**Introduction**

I have completed my first internship in TAI (Turkish Aerospace Industries Inc.) between the time intervals 11.06.2012 and 17.08.2012. I worked at the Software Department of Uzay Sistemleri Başkanlığı.

In the first two days of my internship, some presentations about the company, its work fields and future expectations were done. Contents of those presentations will be explained in the “Description of the Company” part in detailed.

After the presentations, I went to my department and the software verification engineer Serkan Dural gave training about our position in the department and what kind of works Software Department do. Basically, there are two units in the Software Department which are Software Development Unit and Software Test Unit. In the Software Development Unit, the software, which will be used in the flight program, are built to satisfy some requirements which are prepared by software development engineers according to needs of whole project.

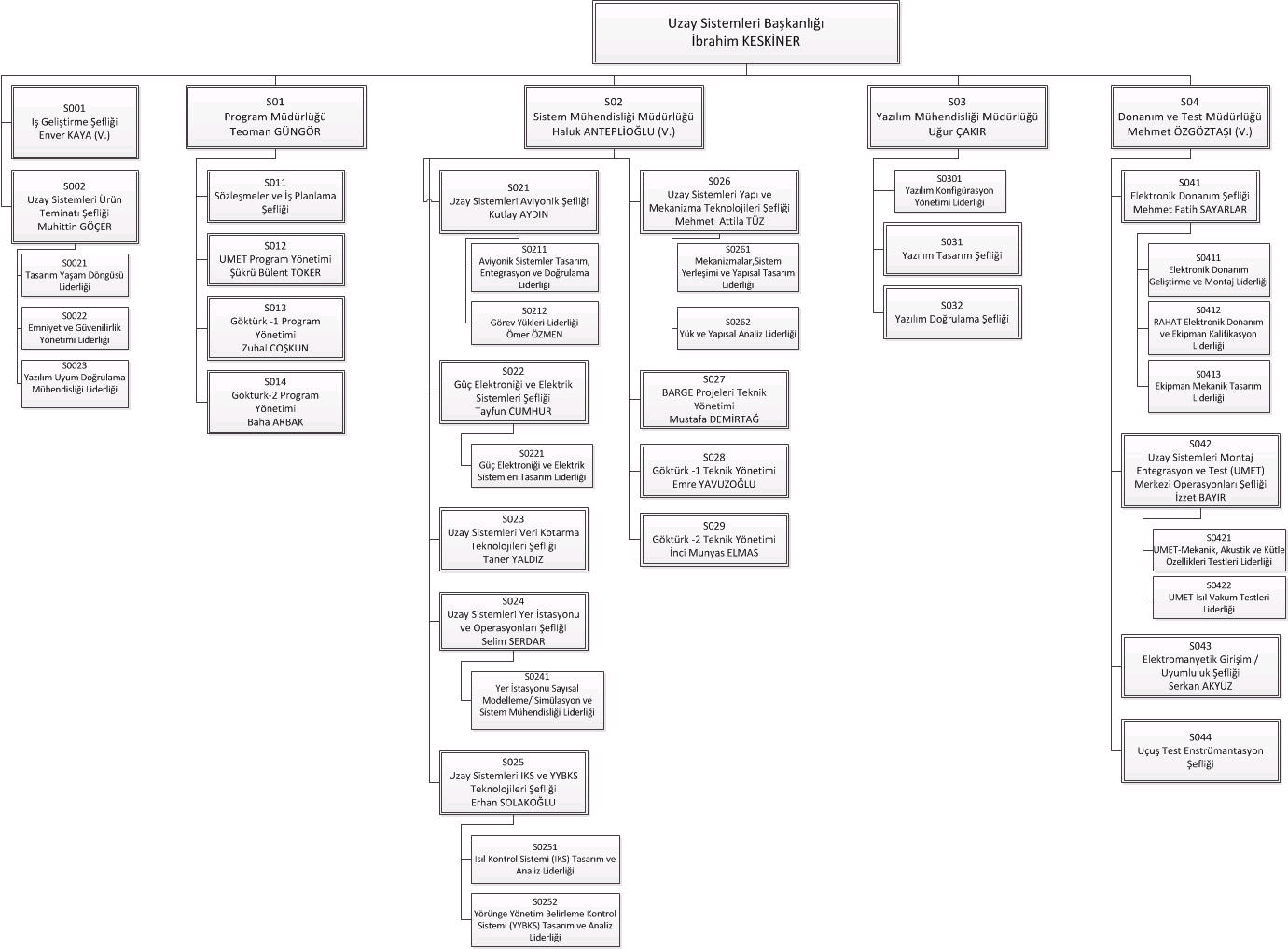
In the Software Test Unit, the software built by software development engineers is tested whether it satisfies all requirements or not. To do this, whole code built by software development engineer is divided into small partitions. Then these partitions are tested individually by using some test codes written by this department. These codes test all possible results in the requirements. Inputs and outputs coming from other partitions are sent manually. My position in the department is mainly to write these test codes. In addition to, review test codes written by other authors in the Software Test Unit and verify whether these codes satisfy all requirements or not. If not, test code is changed and published in new version by its author. Throughout the internship after the presentations and training, I worked on Display Logic (DL) partition of Human Machine Interface (HMI). HMI is lower software module of OFP Software. These parts will be discussed in the section “Work done” in detailed.

**TAI (Turkish Aerospace Industry)**

TAI, the center of technology in design, development, manufacturing, integration of aerospace systems, modernization and after sales support in Turkey, was established on 15 May 1984. Located in Akıncı - Ankara, TAI’s modern aircraft facilities, which cover an area of 5 million square meters with an industrial facility of over 200,000 square meters under roof, is furnished with high technology machinery and equipment that provide extensive manufacturing capabilities. Furthermore, in order to develop a ground for defense-aerospace skills, and create synergy among small to medium enterprises and universities, a significant part of engineering-based activities are conducted at the Techno Park located in the Middle East Technical University. The quality system of the Company meets the stringent world standards including NATO AQAP-2110, ISO-9001:2000 and AS EN 9100. TAI has a total of 3000 highly motivated employees, 1200 of whom are engineers. The shareholders of the Company are; the Turkish Armed Forces Foundation, the Under secretariat for Defense Industries and Turkish Aeronautical Association. TAI, which participates in global-scale design and development programs, is also engaged in the design and manufacturing of structural components with leading international aerospace companies. With its proven experience, TAI is a uniquely qualified supplier for Aermacchi, AgustaWestland, Airbus, Boeing, EADS CASA, Eurocopter, Lockheed Martin, Northrop Grumman, MDHI, Sikorsky and many more. Since its establishment, TAI’s employees, baring in mind their target to develop not only Turkey’s national power, but also the technological capacity that will support the military capacity, have been carrying out their activities to develop the necessary capabilities and products in order to meet the aerospace requirements of the Turkish Armed Forces with “indigenous” systems. To this end, TAI, in line with its vision and mission statements, has established a modern aerospace facility, and successfully realized the co-production of F-16 fighters, CN-235 light transport/maritime patrol/surveillance aircraft, SF-260 trainers, Cougar AS-532 general purpose helicopters. With its proven experience and know-how, TAI has improved its capabilities in the fields of design, production, modernization, modification and systems integration of fixed and rotary wing air platforms, unmanned aerial vehicles and satellite. Being the main contractor of ATAK - Attack/Tactical Reconnaissance Helicopters Program, TAI will not only customize, but also produce and provide integrated logistics support of the helicopter in accordance with the user needs. TAI, which is the prime contractor of the Turkish Unmanned Aerial Vehicle (MALE) production program, is engaged in design and development of Primary & Basic Trainer (HÜRKUŞ) Aircraft. Baring in mind its target to provide the Turkish Armed Forces with indigenous systems, TAI continues its activities regarding the design and production of Turkish Primary and Basic Trainer Aircraft (HÜRKUŞ) and Turkish Indigenous Medium Altitude Long Endurance (MALE) Unmanned Aerial Vehicle (TIHA).

TAI, which actively participates in the custom satellite development program of Turkey, will be the local integrator company for the International Satellite Acquisition Programs. To this end, a new Satellite Assembly and Integration Test Facility will be built. In addition to indigenous programs, TAI’s core business also includes modernization, modification and systems integration programs and after sales support of both fixed and rotary wing military and commercial aircraft that are in the inventory of Turkey and friendly countries. TAI is the prime contractor of the avionic modernization programs of the C-130 transport aircraft and T-38 training aircraft that are in the inventory of the Turkish Air Force Command (TuAF). Major modernization programs include Glass Cockpit modification of Turkish Black Hawk helicopters, electronic warfare retrofit and structural modifications on TuAF F-16s, Falcon Star and Mid Life Upgrade modifications of F-16’s that are in the inventory of the Royal Jordanian Air Force, modification and modernization of Cougar AS-532, modification of CN-235 platforms for MPA/MSA missions for the Turkish Navy and Coast Guard, modification of ATR-72 platforms for the Turkish Navy as well as structural modification and systems integration activities required for the conversion of B737-700 aircraft into AEW&C aircraft. TAI participates, as a partner, in the global scale Joint Strike Fighter (JSF/F-35) and A400M design and development programs. Being the shareholder of Airbus Military S.L., as National Industrial Institution, TAI has been participating in the design and development activities of A400M with the leading European aerospace companies namely; Airbus, EADS and FLABEL. TAI, which also keeps on developing its capabilities in commercial aviation, has also become a full risk-sharing partner of Airbus in the A350XWB program for the aileron work package. Furthermore, by utilizing capabilities of the Turkish Armed Forces’ Maintenance Centers, TAI also gives maintenance, repair and overhaul services to its customers. Determined to keep abreast of global technological developments and secure its place among the major aerospace companies, TAI aims to lead Turkey to new horizons in aviation in the 21st century.

**Organizational Structure of the Company and Department**



**Erciyes (C-130) Project**

****

The contract of C-130 Avionics Modernization (Erciyes) Program was signed between the under secretariat for Defense Industries (SSM) and TAI in December 2006. Under the program, the avionics modernization of a total of 13 TuAF C-130 aircraft (seven C-130E and six C-130B) will be realized.

The scope of Erciyes Program:

* Making the A/C ready for international flight rules like GATM, RVSM,
* Analog to Glass Cockpit,
* Unique and expendable, Mission Computer,
* Night Vision Goggle Compatible Lighting System,
* Design, manufacture and deliver the System Integration Laboratory after prototyping,
* Design, manufacture and deliver the Ground Mission Planning system after prototyping.

TAI will complete the modernization of two prototype aircraft, and will oversee the modernization of the remaining 11 aircraft at TuAF 2nd Air Supply and Maintenance Center. TAI is also to provide Integrated Logistics Support for the new systems.

When the Erciyes Program, which will be carried out in cooperation with TuAF 2nd Air Supply and Maintenance Center, is completed, the aircraft will not only will be compatible with international flight rules, but their flight safety will be increased as well. The aircraft will also become long-term supportable with the modernized systems.

The program is very important for TAI, given the complexity of the modernization and avionics configuration, which are to be carried out completely with national capabilities. In addition to the development of the indigenous mission computer, the software development operations, which will be installed to the computer, will also be realized by TAI.

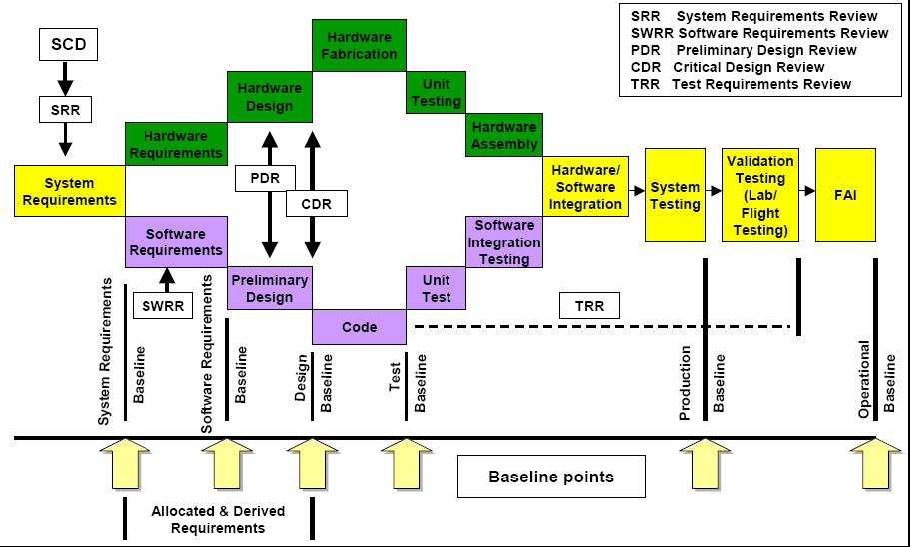
Under the program, the technical documentation and user manuals will be prepared by TAI and delivered to TuAF. Furthermore, upon delivery of the aircraft, TAI will also provide an Integrated Logistics Support for 20 years.

With the realization of the Erciyes Program, TAI will gain the capability to meet the modernization requirements of all countries that have C-130 aircraft in their inventories.

The first Turkish Air Force (TuAF) C-130 aircraft, the avionics modernization of which will be realized by TAI, arrived at TAI’s facilities on December 26, 2007. The aircraft, which took off from the 12th Transmission Base Commandership based in Kayseri, landed at TAI’s facilities, where it will go through series of avionics modernization activities.

With the arrival of the first C-130, the prototype period, which is planned to last 30-months, and comprises the modernization of the first two aircraft by TAI, was initiated. Under the Erciyes program, expected to last 56 months, TAI will oversee the modernization of the remaining 11 aircraft at TuAF 2nd Air Supply and Maintenance Center.

**Tasks of Verification and Validation Team**



**Software Verification Process**

This section discusses the objectives and activities of the software verification process.

Verification is a technical assessment of the results of both the software development

processes and the software verification process. The software verification process is

applied as defined by the software planning process (section 4) and the Software

Verification Plan (subsection 11.3).

Verification is not simply testing. Testing, in general, cannot show the absence of errors. As a result, the following subsections use the term "verify" instead of "test" when the software

verification process objectives being discussed are typically a combination of reviews,

analyses and test.

Tables A-3 through A-7 of Annex A contain a summary of the objectives and outputs of

the software verification process, by software level.

**Note:** *For lower software levels, less emphasis is on:*

* *Verification of low-level requirements.*
* *Verification of the software architecture.*
* *Degree of test coverage.*
* *Control of verification procedures.*
* *Independence of software verification process activities.*
* *Overlapping software verification process activities, that is, multiple*

*verification activities, each of which may be capable of detecting the*

*same class of error.*

* *Robustness testing.*
* *Verification activities with an indirect effect on error prevention or*

*detection, for example, conformance to software development standards.*

**6.1 Software Verification Process Objectives**

The purpose of the software verification process is to detect and report errors that may

have been introduced during the software development processes. Removal of the errors is an activity of the software development processes.

The general objectives of the software verification process are to verify that:

a. The system requirements allocated to software have been developed into

software high-level requirements that satisfy those system requirements.

b. The high-level requirements have been developed into software architecture

and low-level requirements that satisfy the high-level requirements. If one or

more levels of software requirements are developed between high-level

requirements and low-level requirements, the successive levels of requirements

are developed such that each successively lower level satisfies its higher level

requirements. If code is generated directly from high-level requirements, this

objective does not apply.

c. The software architecture and low-level requirements have been developed

into Source Code that satisfies the low-level requirements and software

architecture.

d. The Executable Object Code satisfies the software requirements.

e. The means used to satisfy these objectives are technically correct and

complete for the software level.

**6.2 Software Verification Process Activities**

Software verification process objectives are satisfied through a combination of reviews,analyses, the development of test cases and procedures, and the subsequent execution ofthose test procedures. Reviews and analyses provide an assessment of the accuracy,completeness, and verifiability of the software requirements, software architecture, and Source Code. The development of test cases may provide further assessment of the internalconsistency and completeness of the requirements. The execution of the test procedures provides a demonstration of compliance with the requirements.

The inputs to the software verification process include the system requirements, the

software requirements and architecture, traceability data, Source Code, Executable Object Code, and the Software Verification Plan.

The outputs of the software verification process are recorded in Software Verification

Cases and Procedures (subsection 11.13) and Software Verification Results (subsection 11.14). The need for the requirements to be verifiable once they have been implemented in the software may itself impose additional requirements or constraints on the software

development processes.

The verification process provides traceability between the implementation of the software requirements and verification of those software requirements:

The traceability between the software requirements and the test cases is

accomplished by the requirements-based coverage analysis.

The traceability between the code structure and the test cases is accomplished by

the structural coverage analysis.

Guidance for the software verification activities includes:

a. High-level requirements and traceability to those high-level requirements should be verified.

b. The results of the traceability analyses and requirements-based and structural

coverage analyses should show that each software requirement is traceable to the code that implements it and to the review, analysis, or test case that verifies it.

c. If the code tested is not identical to the airborne software, those differences should be specified and justified.

d. When it is not possible to verify specific software requirements by exercising the software in a realistic test environment, other means should be provided and their justification for satisfying the software verification process objectives defined in the Software Verification Plan or Software Verification Results.

e. Deficiencies and errors discovered during the software verification process should be reported to the software development processes for clarification and correction.

**6.3 Software Reviews and Analyses**

Reviews and analyses are applied to the results of the software development processes and software verification process. One distinction between reviews and analyses is that analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment 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.

**6.3.1 Reviews and Analyses of the High-Level Requirements**

The objective of these reviews and analyses is to detect and report requirements errors that may have been introduced during the software requirements process. These reviews and analyses confirm that the high-level requirements satisfy these objectives:

a. Compliance with system requirements: The objective is to ensure that the system functions to be performed by the software are defined, that the functional, performance, and safety-related requirements of the system are satisfied by the software high-level requirements, and that derived requirements and the reason for their existence are correctly defined.

b. Accuracy and consistency: The objective is to ensure that each high-level

requirement is accurate, unambiguous and sufficiently detailed and that the

requirements do not conflict with each other.

c. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the high-level requirements and the hardware/software features of the target computer, especially, system response times and input/output hardware.

d. Verifiability: The objective is to ensure that each high-level requirement can be verified.

e. Conformance to standards: The objective is to ensure that the Software

Requirements Standards were followed during the software requirements process and that deviations from the standards are justified.

f. Traceability: The objective is to ensure that the functional, performance, and safetyrelated requirements of the system that are allocated to software were developed into the software high-level requirements.

g. Algorithm aspects: The objective is to ensure the accuracy and behavior of the proposed algorithms, especially in the area of discontinuities.

**6.3.2 Reviews and Analyses of the Low-Level Requirements**

The objective of these reviews and analyses is to detect and report requirements errors that may have been introduced during the software design process.

These reviews and analyses confirm that the software low-level requirements satisfy these objectives:

a. Compliance with high-level requirements: The objective is to ensure that the

software low-level requirements satisfy the software high-level requirements and that derived requirements and the design basis for their existence are correctly defined.

b. Accuracy and consistency: The objective is to ensure that each low-level

requirement is accurate and unambiguous and that the low-level requirements do not conflict with each other.

c. Compatibility with the target computer: The objective is to ensure that no conflicts exist between the software requirements and the hardware/software features of the target computer, especially, the use of resources (such as bus loading), system response times, and input/output hardware.

d. Verifiability: The objective is to ensure that each low-level requirement can be verified.

e. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process, and that deviations from the standards are justified.

f. Traceability: The objective is to ensure that the high-level requirements and derived requirements were developed into the low-level requirements.

g. Algorithm aspects: The objective is to ensure the accuracy and behavior of the

proposed algorithms, especially in the area of discontinuities.

**6.3.3 Reviews and Analyses of the Software Architecture**

The objective of these reviews and analyses is to detect and report errors that may have been introduced during the development of the software architecture.

These reviews and analyses confirm that the software architecture satisfies these objectives:

a. Compatibility with the high-level requirements: The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes.

b. Consistency: The objective is to ensure that a correct relationship exists between the components of the software architecture. This relationship exists via data flow and control flow.

c. Compatibility with the target computer: The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization and interrupts, between the software architecture and the hardware/software features of the target computer.

d. Verifiability: The objective is to ensure that the software architecture can be

verified, for example, there are no unbounded recursive algorithms.

e. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, especially complexity restrictions and design constructs that would not comply with the system safety objectives.

f. Partitioning integrity: The objective is to ensure that partitioning breaches are

prevented or isolated.

**6.3.4 Reviews and Analyses of the Source Code**

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.

The topics should include:

a. 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.

b. 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.

c. 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.

d. 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.

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

f. 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 uninitialized variables or constants, unused variables or constants, and data corruption due to task or interrupt conflicts.

**6.3.5 Reviews and Analyses of the Outputs of the Integration Process**

The objective is to ensure that the results of the integration process are complete and

correct. This could be performed by a detailed examination of the linking and loading data and memory map. The topics should include:

a. Incorrect hardware addresses.

b. Memory overlaps.

c. Missing software components.

**6.3.6 Reviews and Analyses of the Test Cases, Procedures and Results**

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 should include:

a. Test cases: The verification of test cases is presented in paragraph 6.4.4.

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.

**6.4 Software Testing Process**

Testing of airborne software has two complementary objectives. One objective is to

demonstrate that the software satisfies its requirements. The second objective is 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.

The objectives of the three types of testing in the figure are:

* Hardware/software integration testing: To verify correct operation of the software in the target computer environment.
* Software integration testing: To verify the interrelationships between software requirements and components and to verify the implementation of the software requirements and software components within the software architecture.
* Low-level testing: To verify the implementation of software low-level requirements.

**Note:** *If a test case and its corresponding test procedure are developed and executed for hardware/software integration testing 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.*

To satisfy the software testing objectives:

a. Test cases should be based primarily on the software requirements.

b. Test cases should be developed to verify correct functionality and to establish conditions that reveal potential errors.

c. Software requirements coverage analysis should determine what software

requirements were not tested.

d. Structural coverage analysis should determine what software structures were not exercised.



**FIGURE 6-1**

**SOFTWARE TESTING PROCESS**

**6.4.1 Test Environment**

More than one test environment may be needed to satisfy the objectives for software

testing. An excellent test environment includes the software loaded into the target computer and tested in a high fidelity simulation of the target computer environment.

**Note:** *In many cases, the requirements-based coverage and structural coverage*

*necessary can be achieved only with more precise control and monitoring of*

*the test inputs and code execution than generally possible in a fully integrated*

*environment. Such testing may need to be performed on a small software*

*component that is functionally isolated from other software components.*

Certification credit may be given for testing done using a target computer emulator or a host computer simulator.

Guidance for the test environment includes:

a. Selected tests should be performed in the integrated target computer environment, since some errors are only detected in this environment.

**6.4.2 Requirements-Based Test Case Selection**

Requirements-based testing is emphasized because this strategy has been found to be the most effective at revealing errors. Guidance for requirements-based test case selection includes:

a. To implement the software testing objectives, two categories of test cases should be included: normal range test cases and robustness (abnormal range) test cases.

b. The specific test cases should be developed from the software requirements and the error sources inherent in the software development processes.

**6.4.2.1 Normal Range Test Cases**

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 include:

a. Real and integer input variables should be exercised using valid equivalence classes and boundary values.

b. 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.

c. For state transitions, test cases should be developed to exercise the transitions possible during normal operation.

d. For software requirements expressed by logic equations, the normal range test cases should verify the variable usage and the Boolean operators.

**Note:** *One method is to test all combinations of the variables. For complex*

*expressions, this method is impractical due to the large number of test cases*

*required. A different strategy that ensures the required coverage could be*

*developed. For example, for Level A, the Boolean operators could be verified*

*by analysis or review, and to complement this activity, test cases could be*

*established to provide modified condition/decision coverage.*

**6.4.2.2 Robustness Test Cases**

The objective of robustness test cases is to demonstrate the ability of the software to

respond to abnormal inputs and conditions. Robustness test cases include:

a. Real and integer variables should be exercised using equivalence class selection of invalid values.

b. System initialization should be exercised during abnormal conditions.

c. The possible failure modes of the incoming data should be determined, especially complex, digital data strings from an external system.

d. For loops where the loop count is a computed value, test cases should be

developed to attempt to compute out-of-range loop count values, and thus

demonstrate the robustness of the loop-related code.

e. A check should be made to ensure that protection mechanisms for exceeded frame times respond correctly.

f. For time-related functions, such as filters, integrators and delays, test cases should be developed for arithmetic overflow protection mechanisms.

g. For state transitions, test cases should be developed to provoke transitions that are not allowed by the software requirements.

**6.4.3 Requirements-Based Testing Methods**

Requirements-based testing methods consist of methods for requirements-based

hardware/software integration testing, requirements-based software integration testing, and requirements-based low-level testing. With the exception of hardware/software integration testing, these methods do not prescribe a specific test environment or strategy.

Guidance includes:

a. Requirements-Based Hardware/Software Integration Testing: 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.

b. Requirements-Based 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. 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.

c. Requirements-Based Low-Level Testing: This testing method should concentrate on demonstrating that each software component complies with its low-level requirements. The objective of requirements-based low-level testing is to ensure that the software components satisfy their low-level requirements.

Typical errors revealed by this testing method include:

* Failure of an algorithm to satisfy a software requirement.
* Incorrect loop operations.
* Incorrect logic decisions.
* Failure to process correctly legitimate combinations of input conditions.
* Incorrect responses to missing or corrupted input data.
* Incorrect handling of exceptions, such as arithmetic faults or violations of array limits.
* Incorrect computation sequence.
* Inadequate algorithm precision, accuracy or performance.

**6.4.4 Test 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 (subparagraph 6.4.4.3).

**6.4.4.1 Requirements-Based Test Coverage Analysis**

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:

a. Test cases exist for each software requirement.

b. Test cases satisfy the criteria of normal and robustness testing as defined in

paragraph 6.4.2.

**6.4.4.2 Structural Coverage Analysis**

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:

a. The analysis should confirm the degree of structural coverage appropriate to the

software level.

b. 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.

c. The analysis should confirm the data coupling and control coupling between the

code components.

**6.4.4.3 Structural Coverage Analysis Resolution**

Structural coverage analysis may reveal code structure that was not exercised during

testing. Resolution would require additional software verification process activity.

This unexecuted code structure may be the result of:

a. Shortcomings in requirements-based test cases or procedures: The test cases

should be supplemented or test procedures changed to provide the missing

coverage. The method(s) used to perform the requirements-based coverage analysis may need to be reviewed.

b. Inadequacies in software requirements: The software requirements should be

modified and additional test cases developed and test procedures executed.

c. Dead code: The code should be removed and an analysis performed to assess the effect and the need for reverification.

d. Deactivated code: For deactivated code which is not intended to be executed in any configuration used within an aircraft or engine, a combination of analysis and testing should show that the means by which such code could be inadvertently executed are prevented, isolated, or eliminated. For deactivated code which is only executed in certain configurations of the target computer environment, the operational configuration needed for normal execution of this code should be established and additional test cases and test procedures developed to satisfy the required coverage objectives.

 Software life cycle data and processes

# SYSTEM OVERVIEW

The purpose of Erciyes Program is the avionics modernization of C-130E and C-130B type aircrafts for Turkish Air Force (TuAF). Operational capabilities like Reduced Vertical Separation Minima (RVSM), Required Navigation Performance 1 Nautical Mile (RNP-1), Category II Instrument Landing System Approach, Area Navigation (RNAV)/ Global Positioning System (GPS) Approach, and Lateral Navigation (LNAV) will be incorporated by C-130E/B modernization. All the aircrafts will be equipped with night vision compatible lighting system and glass cockpit. After the successful completion of the Erciyes Project, TuAF C-130 aircrafts will be compliant with para 3. Certification Requirements of Appendix C of the ERCİYES Contract [3].

## Software Verification Process

The software verification process encompasses the formal verification activities on the software life cycle data. The outputs of this process are review records, analyis records, formal verification test cases and procedures which fulfill DO-178B [1] Software Life Cycle Data item 11.13, and software verification results, which fulfills data item 11.14. Included as part of the verification results are a traceability analysis to software requirements, and a structural coverage analysis of software execution.

The verification process addresses all the objectives and activities defined in DO-178B [1] section 6.0.

# ORGANIZATION

Organization: Organizational responsibilities within the software verification process and interfaces with the other software life cycle processes, *{DO-178B Section 11.3.a}.*

Figure-1 shows the organizational structure of the OFP software project.

**Development Team**

**OFP Software**

**Responsible Leader**

**Verification Team**

**SQAR**

Support

**PM Specialist**

**SCMR**

**SW Integration Team**

**CVE and AS**

Figure 1 Organizational Structure

The following table gives the responsible team for each type of verification activity that will be conducted for each OFP module.

|  |  |
| --- | --- |
| **Verification activity** | **Responsible Team** |
| R-BT development, review, execution,  Integration test development, review, execution  SCA and SCA review,  RCA,  Object Code Analysis & review  STD preparation and review  STR preparation and review | Verification Team |
| Software Requirements Reviews,  Design Reviews,  Source code reviews,  Traceability analysis and review among system requirements, software requirements, design and source code,  Model Based Testing  Model Based Test Reviews  Model Based Coverage  Model Based Coverage Review | Development Team |
| Data and Control Coupling Analysis  Software Integration Testing and Review | SW Integration Team |
| Mainly responsible for the Certification Liaison Process, however CVE and AS also participate in the peer review of the SW life cycle outputs in order to ensure that the product satisfies the certification needs. | CVE and AS |

Table 2 Organizational responsibilities

# VERIFICATION PROCESS

This section describes the verification process that will be applied in verifying the OFP software. The following figures show the role of the verification process throughout SW development life cycle.

OFP software architecture is based on approximately 20 partitions. Each partition is a separately build and linked component. Partitions communicate with each other using ARINC-653 space and time partitioning specification.

Verification activities are conducted in two levels. In partition level, each artifact that composes that partition is verified. In OFP level, the integrity of the partitions is verified.

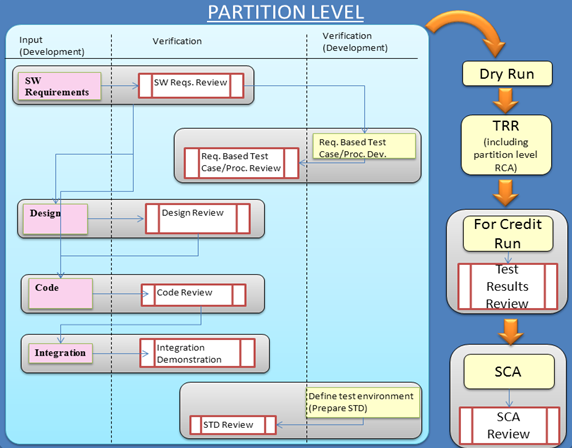


Figure 2 Partition Level SW Verification Life Cycle

# SOFTWARE OVERVIEW

## Software Modules

The OFP Software contains of the following software modules: Aircraft Subsystems (ACS), Human Machine Interface (HMI), System Management (SYS), Flight Management System (FMS), Database (DB) and MAP (See Figure 3). Each software module contains one or more partitions. Besides the OFP software modules, Figure 3 also shows third party components that are running on the CCC hardware such as, Integrity 178B OS, BSP&Drivers, ARINC-653 APEX, IPFLite, PJFS, Boot loader and Data Loader. COTS and third party components are represented as rounded corner boxes in Figure 3. See Sections 10.4 and 10.4.1 for COTS and third party components.

OFP Software Architecture

ACS module provides DEVMGR\_A, DEVMGR\_D, IO\_A429, IO\_M1553, IO\_SIGNAL,

SYS module consists of CCC\_MGR, APPL\_MON, APPL\_MAINT, DISP\_MON, DISP\_MAINT, MAP\_MON and MAP\_MAINT partitions. SYS module provides system control services, storage services, partition management and initialization, time management, fault management, master/backup logic and log management for maintenance purposes.

|  |  |
| --- | --- |
| **Organization Name** | **Responsibilities** |
| Software Project Responsible | Responsible for the Software Planning Process |
| Systems Engineering Group | Responsible for developing Operational Requirements Specifications. |
| Avionics System Engineering Group | Responsible for developing Functional Requirements Specifications. |
| Development Team | Responsible for the Software Development Processes including the Requirements, Design, Coding and Integration processes. Also supports Software Planning Process for SDP and PSAC planning. |
| Integration Team | Responsible for Integration processes and build scripts. |
| Verification Team | Responsible for the Software Verification Process, and supports Software Planning Process for SVP. |
| Software Configuration Management | Responsible for the Software Configuration Management Process, and supports Software Planning Process for SCMP. |
| Software Quality Assurance | Responsible for the Software Quality Assurance Process, and supports Software Planning Process for SQAP. Supports the Certification Liaison Process. |
| AS and CVE | Responsible for the Certification Liaison Process. |

OFP Software Development Organization And Responsibilities

# Testing Techniques

To be applied test techniques at test cases level are based on ISTQB dynamic test techniques. The dynamic test techniques can be summarized as:

* **Specification/Requirements based testing:** The specific test cases should be developed from the software specifications.
* **Positive (Normal Range) testing:** The normal range test cases demonstrate the ability of the software to respond to normal inputs and conditions. Normal range test cases include:
  + Real and integer input variables should be exercised using valid equivalence classes and 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.
* **Negative (Robustness) testing:** The robustness test cases demonstrate the ability of the software to respond to abnormal inputs and conditions. Robustness test cases include:
  + Real and integer variables should be exercised using equivalence class selection of invalid values.
  + System initialization should be exercised during abnormal conditions.
  + The possible failure modes of the incoming data should be determined, especially complex, digital data strings from an external system.
  + For loops where the loop count is a computed value, test cases should be developed to attempt to compute out-of-range loop count values, and thus demonstrate the robustness of the loop-related code.
  + A check should be made to ensure that protection mechanisms for exceeded frame times respond correctly.
  + For time-related functions, such as filters, integrators and delays, test cases should be developed for arithmetic overflow protection mechanisms.
  + For state transitions, test cases should be developed to provoke transitions that are not allowed by the software requirements
* **Equivalence partitioning**: A portion of the SW input or output domains for which the SW behavior is assumed to be the same from the SW specification.
* **Boundary Value analysis:** A test case design technique for SW or for SW part an input value or output value which is on the boundary between equivalence, or an incremental distance either side of the boundary.
* **State transition testing:** A test case design technique in which test cases are designed to execute a transition between two allowable states of a system.
* **Cause-effect graphing:** A test case design technique where a graphical representation of inputs or stimuli (causes) with their associated outputs (effects) is shown.
* **Syntax testing:** A test case design techniques for a software or system in which test case design is based upon the syntax of the input (interface).
* **Fault guessing:** A test case design technique where the experience of the tester is used to postulate what fault might occur, and to design test specifically to expose them.

**Test Phases**

The following test phases are identified and will be used once or multiple times during the test process.

Some requirements based tests are divided in different test phases. The objective is to able categorize them conform the used test standards and making them understandable from the testing point of view.

The identified test phases are:

* **Entry Check**: Static check at every internal delivery of the executable software package.
* **Regression tests:** Test to expose the existing functional behavior is remained and functioning correctly as before.
* **Interface testing**: The purpose of interface testing is to test the interfaces, particularly the external interfaces with the system. The emphasis is on verifying exchange of data, transmission and control, and processing times.
* **GUI testing**: Is the process of testing a graphical user interface to ensure it meets its written specifications
* **HW/SW Integration testing:** **I**s the testing the [application](http://www.opfro.org/Components/WorkProducts/IntegrationSet/Application/Application.html) against its HW integration [requirements](http://www.opfro.org/Glossary/GlossaryR.html#requirement). This test phases which in the individual software modules (CSCI’s) are tested. Test Cases are constructed to test that modules (CSCI’s) within the test scope interact correctly with the HW platform and Operating System resources provided by the target SBC.
* **SW Integration testing:** **I**s the testing the [application](http://www.opfro.org/Components/WorkProducts/IntegrationSet/Application/Application.html) against its SW integration [requirements](http://www.opfro.org/Glossary/GlossaryR.html#requirement). This test phases which in the individual software modules (CSCI’s) are combined and tested as a group. Test Cases are constructed to test that all modules (CSCI’s) within the test scope interact correctly with other SW modules (CSCI’s) conform the interface control specification.
* **Functional testing:** Is testing conducted on a complete, integrated system to evaluate the software on the integrated system compliance with its specified software [requirements](http://en.wikipedia.org/wiki/Requirements). These help you in testing if the required functionality is working as per the specifications and if the expected result is correct.
* **Robustness testing:** **I**s the [system testing](http://www.opfro.org/Components/WorkUnits/Activities/Testing/SystemTesting.html) of an integrated [application](http://www.opfro.org/Components/WorkProducts/IntegrationSet/Application/Application.html) against its [robustness](http://www.opfro.org/Glossary/GlossaryR.html#robustness) [requirements](http://www.opfro.org/Glossary/GlossaryR.html#requirement). The goals of robustness testing are to cause the application to [fail](http://www.opfro.org/Glossary/GlossaryF.html#failure) under **invalid conditions** so that the underlying faults can be identified, analyzed, fixed, and prevented in the future.
* **Stabilization / Memory Management testing:** Is testing conducted on a complete, integrated system to evaluate the stabilization of the software indicated in terms of MTBF.
* **Performance testing:** Is the system testing of an integrated application against its performance requirements. These tests are to determine that the system meets performance criteria under a particular workload.
* **Stress testing:** Is testing that determines the [robustness](http://en.wikipedia.org/wiki/Robust) of software by testing beyond the limits of normal operation. Stress testing is particularly important for "[mission critical](http://en.wikipedia.org/wiki/Mission_critical)" software, but is used for all types of software. Stress tests commonly put a greater emphasis on robustness, [availability](http://en.wikipedia.org/wiki/Availability), and [fault handling](http://en.wikipedia.org/wiki/Error_handling) under a heavy load, than on what would be considered correct behavior under normal circumstances.

**Exit check:** Static check will be performed at every external delivery of the executable software package. The checklists will contains mainly configuration items like every requirements -, design - and verification document will be reviewed and are in defined status. All PR’s and CR’s regarding to the related block will be solved or implemented. Unsolved PR should be controlled by CCB chaired by the DER. The source code, object code and executable code will be labeled as defined in configuration management plan.

**Approach of boxes**

Software testing methods are traditionally divided into black box testing and white box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

**Black box testing**

Black box testing treats the software as a "black box"—without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing.

* **Specification-based testing**: Specification-based testing aims to test the functionality of software according to the applicable requirements. Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case.

Specification-based testing is necessary, but it is insufficient to guard against certain risks.

* **Advantages and disadvantages**: The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code *must* have bugs. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers do not. *But,* on the other hand, black box testing has been said to be "like a walk in a dark labyrinth without a flashlight," because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when (1) a tester writes many test cases to check something that could have been tested by only one test case, and/or (2) some parts of the back-end are not tested at all.

Therefore, black box testing has the advantage of "an unaffiliated opinion," on the one hand, and the disadvantage of "blind exploring," on the other.

**White box testing**

White box testing is when the tester has access to the internal data structures and algorithms including the code that implement these.

Types of white box testing

The following types of white box testing exist:

API testing (application programming interface) - Testing of the application using Public and Private APIs

Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)

Fault injection methods - improving the coverage of a test by introducing faults to test code paths

Mutation testing methods

Static testing - White box testing includes all static testing

Code completeness evaluation

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.

Two common forms of code coverage are:

*Function coverage*, which reports on functions executed

*Statement coverage*, which reports on the number of lines executed to complete the test

They both return a code coverage metric, measured as a percentage.

**Grey Box Testing**

Grey box testing involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

**Checklist for Test Case / Procedure Review**

Correctness and Completeness

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Normal Range Tests

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Robustness Tests

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Note: In case the answer is `No` or NA (Not Applicable) an explanation is required.

**Glossary**

ACO Aircraft Certification Office

ADC Air Data Computer

AHRS Attitude/Heading Computer

CCB Change Control Board

CCC Central Control Computer

CDU Control Display Unit

CMFD Color Multi Functional Display

CR Change Request

CSCI Computer Software Configuration Item

DER Designated Engineering Representative

ICD Interface Control Document

ISTQB International Software Testing Qualifications Board

Kinemap Third party provider for libraries needed for MAP

LDRA Qualified Low Level Test Tool

MAP SW Application needed navigation purposes

MC/DC Modified Condition / Decision Coverage

OFP Operational Flight Program

ORS Operational Requirements Specification

PSAC Plan for Software Aspects of Certification

PR Problem Report

RTCA Radio Technical Commission for Aeronautics

SAS Software Accomplishment Summary

SCA Software Structural Analyze

SCADE Qualified SW Development Environment tool

SCM(T) Software Configuration Management (Team)

SDD Software Design Document

SDP Software Development Plan

SDT Software Development Team

SIL System Integration Laboratory

SQA(T) Software Quality Assurance (Team)

SRS Software Requirements Specification

SVT Software Verification Team

SVP Software Verification Plan

SW Software

SwRT Software Requirements Team

TEMP Test and Evaluation Master Plan

TAI Turkish Aerospace Industries

VAPS Qualified Graphical User Interface Development Env. tool

VOR/ILS VOR/Instrument Landing System