Yes		
cregister	allows access to the I/O port space of the TI processor	Not Used		

- Assemble and list the representative code (or actual code as the case be), with the same compiler option as used by project
- Analyze the line-to-line high level code (ex. C language) to the assembled and listed code

Example format:

Source Code	Assembly code	Traceable to Source (Y/N)	Analysis
j=0;	MOV *-SP[2],#0	Y	Store 0 to stack location, -2 words from top of stack.
for(i=1; i<k; i++)	MOV *-SP[1],	Y	Store 16-bit AL register to stack location,-1 words from top of stack
	MOV AL,*-SP[3]		Store 16-bit AL register from stack location,-3 words from top of stack
	CMP AL,*-SP[1]		Compare contents of AL with stack,-1 word from top of stack.
	BF L2,LEQ		Branch Fast to L2 when AL is Less Then or Equal to *-SP[1]
Conclusion: The assembly code performs the intended function of the source code and is reviewed as traceable to the source code.			

11.5 APPENDIX E: DO-178B OUTPUTS OF SOFTWARE VERIFICATION PROCESS

11.5.1 Verification of Outputs of Software Requirements Process

	Objective		Applicability by SW Level				Output		Control Category by SW level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
1	Software high-level requirements comply with system requirements.	6.3.1a	●	●	○	○	Software Verification Results	11.14	②	②	②	②
2	High-level requirements are accurate and consistent.	6.3.1b	●	●	○	○	Software Verification Results	11.14	②	②	②	②
3	High-level requirements are compatible with target computer.	6.3.1c	○	○			Software Verification Results	11.14	②	②		
4	High-level requirements are verifiable.	6.3.1d	○	○	○		Software Verification Results	11.14	②	②	②	
5	High-level requirements conform to standards.	6.3.1e	○	○	○		Software Verification Results	11.14	②	②	②	
6	High-level requirements are traceable to system requirements.	6.3.1f	○	○	○	○	Software Verification Results	11.14	②	②	②	②
7	Algorithms are accurate.	6.3.1g	●	●	○		Software Verification Results	11.14	②	②	②	

LEGEND:		
	●	The objective should be satisfied with independence.
	○	The objective should be satisfied.
	Blank	Satisfaction of objective is at applicant's discretion.
	①	Data satisfies the objectives of Control Category 1 (CC1).
	②	Data satisfies the objectives of Control Category 2 (CC2).

11.5.2 Verification of Outputs of Software Design Process

	Objective		Applicability by SW Level				Output		Control Category by SW level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
1	Low-level requirements comply with high-level requirements.	6.3.2a	●	●	○		Software Verification Results	11.14	②	②	②	
2	Low-level requirements are accurate and consistent.	6.3.2b	●	●	○		Software Verification Results	11.14	②	②	②	
3	Low-level requirements are compatible with target computer.	6.3.2c	○	○			Software Verification Results	11.14	②	②		
4	Low-level requirements are verifiable.	6.3.2d	○	○			Software Verification Results	11.14	②	②		
5	Low-level requirements conform to standards.	6.3.2e	○	○	○		Software Verification Results	11.14	②	②	②	
6	Low-level requirements are traceable to high-level requirements.	6.3.2f	○	○	○		Software Verification Results	11.14	②	②	②	
7	Algorithms are accurate.	6.3.2g	●	●	○		Software Verification Results	11.14	②	②	②	
8	Software architecture is compatible with high-level requirements.	6.3.3a	●	○	○		Software Verification Results	11.14	②	②	②	
9	Software architecture is consistent.	6.3.2b	●	○	○		Software Verification Results	11.14	②	②	②	
10	Software architecture is compatible with target computer.	6.3.3c	○	○			Software Verification Results	11.14	②	②		
11	Software architecture is verifiable.	6.3.3d	○	○			Software Verification Results	11.14	②	②		
12	Software architecture conforms to standards.	6.3.3e	○	○	○		Software Verification Results	11.14	②	②	②	
13	Software partitioning integrity is confirmed.	6.3.3f	●	○	○	○	Software Verification Results	11.14	②	②	②	②

LEGEND:		
	●	The objective should be satisfied with independence.
	○	The objective should be satisfied.
	Blank	Satisfaction of objective is at applicant's discretion.
	①	Data satisfies the objectives of Control Category 1 (CC1).
	②	Data satisfies the objectives of Control Category 2 (CC2).

11.5.3 Verification of Outputs of Software Coding & Integration Process

	Objective		Applicability by SW Level				Output		Control Category by SW level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
1	Source Code complies with low-level requirements.	6.3.4a	●	●	○		Software Verification Results	11.14	②	②	②	
2	Source Code complies with software architecture.	6.3.4b	●	○	○		Software Verification Results	11.14	②	②	②	
3	Source Code is verifiable.	6.3.4c	○	○			Software Verification Results	11.14	②	②		
4	Source Code conforms to standards.	6.3.4d	○	○	○		Software Verification Results	11.14	②	②	②	
5	Source Code is traceable to low-level requirements.	6.3.4e	○	○	○		Software Verification Results	11.14	②	②	②	
6	Source Code is accurate and consistent.	6.3.4f	●	○	○		Software Verification Results	11.14	②	②	②	
7	Output of software integration process is complete and correct.	6.3.5	○	○	○		Software Verification Results	11.14	②	②	②	

LEGEND:	●	The objective should be satisfied with independence.
	○	The objective should be satisfied.
	Blank	Satisfaction of objective is at applicant's discretion.
	①	Data satisfies the objectives of Control Category 1 (CC1).
	②	Data satisfies the objectives of Control Category 2 (CC2).

11.5.4 Verification of Outputs of Integration Process

	Objective		Applicability by SW Level				Output		Control Category by SW level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
1	Executable Object Code complies with high-level requirements.	6.4.2.1 6.4.3	○	○	○	○	Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	② ②
2	Executable Object Code is robust with high-level requirements.	6.4.2.2 6.4.3	○	○	○	○	Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	② ②
3	Executable Object Code complies with low-level requirements.	6.4.2.1 6.4.3	●	●	○		Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	
4	Executable Object Code is robust with low-level requirements.	6.4.2.2 6.4.3	●	○	○		Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	
5	Executable Object Code is compatible with target computer.	6.4.3a	○	○	○	○	Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	② ②

LEGEND:		
	●	The objective should be satisfied with independence.
	○	The objective should be satisfied.
	Blank	Satisfaction of objective is at applicant's discretion.
	①	Data satisfies the objectives of Control Category 1 (CC1).
	②	Data satisfies the objectives of Control Category 2 (CC2).

11.5.5 Verification of Outputs of Verification Process

	Objective		Applicability by SW Level				Output		Control Category by SW level			
	Description	Ref.	A	B	C	D	Description	Ref.	A	B	C	D
1	Test procedures are correct.	6.3.6b	●	○	○		Software Verification Cases and Procedures	11.13	②	②	②	
2	Test results are correct and discrepancies explained.	6.3.6c	●	○	○		Software Verification Results	11.14	②	②	②	
3	Test coverage of high-level requirements is achieved.	6.4.4.1	●	○	○	○	Software Verification Results	11.14	②	②	②	②
4	Test coverage of low-level requirements is achieved.	6.4.4.1	●	○	○		Software Verification Results	11.14	②	②	②	
5	Test coverage of software structure (modified condition/decision) is achieved.	6.4.4.2	●				Software Verification Results	11.14	②			
6	Test coverage of software structure (decision coverage) is achieved.	6.4.4.2a 6.4.4.2b	●	●			Software Verification Results	11.14	②	②		
7	Test coverage of software structure (statement coverage) is achieved.	6.4.4.2a 6.4.4.2b	●	●	○		Software Verification Results	11.14	②	②	②	
8	Test coverage of software structure (data coupling and control coupling) is achieved.	6.4.4.2c	●	●	○		Software Verification Results	11.14	②	②	②	

LEGEND:		
	●	The objective should be satisfied with independence.
	○	The objective should be satisfied.
	Blank	Satisfaction of objective is at applicant's discretion.
	①	Data satisfies the objectives of Control Category 1 (CC1).
	②	Data satisfies the objectives of Control Category 2 (CC2).

12.0 DO-178B SW CERTIFICATION

This chapter presents the understanding on the DO-178B certification audit for the task(s) carried out to meet the DO-178B objectives fully or partially, as specified in the project plans. The chapter describes the purpose, procedure, typical problems founds, and suggested mitigation during the certification audits. The chapter workflow is organized as:

1. Purpose and Types of Audit
2. Typical entry criteria to start the audit
3. Typical Audit Agenda
4. Typical Audit Execution
5. Common issues raised during the audit and proposed mitigation
6. Preparation for Certification Audit

12.1 Certification Audit

12.1.1 Background

1. RTCA/DO-178B, Section 9.2

Certification authority reviews may take place at the applicant's facilities or the applicant's suppliers' facilities.

2. RTCA/DO-178B, Section 10.3

The certification authority may review at its discretion the software life cycle processes and their outputs during the software life cycle as discussed in subsection 9.2

3. Order 8110.49

- Provides guidelines for performing software reviews
- Documents the review approach, which is detailed in FAA Job Aid "Conducting Software Reviews Prior to Certification
- When FAA should be involved
- To what extent and areas FAA should be involved (LOFI – Level of FAA Involvement)

12.1.2 Purpose of the Review

The purpose of the software review is to assess whether or not the software developed for a project complies with the applicable objectives of RTCA/DO-178B.

More specifically:

- Address issues in a timely manner
- Physically examine compliance data
- Verify adherence to plans and procedures

Terms and definition:

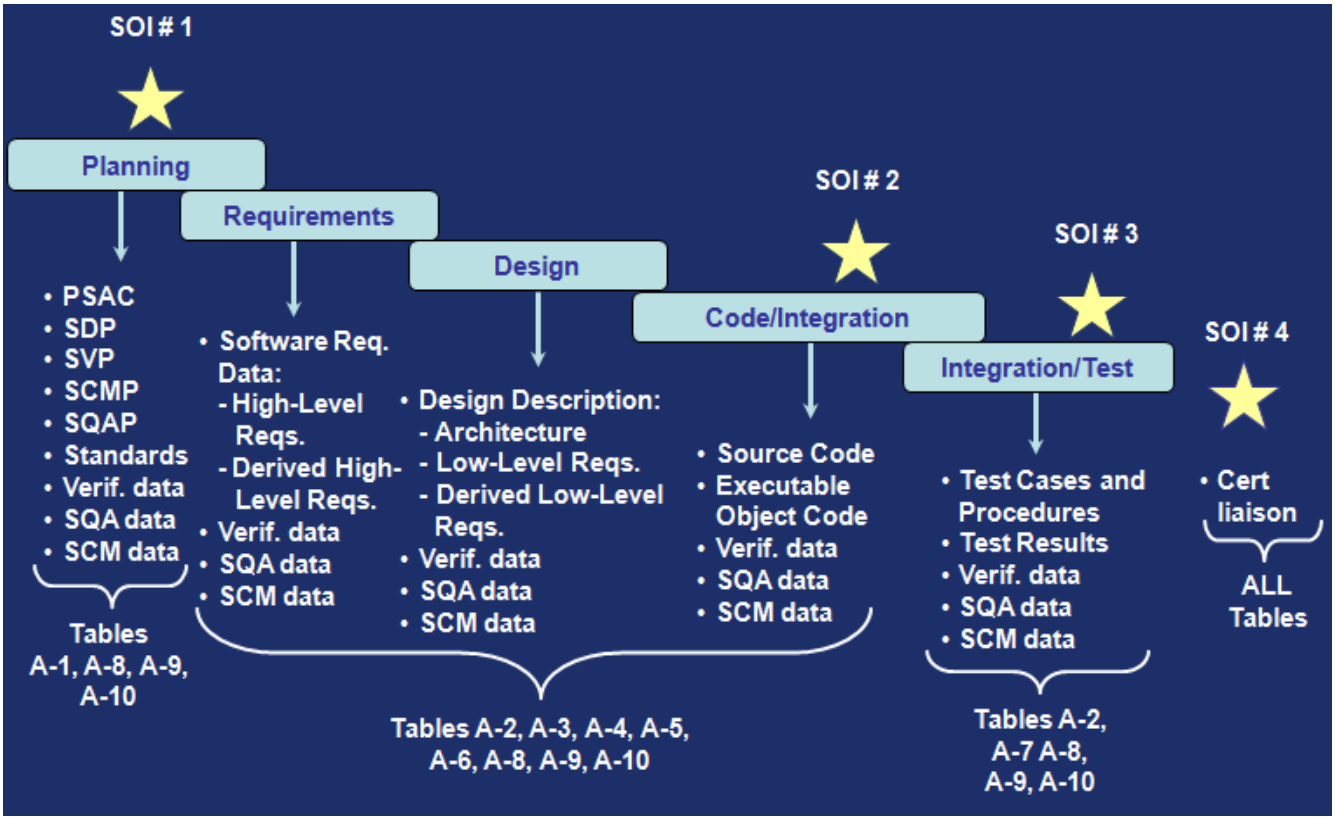
- Compliance is the satisfaction of a DO-178B objective.
- A finding is the identification of a failure to show compliance to one or more of the RTCA/DO-178B objectives.
- An observation is the identification of a potential software life cycle process improvement. A observation is not a RTCA/DO-178B compliance issue and does not need to be addressed before software approval.
- An action is an assignment to an organization or person with a date for completion to correct a finding, error, or deficiency identified when conducting a software review.

12.1.3 Types of Review

There are four types of software certification reviews:

Planning	Stage of Involvement (SOI) # 1
Development	Stage of Involvement (SOI) # 2
Verification	Stage of Involvement (SOI) # 3
Final Certification	Stage of Involvement (SOI) # 4

12.1.4 Stages of Involvement & DO-178B Objectives



12.1.5 SOI Readiness Criteria

SOI	Description	Readiness Criteria	Data Required	DO-178B Annex A Table
1	Determine if plans and standards provide an acceptable means for satisfying objectives of RTCA/DO-178B	Initial software planning process is complete: • Plans and standards have been internally reviewed • Plans and standards have been reviewed by SQA • Plans and standards are approved and under configuration control	• PSAC • SDP • SVP • SCMP • SQAP • Standards • Tool Qual. Plans • Verification Data • SQA data • SCM data	A-1, A-8, A-9, A-10
2	Determine if software development is in accordance with approved plans and standards	Conducted when at least 50% of development data is complete: • High-level requirements are documented, reviewed, and traceable to system requirements • Software architecture is defined and reviews and analyses are complete • Low-level requirements are documented, reviewed, and traceable to high-level requirements • Source code implements low-level requirements, is traceable to low-level requirements, and has been reviewed	• Standards for Software Development • Software Requirements Data • Design Description • Source Code • Software Verification Results • Problem Reports • Software Configuration Management Records • Software Quality Assurance Records	A-2, A-3, A-4, A-5, A-8, A-9, A-10

3	• Determine if software verification is in accordance with approved plans
	• Ensure requirements, design, code and integration are appropriately verified
	• Ensure verification process will achieve:
	  ○ Requirements based test coverage
	  ○ Appropriate level of structural coverage
		Conducted when at least 50% of verification and testing data is complete:
		• Development data is complete, reviewed, and under configuration control
		• Test cases and procedures are documented, reviewed, and under configuration control
		• Test cases and procedures have been executed (formally or informally)
		• Test results are documented
		• Testing environment is documented and controlled
			• Software Requirements Data
			• Design Description
			• Source Code
			• Software Verification Cases and Procedures
			• Software Verification Results
			• Problem Reports
			• Software Configuration Management Records
			• Software Quality Assurance Records
				A-2,
				A-6,
				A-7,
				A-8,
				A-9,
				A-10

4	• Determine compliance of final product with objectives of RTCA/DO-178B • Verify that all software related problem reports, action items, and certification issues have been addressed	Conducted when final software build is complete and ready for formal system certification approval: • Software Conformity Review has been conducted • Software Accomplishment Summary and Software Configuration Index are complete • All other software life cycle data are complete and under configuration control	• Software Conformity Records • Software Life Cycle Environment Configuration Index • Software Configuration Index • Problem Reports • Software Accomplishment Summary	All tables

12.1.6 Typical Audit Agenda

Following is the typical SOI audit agenda:

1. Introduction of all participants
2. Presentation and Reviews based on SOI

Stage of Involvement#1

a. Presentation having content at the minimum as:

- System Overview
- Development, Verification approach:
- Software architecture draft, as applicable
- Organization structure with development/verification team, meeting independence, QA organization
- SCM organization, problem reporting workflow
- CM Version of CC1 items under review

b. Closure Review of offline planning reviews remarks

Stage of Involvement#2

a. Presentation having content at the minimum as:

- System Overview
- Overview on the Process change
- SOI1 audit recap, closure summary
- CM Version of CC1 items under review including system and customer baseline as applicable
- Status snapshot on development

b. Review of development artifacts on the requirements, design, code including traceability, review artifacts and problem report

Stage of Involvement#3

a. Presentation having content at the minimum as:

- System Overview
- Overview on the Process change
- SOI2 audit recap, closure summary

- CM Version of CC1 items under review including system and customer baseline as applicable
- Status snapshot on development and verification

b. Review of development and verification artifacts on the requirements, design, code, test, result, structural coverage including traceability, review artifacts and problem report

c. Demonstration of test setup, and random test execution

Stage of Involvement#4

a. Presentation having content at the minimum as:

- SOI3 audit recap, closure summary
- CM Version of CC1 items under review (via SCI) including system and customer baseline as applicable
- Status snapshot on development and verification

b. Review of Formal Test Execution Results and Review records

c. Review on offline comment on the accomplishment summary

d. Review of archieve on the CM system

3. Debrief of the audit

Typically the audit report is shared by the certification responsible containing:

- List of attendees' with signature
- Item list with their CM version
- List of Findings, Actions, Observation

12.1.7 Typical Audit Execution

Following is an example of a typical SOI3 review after the presentation (or overview) of the project is done by the auditee.

1. Assessment of SOI2 closure
2. Assessment of change of plans / checklist if any.

   Focus area:

   - Changes in the checklists should be reflected in the review records

- Process changes during the planning revisions, and thereby its impact on applicable artifacts

3. Conduct slice review:

   This involves the review of randomly picked trace of high level requirement and its thread to design, code and test

   Some of the points considered for picking the slice:

   - Arithmetic (or equation based) requirement to follow up the test on robustness & usage of Data Dictionary (for range checks)
   - Requirement containing conditions to follow up the test sufficiency for MCDC

   Focus area:

   - Items under review are configuration controlled
     - Review records of the requirements, design, code, test case, procedure and result
     - Completeness of review checklist, each point being addressed as Yes, No or N/A. Rational provided for filling the checklist as "Yes", "N/A". For every "No", action being raised and tracked to closure
     - Requirement coverage to the test case is complete, and if it is partially covered it is documented
     - Structural Coverage obtained and its analysis
     - Independence criteria as per the plan

4. Review the PR tracking mechanism

   Focus area:

   - Completeness of PR field as per the plan
   - CCB records
   - Review of resolution and verification on the version stated in Problem Report

5. Witness few test case execution for different levels of test and tool set

   Focus area:

   - Is the setup instruction sufficient to reproduce the test environment and items required for test execution (source code, executable, test script)?
   - Can the randomly selected test be executed?

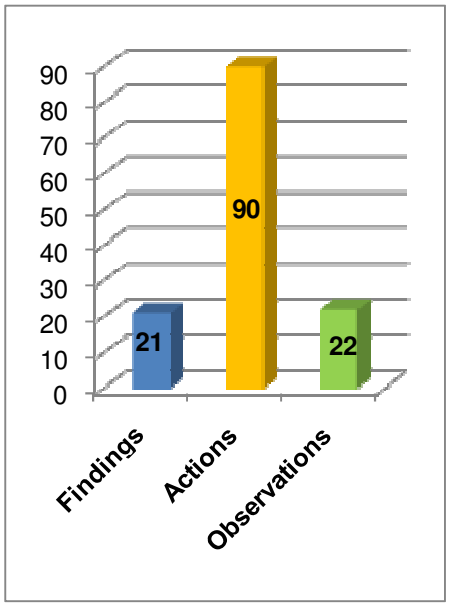
6. Tool Qualification
7. Audit the Quality team, to see the evidence as per SQAP is available

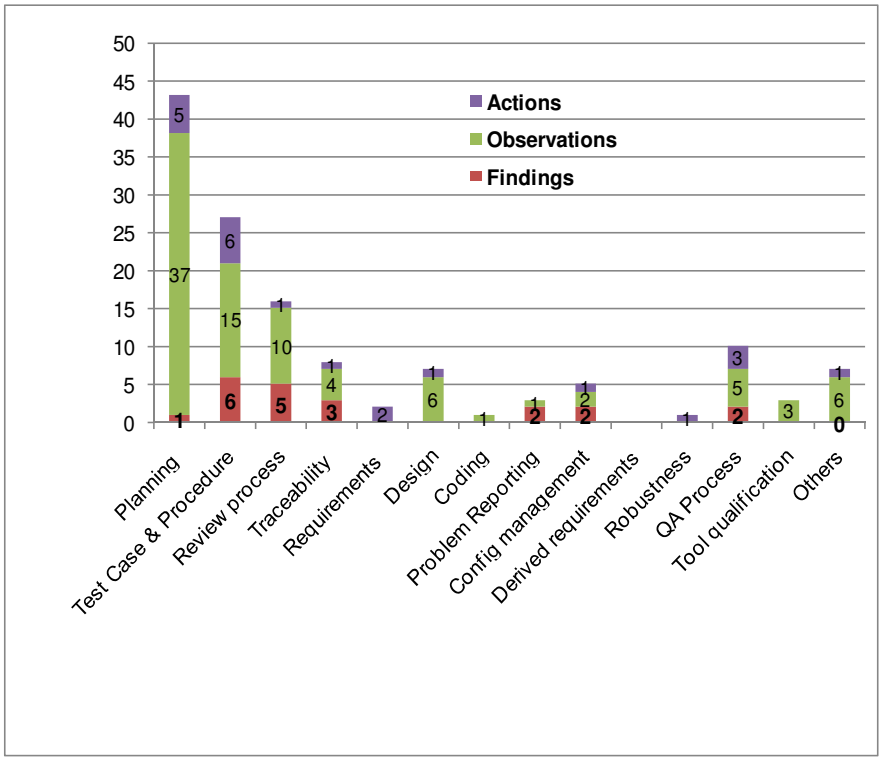
12.2 Lessons Learnt & Common Issues

12.2.1 Analysis of the outcome of SOI audits

Most of the issues are found in two stages of Involvement Audits that are SOI-2 and SOI-3 during DO-178B & DO-254 certification audits.

SOURCE: SOI audits on 10+ projects during 2008-09. The number includes the allocation, both to a reputed Indian MNC in aerospace domain and its customer.





12.2.2 Issues & Mitigation Found during the Audit

Following are the issues noted based on the certification reviews & the proposed mitigation.

A. Planning Phase

COMMON ISSUES & CONCERN	CORRECTIVE ACTION / SUGGESTED MITIGATION
Inconsistencies or, contradictory statements between the planning documents	Make DO-178B Planning Compliance Matrix for each plans, PSAC, SDP, SVP, CMP, QAP & Standards Avoiding duplicate information in multiple plan Perform walkthrough of the planning artifacts
Planning document has additional details that are not relevant for the specific plan. Example – Detailing the verification strategy, environment etc in Software Development Plan	
Lack in the identification of need of Tool Qualification	

COMMON ISSUES & CONCERN	CORRECTIVE ACTION / SUGGESTED MITIGATION
Checklists non compliant to DO-178B objective	Make checklist compliance matrix to DO-178B objective
Standards too generic and does not cover the tool usage perspective	Make the standards inline to the usage of tools like DOORS, Rhapsody, Rational Test Real Time, VectorCast, Matlab/Simulink etc
Lack of test case selection criteria	Need to add the test selection criteria
Transition Criteria – either too generic or too difficult to be adhered to	Make transition criteria checklist that could be quantified

Focused attention on the planning phase – Development & V&V approach, Standards, Checklists & Transition Criteria are the key and seen as important contributor to the success during project execution

B. Development Phase

COMMON ISSUES & CONCERN	CORRECTIVE ACTION / SUGGESTED MITIGATION
Incorrect decomposition of Data and control flow in software architecture	Usage of tool that can support the decomposition of process & Data such as Rhapsody, Matlab etc
Insufficiency of Requirement coverage (HLR ⇔ LLR ⇔ Code)	Use DOORS. Review the requirement coverage from forward/backward trace
Derived Requirement Rationale	As part of requirement standard, make a rationale attribute to support the existence of derived requirements
Requirements not verifiable	Involvement of Test engineer during requirements review for the verifiability aspect
Manual Code review approach ends up in not finding the key functional issue	Use static rule checkers. HCL proposes to using MISRA:2004 code compliance and tools such as PC-Lint, Logiscope. This can save effort & be more productive
Issues with the software when loaded on the target board on system environment (more than target board)	Perform developmental integration on the setup that is closer to the system

C. Testing Phase

COMMON ISSUES & CONCERN	CORRECTIVE ACTION / SUGGESTED MITIGATION
Insufficient Requirements coverage w.ith respect to Test Case, i.e, multiple test case(s) or block of test cases traces to multiple requirement. Hence the credibility of requirement coverage of individual requirement questionable	Test Case format to be planned with sample test case(s), requirement(s). Trace the requirement to the each test blocks within the test case suite
Robustness Test not sufficient or does not exists	Have testing guidance on the robustness criteria, team orientation, and review checklist asking for specific question on robustness
Description of test case	Agreement on the sample test case descriptions. Description to state the object of the test case and not the test case itself
Data Dictionary, Test Case mismatch	Standard to document the content of data dictionary, design and test review to have the specific checklist point on adherence to data dictionary
Insufficient / Incorrect Test Setup Instruction	Person not part of project executing the test independently. QA witnessing the same
	QA to perform the setup, build and load audit prior to test execution. CM review of test result archive in association to the test procedure versions
Coverage Analysis justification inappropriate	Clearly conclude if the lack of coverage requires any change in test, requirement, software or, it is justifiable for its presence like defensive code, standards etc
Data & Control Coupling either overdone or, does not meet the objectives	Perform the Data and Control Coupling as described in paper - Paper-DO-178B_SW_Low_Level_Testing.pdf

D. Configuration Management

COMMON ISSUES & CONCERN	CORRECTIVE ACTION / SUGGESTED MITIGATION
Problem report analysis and its linkage to the version of the impacted items	Avoiding multiple CM system if possible. strengthening CCB approval on this focus area including independent quality check
Incorrect / Insufficient Impact Analysis	Ensure the impacted items have been updated as per and only what the analysis says with revision. Justify as part of CCB if changes have been done additionally

COMMON ISSUES & CONCERN	CORRECTIVE ACTION / SUGGESTED MITIGATION
Inconsistency in the configuration version of review artifact of test cases, test procedure and the requirements, and incorrect checklist version itself. Same points holds good for development & test results output also	Perform QC check as part of release for association of review artifacts to the configuration version, QA audit

E. Quality Assurance

COMMON ISSUES & CONCERN	CORRECTIVE ACTION / SUGGESTED MITIGATION
Incomplete Review record fields and its closure	Generate review guideline & induction plan for every reviewer. QA audit on checklist closure
Transition criteria adherence	QA audit

Whoever tries to steal the contents of this book shall die "Naked & Alone".

- Curse of the Author