ECSS-E-HB-40A
11 December 2013

Space engineering

Software engineering handbook

ECSS Secretariat
ESA-ESTEC
Requirements & Standards Division
Noordwijk, The Netherlands

Foreword

This Handbook is one document of the series of ECSS Documents intended to be used as supporting material for ECSS Standards in space projects and applications. ECSS is a cooperative effort of the European Space Agency, national space agencies and European industry associations for the purpose of developing and maintaining common standards.

The material in this Handbook is defined in terms of description and recommendation how to organize and perform the work of space software engineering.

This handbook has been prepared by the ECSS-E-HB-40A Working Group, reviewed by the ECSS Executive Secretariat and approved by the ECSS Technical Authority.

Disclaimer

Published by: ESA Requirements and Standards Division
ESTEC, P.O. Box 299,
2200 AG Noordwijk
The Netherlands

Change Log

ECSS-E-HB-40A 11 December 2013	First Issue

Table of contents

Figures

Tables

Introduction

The ECSS-E-ST-40C Standard defines the principles and requirements applicable to space software engineering. This ECSS-E-HB-40A handbook provides guidance on the use of the ECSS-E-ST-40C.

History of the ECSS-E-40

At the beginning was ESA PSS-05. It was a prescriptive list of requirements ordered all along a waterfall lifecycle. It was necessary to improve it because it was too prescriptive and not flexible enough to apply new technologies such as UML.

ECSS was created in the 90's, and ECSS-E-40A was published in 1999. It was derived from ISO 12207, which is a process model. A process model proposes a set of abstract processes, and the software developer defines its own lifecycle that enters and leaves and re-enters the various processes. The process model was very abstract, with sort of meta-processes that were "invoking" other sub-processes. The new ECSS-E-40A was not prescriptive and very flexible to any kind of lifecycle.

ECSS-E-40A was improved because it was too abstract and it was not clear what had to be done. ECSS-E-40B was worked out in order to downsize the abstraction. The invocation was simplified, some processes were grouped. ECSS-E-40B was sent for public review.

The public review recommended improving further the pragmatic aspects of the standard. Therefore another ECSS-E-40B version was produced where the process model was streamlined.

At a workshop in 2004 on the use of ECSS-E-40B, it was recognised that some of the requirements left room for interpretation, which in turn lead to many discussions in the project reviews (especially when they were overlooked during the Software Development Plan review). Therefore the version C of ECSS-E-ST-40 was produced to improve the usability of the standard, refining and streamlining the open requirements, and somehow coming closer to the ESA PSS-05 spirit.

1
Scope

This Handbook provides advice, interpretations, elaborations and software engineering best practices for the implementation of the requirements specified in ECSS-E-ST-40C. The handbook is intended to be applicable to both flight and ground. It has been produced to complement the ECSS-E-ST-40C Standard, in the area where space project experience has reported issues related to the applicability, the interpretation or the feasibility of the Standard. It should be read to clarify the spirit of the Standard, the intention of the authors or the industrial best practices when applying the Standard to a space project.

The Handbook is not a software engineering book addressing the technical description and respective merits of software engineering methods and tools.

ECSS-E-HB-40A covers, in particular, the following:

a. In section 4.1, the description of the context in which the software engineering standard operates, together with the explanation of the importance of following standards to get proper engineering.

b. In section 4.2, elaboration on key concepts that are essential to get compliance with the Standard, such as the roles, the software characteristics, the criticality, the tailoring and the contractual aspects.

c. In section 5, following the table of content of the ECSS-E-ST-40C Standard, discussion on the topics addressed in the Standard, with the view of addressing the issues that have been reported in projects about the interpretation, the application or the feasibility of the requirements. This includes in particular:

    1. Requirement engineering and the relationship between system and software
    2. Implementation of the requirements of ECSS-E-ST-40 when different life-cycle paradigms are applied (e.g., waterfall, incremental, evolutionary, agile) and at different levels of the Customer-Supplier Network
    3. Architecture, design and implementation, including real-time aspects
    4. Unit and integration testing considerations, testing coverage
    5. Validation and acceptance, including software validation facility and ISVV implementation
    6. Verification techniques, requirements and plan
    7. Software operation and maintenance considerations.

d. In section 6 and 7, more information about selected topics addressed in section 5 such as (in section 6) use cases, life cycle, model based engineering, testing, automatic code generation, and (in section 7) technical budget and margin, computational model and schedule analysis.

NOTE In order to improve the readability of the Handbook, the following logic has been selected for sections 5, 6, and 7:

- section 5 follows the table of content of ECSS-E-ST-40C at least up to level 3 and generally up to level 4. For each sub clause of ECSS-E-ST-40C:
  + either information is given fully in section 5,
  + or there is a pointer into section 6 or section 7
  + or the paragraph has been left intentionally empty for consistency with the ECSS-E-ST-40C table of content, in this case, only " –" is mentioned.
- section 6 expands selected parts of section 5 when:
  + either the volume of information was considered too large to stay in section 5,
  + or the topic is addressed in several places of section 5

  In any case, there is a pointer from section 5 to section 6, and section 6 mentions the various places in ECSS-E-ST-40C where the topic is addressed.
- section 7 follows the same principles as section 6, but gathers the topics related to margins and to real-time.

e. In Annex A, as a complement to the ECSS-E-ST-40C Annex A called Document Requirement List [DRL], the documents expected at the Technical Reviews such as SWRR, DDR, TRR and TRB.

f. In Annex B, software engineering techniques appropriate for the implementation of specific ECSS-E-ST-40C clauses and their selection criteria, covering most of the software lifecycle.

g. In Annex C, an example of the Document Requirement Definition of the Software Maintenance Plan.

2 References

ECSS-S-ST-00-01C	ECSS system - Glossary of terms
ECSS-E-ST-10C	Space engineering – System engineering general requirements
ECSS-E-ST-40C	Space engineering - Software
ECSS-E-ST-70-01C	Space engineering - On board control procedures
ECSS-E-TM-40-07A	Space engineering - Simulation modelling platform
ECSS-M-ST-40C	Space project management - Configuration and information management
ECSS-Q-ST-80C	Space product assurance - Software product assurance
Def Stan 00-54 (March 1999)	Requirements for Safety Related Electronic Hardware in Defence Equipment" Part No: 1: Requirements: Issue 1
ESA ISVV Guide	ESA Guide for Independent Software Verification and Validation, Version 2.0, December 29, 2008
IEEE 610.12-1990 (Jan 1990)	IEEE Standard Glossary of Software Engineering Terminology
IEEE 754-1985 (Aug 1985)	IEEE Standard for Binary Floating-Point Arithmetic
ISO/IEC 12207:2008	Systems and software engineering – Software life cycle processes
NASA Study on Flight Software Complexity (May 2009)	Final Report. NASA Study on Flight Software Complexity. Commissioned by the NASA Office of Chief Engineer. Technical Excellence Program Adam West, Program Manager
RNC-CNES-E-HB-70-501 (September 2008)	Space engineering monitoring and control specification guide, Version 2, September 16, 2008
NASA Lessons Learned	http://llis.nasa.gov/llis/search/home.jsp
NASA System Engineering Handbook	NASA/SP-2007-6105 Rev1 December 2007
ESA PSS-05-0 Issue 2 (February 1991)	ESA software engineering standards Issue 2
Flight Computer Initialisation Sequence	Space Avionics Open Interface initiative document: SAVOIR/12-007/FT http://savoir.estec.esa.int

3
Terms, definitions and abbreviated terms

3.1 Terms from other documents

For the purpose of this document, the terms and definitions from ECSS-S-ST-00-01C and ECSS-E-ST-40C apply.

3.2 Terms specific to the present document

3.2.1 software product:

set of computer programs, procedures, documentation and their associated data

3.2.2 acceptance test

test of a system or functional unit usually performed by the customer on his premises after installation, with the participation of the supplier to ensure that the contractual requirements are met

[adapted from ISO/IEC 2382--20:1990]

NOTE ECSS-E-ST-40C relies on ECSS-E-ST-00-01 for the definition of acceptance test and software product (and consequently of its synonyms "software" and "software item"). However, these two terms have disappeared in its last revision C. The definitions of ECSS-E-ST-40B (and therefore ECSS-Q-ST-80B) are restored here.

3.3 Abbreviated terms

For the purpose of this document, the abbreviated terms from ECSS-S-ST-00-01C and ECSS-E-ST-40C apply.

Abbreviation	Meaning
AR	acceptance review NOTE The term SW-AR can be used for clarity to denote ARs that solely involve software products.
CDR	critical design review NOTE The term SW-CDR can be used for clarity to denote CDRs that solely involve software products.
CMMI	capability maturity model integration
COTS	commercial-off-the-shelf

Abbreviation	Meaning
CPU	central processing unit
DDF	design definition file
DDR	detailed design review
DJF	design justification file
DRD	document requirements definition
ECSS	European Cooperation for Space Standardization
eo	expected output
GS	ground segment
HMI	human machine interface
HSIA	hardware-software interaction analysis
HW	hardware
ICD	interface control document
INTRSA	international registration scheme for assessors
IRD	interface requirements document
ISO	International Organization for Standardization
ISV	independent software validation
ISVV	independent software verification and validation
MGT	management file
MF	maintenance file
MOTS	modified off-the-shelf
OBCP	on-board control procedure
OP	operational plan
ORR	operational readiness review
OTS	off-the-shelf
PAF	product assurance file
PDR	preliminary design review NOTE The term SW-PDR can be used for clarity to denote PDRs that solely involve software products.
PRR	preliminary requirement review
QR	qualification review NOTE The term SW-QR can be used for clarity to denote QRs that solely involve software products.
RB	requirements baseline
SCAMPI	standard CMMI appraisal method for process improvement
SDE	software development environment
SOS	software operation support

Abbreviation	Meaning
SPA	software product assurance
SPAMR	software product assurance milestone report
SPAP	software product assurance plan
SPR	software problem report
SRB	software review board
SRR	system requirements review NOTE The term SW-SRR can be used for clarity to denote SRRs that solely involve software products.
SW	software
SWE	software engineering
TRR	test readiness review
TS	technical specification

4
Introduction to space software

4.1 Getting started

4.1.1 Space projects

Space projects vary widely in purpose, size, complexity and availability of resources. Depending on the intended scenarios, the space projects will cover from small up to very complex developments which may consist of a space segment (e.g. manned spacecraft's like COLUMBUS, unmanned space vehicles like ATV or ARIANE rockets, all kinds of satellites, and payloads within a spacecraft or externally attached residing directly in space), of a launch service segment (e.g. ARIANE Launch complex at Kourou) and of a ground segment (e.g. operations and control centres as well as data evaluation sites).

The production of space systems calls for the cooperation of several organizations that share the common objective of providing a product that satisfies the customer's needs (performance within cost and schedule constraints). This setup requires a strong project management for all involved parties. To base business agreements (e.g. contracts) which define all kinds of applicable standards that are binding for the space project.

4.1.2 Space standards: The ECSS System

In order to enable all experts to "speak the same language" in a space project, a comprehensive set of coherent European Space Standards has been developed as a cooperative effort between the European space agencies and space industries under the heading of **European Cooperation for Space Standardization** (abbreviated: **ECSS**). This set of standards is addressing all essential aspects of the three major domains for the successful implementation of space programmes and projects and is covering all aspects of interest that might appear for the procurement of a generic space product.

Besides general objectives and standardization policies as in the top level documents entitled "ECSS System - Description, implementation and general requirements" (ECSS-S-ST-00C) and terminology "ECSS System - Glossary of terms" (ECSS-S-ST-00-01C) for a space project, the three major domains:

- Space project management,
- Space engineering, and
- Space product assurance

are addressed and their activities are described in a necessary level of detail for application / execution in the form of processes.

4.1.3 Key characteristics of the ECSS System

The ECSS System has four prominent characteristics, which are to provide:

a. a comprehensive and coherent set of standards offering a complete and stable framework within which customers and suppliers at all levels can implement a project in an efficient and cost effective manner;

b. a system of standards constructed in such a way that it can be applied throughout the life cycle of space projects and programmes;

c. a system that can be utilized in activities ranging from small, individual, less complex projects to very large programmes comprising several projects involving many products and interfaces;

d. a controlled method for tailoring the standards such that they can be applied selectively depending on the type, or phase of the project for which they are being used.

As a consequence, the ECSS System focuses primarily on what is required to comply with each standard, rather than how to achieve this. This approach provides the flexibility for different customers and suppliers to use established "in-house" procedures, or processes, to comply with these standards. The ECSS System also includes supporting standards, which identify specific procedures or processes, and Handbooks, which provide technical data and guidelines for procedures and processes. Handbooks are documents providing orientation, advice or recommendations on non-normative matters.

4.1.4 Establishing ECSS Standards for a space project

There is a 7-step process for the a) preparation and b) application of tailoring to establish the ECSS Standards and their requirements for a space project.

a) Preparatory activities:

1) Identification of project characteristics;

   means to identify strategic and technical aspects of the space project.

2) Analysis of project characteristics and identification of risks;

   enforces a close analytical look to significant cost, risk and technical drivers as well as critical issues and specific constraints. These are used to identify and evaluate inherent and induced risks.

b) Tailoring activities:

3) Selection of applicable ECSS Standards;

   comprises to evaluate the ECSS Standards for relevance to the overall space project's needs. Those standards found to be relevant are identified as applicable standards for the implementation of the project.

4) Selection of applicable ECSS requirements;

   Having established the list of applicable ECSS Standards for a project, the extent to which the requirements (contained within these standards) are made applicable is to be assessed against cost, schedule, and technical drivers, as well as against the identified risks and their mitigation strategies.

5) Addition of new requirements;

   Where the applicable ECSS Standards do not include a specific requirement needed for the space project, a new requirement needs to be generated or adopted from an international standard.

6) Harmonization of applicable requirements;

Having completed the selection of applicable ECSS Standards and requirements and the addition of any new requirements, the coherence and consistency of the overall set of requirements to be applied to the space project is reviewed to eliminate the risk of conflict, duplication, or lack of necessary requirements.

7) Documenting ECSS Standards and requirements applicability

The process steps 1 to 6 intentionally do not define a specific format for generating and recording the results of the process. This approach is in line with the ECSS principle of identifying what is required, but not how this is to be achieved, and provides a degree of freedom for customers to select the most appropriate way to present the data within their particular environment.

One method of recording the applicability data in an efficient and structured manner is to consolidate it into an "**Applicable Requirements Matrix**".

Further details on each of these activities are contained in document "**ECSS System - Description and implementation**" (**ECSS-S-ST-00C**).

4.1.5 Software / ECSS Standards relevant for Software

Within a space project "software" plays a key-role with respect to system functionality and operations. It is found at all levels, ranging from system functions down to firmware, including safety and mission critical functions. The Figure 4-1 identifies the main relations of the ECSS-E-ST-40C with the other ECSS Standards.

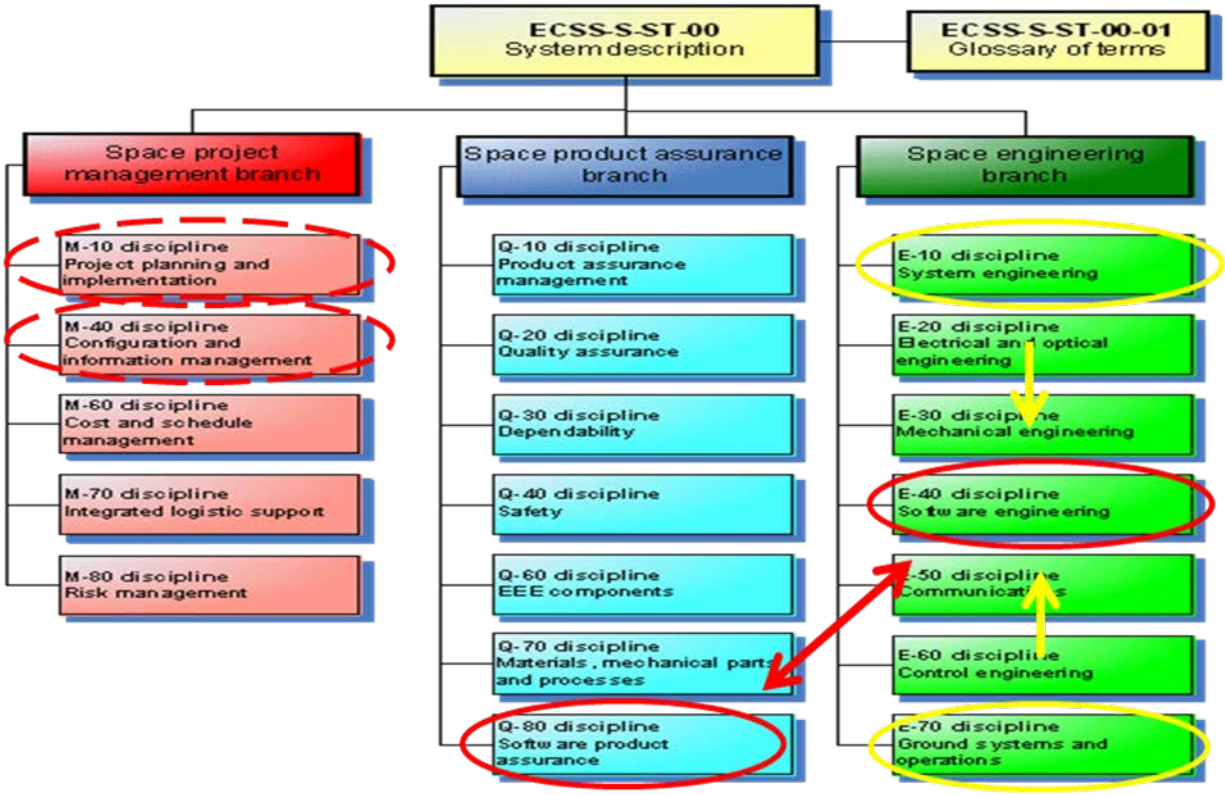


Figure 4-1: ECSS relations for discipline software

The **ECSS-S-ST-00C** document provides a general introduction to the ECSS System and to the use of the ECSS documents in all the space projects and gives therefore background information also for

software projects. The M standards for project management define phases, processes, reviews and rules for space project organization that apply also for software projects. In particular the **ECSS-M-ST-40C Rev1** defines the space configuration and information management requirements also for software projects. In addition, the **ECSS-M-ST-10C Rev1** requirements are tailored for software in the ECSS-S-ST-40 Software Management Process.

The context of the space software engineering activities is the overall Space System Engineering process. System engineering is defined as an interdisciplinary approach governing the total technical effort to transform requirements into a system solution. It's framework is defined by the "**Space engineering - System Engineering General Requirements**" (**ECSS-E-ST-10C**) **Standard** which is intended to apply to all space systems and products, at any level of the system decomposition, including hardware, software, procedures, man-in-the-loop, facilities and services.

The "**Space Engineering Software**" **Standard** (**ECSS-E-ST-40C**) covers the software development of the "Space system product software", i.e. all software that is part of a space system product tree and that is developed as part of a space project needs to apply this standard (for software deliverables and non-deliverables). It focuses on space software engineering processes requirements and their expected outputs. A special emphasis is put in the standard on the system-to-software relationship and on the verification and validation of software items.

This Standard is, to the extent made applicable by project business agreements, to be applied by all the elements of a space system, including the space segment, the launch service segment and the ground segment. It covers all aspects of space software engineering including requirements definition, design, production, verification and validation, transfer, operations and maintenance.

The Standard defines the scope of the space software engineering processes and its interfaces with management and product assurance, which are addressed in the Management (–M) and Product assurance (–Q) branches of the ECSS System, and explains how they apply in the software engineering processes. Together with the requirements found in the other branches of the ECSS Standards, this set provides a coherent and complete framework for software engineering in a space project. Software may be either a subsystem of a more complex system or it may also be an independent system.

In case of developing ground systems the "Space engineering - System Engineering General Requirements" (ECSS-E-ST-10C) Standard is extended by the "Space Engineering - Ground Systems and Operations" Standard (ECSS-E-ST-70C). Ground systems and operations are key elements of a space system and play an essential role in achieving mission success. Mission success is defined as the achievement of the target mission objectives as expressed in terms of the quantity, quality and availability of delivered mission products and services within a given cost envelope. Mission success requires successful completion of a long and complex process covering the definition, design, production, verification, validation, post-launch operations and post operational activities, involving both ground segment and space segment elements. It involves technical activities, as well as human and financial resources, and encompasses the full range of space engineering disciplines. Moreover it necessitates a close link with the design of the space segment in order to ensure proper compatibility between these elements of the complete space system. - Another specific link is made between ECSS-E-ST-70-01C and ECSS-S-ST-40C for On-Board Control Procedures (OBCP) / On-Board Application Programs (OBAP) development and validation processes.

The Space Engineering Software Standard is always complemented by the **Space Product Assurance Standard** (**ECSS-Q-ST-80C**), which specifies the product assurance aspects and is the entry point for ECSS-E-ST-40C into the Q-series of standards. All the requirements of the ECSS-E-ST-40C and ECSS-Q-ST-80C are applicable to the software development projects as a principle, unless criticality driven tailoring (or additional tailoring as per Annex R of ECSS-E-ST-40C for details) or other specific characteristics and constraints driven tailoring is applied. - Requirements for space configuration and information management are in **ECSS-M-ST-40C Rev 1**, which includes the software Document

Requirements Definition (DRD) for the software configuration file. However, the DRDs in the annexes are provided to help customers and suppliers with the project's setup. Together, both these standards either define or refer to the definition of all relevant processes for space software projects. ECSS-Q-ST-20C is the reference, through ECSS-Q-ST-80C, for the software acquisition process, and the software management process tailors ECSS-M-ST-10C for software.

4.1.6 Why are standards a MUST for the software development process?

Different groups of experts (disciplines) contribute to the development of a space project. A fundamental ECSS principle is the so-called "**customer–supplier**" relationship. All space project actors are either a customer or a supplier, or both. In its simplest form, a project can comprise one customer with just one supplier; however, most space projects comprise a number of hierarchical levels, where:

a. the actor at the top level of the hierarchy is the top level customer,

b. the actors at intermediate levels of the hierarchy are both supplier and customer,

c. the actors at the lowest level of the hierarchy are suppliers only.

The customer is, in the general case, the procurer of two associated products: the hardware and the software for a system, subsystem, set, equipment or assembly. The concept of the "customer–supplier" relationship is applied recursively, i.e. the customer can be a supplier to a higher level in the space system. Examples for customers are therefore: European space agencies (ESA, CNES, DLR …) and prime contractor space companies (e.g. for major space projects like COLUMBUS, ATV, ARIANE rockets …) and for suppliers: companies or contractors from industries and from universities contributing by developing parts of the space project. **The fundamental principle of the "customer–supplier" relationship is assumed for all software developments.**

Within the space project, exchanges of products and services are governed by **business agreements**, used as a generic term throughout the ECSS Standards when referring to a legally binding agreement between two or more actors in the customer–supplier chain. These agreements include the terms and conditions agreed between the parties, the rules by which business is conducted, the actors' commitments and obligations for the provision of goods and services, the methods of acceptance and compensation, monetary, or otherwise. Business agreements serve as a framework prescribing the activities throughout the execution of work, and as a reference to verify compliance. Business agreements are recorded in a variety of forms, such as:

a. Contracts

b. Memoranda of understanding,

c. Inter–governmental agreements,

d. Inter–agency agreements,

e. Partnerships,

f. Bartering agreements, and

g. Purchase orders.

The Space Engineering Software Standard is intended to help the customers to formulate their requirements and the suppliers to prepare their responses and to implement the work. Due to its nature it will help to get a complete view of the software from management point-of-view, to clarify the software activities, to focus the software effort according to the space project's context, and to verify the completeness of the intended statement of work. Furthermore the customer will be able to evaluate different proposals by checking the bidder's detailed compliance with the requirements.

The Space Engineering Software Standard may be adapted, i.e. **tailored** for the specific characteristics and constraints of a space project in conformance with ECSS-S-ST-00C. There are several drivers for tailoring, such as dependability and safety aspects, software development constraints, product quality objectives and business objectives. Tailoring for dependability and safety aspects is based on the selection of requirements related to the verification, validation and levels of proof demanded by the criticality of the software. So a number of factors have more or less influence on the tailoring: project characteristics, costs for the development and the operations & maintenance phase, number and skills of people required to develop, operate and maintain the software, criticality and complexity of the software, risk assessments. The type of software development (e.g. database or real-time) and the target system (e.g. embedded processor, host system, programmable device, or application specific integrated circuits) needs also to be taken into account. The tailoring of ECSS standards has direct influences to the business agreement of the space project (due to the binding clauses).

4.1.7 Executing a space software project

4.1.7.1 Overview

After successful negotiations a business agreement will be awarded from a customer to a supplier. The supplier then needs to implement his specific in-house organization for the space software project by assigning appropriate resources for the development of the "Space system product software" in the Project Management, Engineering and Product Assurance.

4.1.7.2 Lessons learned

Lessons learned show that spending the right level of effort already in the Development Phase will pay off at the end, e.g. in the mid- / long-term operations. If this concept is not implemented adequately, the efforts to be spent for corrective actions will be much higher when a problem is detected in a later phase, and are thereby becoming less cost-effective.

4.1.7.3 Project Management

The space software project is managed as a set of processes and will involve the customer (representatives) at main interaction points. These points are called: **reviews**. Their purpose is to synchronize software engineering processes.

Software management consists of supervising and monitoring the development of the software. It ensures that the correct procedures are followed, that the software is delivered on schedule and to budget, and that the final products are of the expected quality, e.g. they are correct, safe, efficient and easy to maintain or modify as required.

The main activities are the production of the **software plans** by defining and instantiating the particular implementation of the applicable software standards in the space software project. The plans include the life cycle description, activities description, milestones and outputs, the techniques, methods and tools to be used, and the risks identification (*not exhaustive list*).

4.1.7.4 Reviews

The reviews relevant to the software engineering processes are defined in detail in the "**Space Project Management - Project Planning and Implementation**" Standard (**ECSS-M-ST-10C Rev1**) as:

a. System Requirements Review (SRR)

b. Preliminary Design Review (PDR)

c. Critical Design Review (CDR)

d. Qualification Review (QR)

e. Acceptance Review (AR)

f. Operational Readiness Review (ORR).

The Space Engineering Software Standard offers in addition the possibility to anticipate the PDR in one ***Software Requirements Review (SWRR)*** or more, and the CDR into one ***Detailed Design Review (DDR***) or more. The reviews occur at different levels in the customer–supplier hierarchy.

When the software development is included in a complete space project, the software engineering processes have also a relationship with the project phases (0, A, B, C, D, E, F) as the system level will induce driving constraints, such as availability of inputs (specifications, interface definitions), potential changes, delivery needs, schedule constraints, planning for system level reviews, etc. For software developments which are performed out of the context of a single space mission because they are developed for reuse, e.g. cross missions ground segment infrastructure , there is no particular link to phases of a space mission project. However, when the software is reused for a project, only mission specific elements are reviewed. The reused process is described in ECSS-Q-ST-80C.

The management process also includes the organization and handling of additional **joint reviews** (project and technical reviews for software), the definition of procedures for the management of the system interface, and the technical budget and margin management.

4.1.7.5 Processes

The notion of engineering processes is fundamental in the Space Engineering Software Standard (ECSS-E-ST-40C). These processes provide the means to describe the overall constraints and interfaces to the engineering processes at system level. At the same time, they provide the possibility to the supplier to implement the individual activities and tasks implied by the processes in accordance with a selected software life cycle.

The Standard is a process model, and does not prescribe a particular software life cycle. Although quite some space project reviews suggest a waterfall model, the software development plan can implement any life cycle, in particular by the use of the technical reviews whose formalism is setup less formal than the project reviews.

ECSS-E-ST-40C Figure 4-2: Overview of the software life cycle processes identifies the main concurrent software engineering processes and shows how and when they are synchronized by the customer-supplier reviews.

The relations between the software and the system reviews are addressed in chapter 5.2.5 in more detail.

4.1.8 Disciplines in Space Software Projects

4.1.8.1 Overview

The term “Discipline” is defined in ECSS-M-ST-10C, as “a specific area of expertise within a general subject”. The name of the discipline normally indicates the type of expertise in order to conduct subsequent software processes (for management, engineering and product assurance).

A discipline is associated with a set of related tasks to be carried out by an individual. A software expert of a discipline may carry out one, or several, development activities. Traditionally, these software experts correspond very closely to activities, so that, for example, an analyst prepares the software requirements, a designer prepares the architecture, and a programmer makes detailed design and coding. Nowadays one single expert may carry out all these three activities or even more, e.g. by using methodologies and toolsets.

The descriptions of the disciplines in the next (sub)sections are not meant to be exhaustive.

4.1.8.2 Software System Engineering (Co-engineering)

The Software System Engineering discipline contributes to the Space System Engineering process and covers tasks such as requirements engineering, system analyses and consolidation of requirements (by various analyses and trade-offs), preparing functional and physical architectures, system design and configurations, and defining (contractual) interfaces, system verification (to be conformant to requirements) and finally system engineering integration and control (supervising the construction of the computer programmes to meet all pre-conditions precisely). Due to the combination and the complementary work this process is regarded as "co-engineering".

4.1.8.3 Software Design and Implementation

In the Software Design and Implementation the detailed structure of the software functions is thought out. The software ultimately consists of components such as programs, processes and procedures, organized to provide functions to the space system. Designs very often consist of hierarchies of components. The software design and implementation include the architectural and detailed design, its documentation (in DDF and code) and justification (in DJF), and the coding, unit testing and integration testing of the components. In software integration and testing, software components, assemblies, and systems are fitted together and tested in a disciplined and rigorous way to ensure that the finished system is as nearly correct as possible.

4.1.8.4 Software Configuration Management

Software Configuration Management is the process of ensuring that the delivered software is complete and forms a set of consistent elements. This is achieved by restricted access to the set of deliverable items, such as programs and documents that constitute each issue or version of the system. A disciplined sequence of activities is enforced when a change is required to a configuration item.

4.1.8.5 Software Product Assurance

Software Product Assurance is a set of processes of ensuring that the software is fit for use and is built in accordance with applicable project requirements. This includes the software characteristics of safety, reliability and maintainability. Fitness for use is achieved by rigorous control (inspections, audits and reviews), by analyses, by witnessing tests, and by ensuring that correct procedures are followed.

4.2 Getting compliant

4.2.1 The ECSS-E-ST-40C roles

4.2.1.1 Customer – Supplier definition

In accordance with the ECSS theoretical concept, a fundamental principle of the ECSS-E-ST-40C standard is the 'customer–supplier' relationship, assumed for all software developments. The project organization is defined in ECSS-M-ST-10C. The customer is, in the general case, the procurer of two associated products: the hardware and the software for a system, subsystem, set, equipment or assembly. The concept of the "customer–supplier" relationship is applied recursively, i.e. the customer can be a supplier to a higher level in the space system. The software customer therefore has two important interfaces:

a. the software customer interfaces with his software and hardware suppliers in order to adequately allocate to them functional and performance requirements, through the functional analysis.

b. the software customer assumes a supplier role to interface in turn with his customer at the next higher level, ensuring that higher level system requirements are adequately taken into account.

The customer main activities are recalled hereafter as a synthesis of the ECSS-E-ST-40C standard, the supplier role is fully defined in the standard and it is not summarised in this introduction

The customer derives the functional, performance, interface and quality requirements for the software, based on system engineering principles and methods. Software items are defined in the system breakdown at different levels. . The customer's requirements are specified by this process, and they provide the starting point for the software engineering.

As the customer supplier relationship is a formal relationship (e.g. contractual when the customer supplier are not in the same organisation), the customer requirements baseline (RB) should be completed by a SOW , to precise the commitment between the two parties and the scope of the activities to be performed by the supplier such as phases to be covered (development, operation , maintenance) , Customer Furnished Item (CFI) , schedule constraints , etc. …

The customer activities are defined in the ECCS-E-ST-40C clause 5.2

When the first level SW supplier subcontracts SW activities to a third party, only those requirements of the ECSS-E-ST-40C standard that pertains to the subcontracted activity are applicable to the third party. The first level SW supplier nevertheless remains responsible of the ECSS-E-ST-40C conformance towards the Customer.

If the customer considers that it is necessary, the customer has to tailor the ECSS standard in the scope of its project.

Non-compliances to the standard or the tailored standard are agreed before the kick-off meeting.

In additional to the formal reviews , the customer and supplier may agree on additional technical reviews to address specific concerns as anticipation of formal reviews, technical focus , etc. (ECSS-E-ST-40C clause 5.3.3)

The customer has in charge to perform the acceptance activities (section 5.7.3)

The customer specifies which part of operation and maintenance activities are to be covered by the SW supplier in the SOW. Even if not explicitly covered by the C/D phase SOW, maintenance process is considered by both customer and supplier in order to ensure that the prerequisites for maintenance phase have been correctly taken into account.

4.2.1.2 User

The ECSS-E-ST-40C introduces the term "user".

The term "user" in ECSS is used in section 5.9 with the introduction of the SOS entity in charge to support the user during the operations. The user is defined in this case as the "final user" of the SW product, and so it is not the customer in the development phase, in the major cases. The final SW user is the operator of the system in which the software is embedded

The ECSS-E-ST-40C mentions in others requirements the notion of use case, user need, etc., which are related to all SW users during the different system integration phases.

It is the Customer responsibility to collect all the user's needs, and in particular the Use Case definitions, which define early the operational constraints. The customer must include the verification of the user's needs in the acceptance process.

4.2.1.3 The SOS Entity

The SOS (Software Operation Support) entity is responsible to support the Software Operation process as introduced in ECSS-E-ST-40C §4.2.9 and as defined in ECSS-E-ST-40C §5.9

The SOS entity main objective is to keep the software operational for the final user. The SOS entity plays a role only after the system Acceptance Review, and acts as a relay between the final user and maintainers, in charge of testing the operational software behaviour in the operational context, tracking user requests, deploying new releases when necessary, ensuring administration activities.

4.2.1.4 Maintainer

The maintenance process contains the activities and tasks of the maintainer. The objective is to modify an existing software product while preserving its original properties. This process includes the migration and retirement of the software product. The process ends with the retirement of the software product.

The activities provided in section 5.9.1.2 are specific to the maintenance process; however, the process utilize other processes in ECSS-E-ST-40C and the term supplier is interpreted as maintainer in this case.

The maintainer manages the maintenance process applying the management process.

4.2.1.5 Operator

The term "operator" appears inside the ECSS-E-ST-40C only:

a. in the clause 5.10.6.2 for the establishment of the migration plan in collaboration with the maintainer

b. in the clause 5.10.7.1 for the establishment of the retirement plan in collaboration with the maintainer

NOTE The responsible for these two plans are the maintainer and not the operator.

The operator is used to express the final user not only of the software product but also of the system which include the software product.

4.2.1.6 Conductor

The term "conductor" is introduced for the ISVV activities. The conductor is the person or the entity that takes in charge the verification and validation tasks of the ISVV. It is the ISVV supplier.

4.2.1.7 Roles relationship

All the roles introduced in the ECSS-E-ST-40C standard are not strictly complementary, but can have some overlaps. This paragraph present the relationship between these roles , that are classified at different level (SW level , system level, user level) and which are introduced in perspective with the different program phases

According to the different space project phases, the role is dispatched on these phases as follows:

a. during SW development phase until the SW AR :

1. the SW customer is in charge to specify the requirement baseline , to procure the SW and to accept the SW product

2. the SW supplier is in charge to develop, to validate the product. The SW supplier is in direct relationship with the SW customer

3. the ISVV conductor is present when an ISVV is required. The ISVV conductor is a third party w.r.t. the software customer and the software supplier. It can be in direct relationship with either of them, provided that the independence and authority are guaranteed according to ECSS-E-ST-40C requirement 5.6.2.2.

4. the SW final user is in fact the system operator. As a major concern, the customer is in charge to make sure that the needs of the final user are properly captured in the requirements baseline, to avoid late introduction of operational constraints.

b. between the SW AR and the system AR :

1. the SW customer is in charge to integrate the SW product inside the system. It will be supported in this phase by the SW maintainer for expertise , analysis and correction or modification if it is necessary

2. the SW maintainer is in charge to support the SW customer. The SW maintainer performs expertise and analysis on SW customer request, to correct the SW product, to implement modification or evolution on SW customer request. In general the SW maintainer is the SW supplier of the previous phase, but this responsibility can be translated to a third party.

3. ISVV activities can be required according to modification or evolution requests and their impact on the critical parts of the software.

4. the SW final user is in fact the system operator. There is no change with regard to the previous phase.

c. After the system AR :

1. the SW final user who is in fact the system operator, is in charge to operate the system

2. The SOS entity role appears in this phase and covers in fact two roles: assure the administration and the maintenance of the operations infrastructure, and assure the role of system maintainer. The SOS entity is in direct relationship with the SW final user (system operator)

3. The SW maintainer is in charge to support the SOS entity. The SW maintainer performs expertise and analysis on SOS entity request, to correct the SW product, to implement modification or evolution on SOS entity request.

The role relationship are synthesised in Figure 4-2.

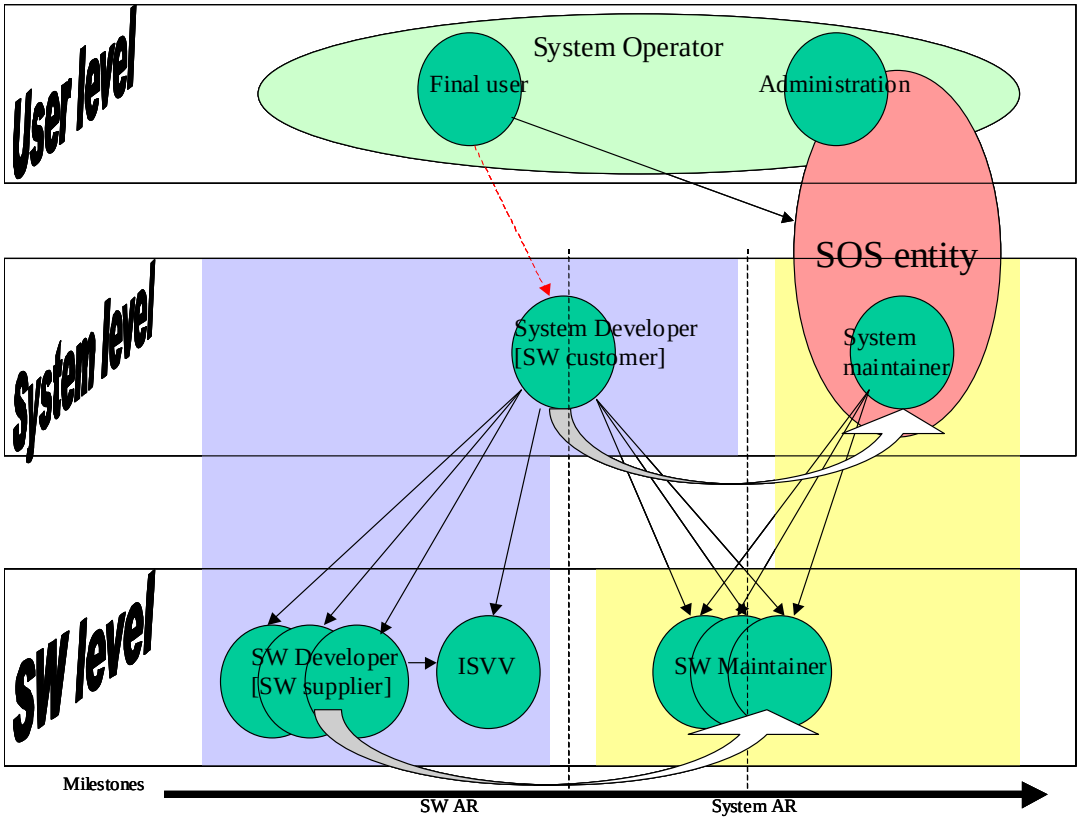


Figure 4-2: Role relationships

4.2.2 Compliance with the ECSS-E-ST-40C

The following text is extracted from ECSS-S-ST-00C:

The ECSS documents themselves do not have legal standing and they do not constitute business agreements: they are made applicable by invoking them in business agreements, most commonly in contracts. The applicability of standards and requirements is specified in the project requirements documents (PRDs), which are included in business agreements, which are agreed by the parties and binding them.

NOTE Description of PRD is provided in ECSS-M-ST-10, Clause 4.1.10.

The top–level customer's PRD forms the basis for the generation of all lower level customer PRDs. An integral part of a PRD, at any level, is the set of ECSS Standards tailored as necessary and documented in an "ECSS Applicability Requirements Matrix" (EARM), as described in clause 7.3.5.

A supplier, at any level, is responsible for demonstrating compliance with the project requirements contained in his customer's PRD, through, for example, the elaboration of a compliance matrix, and ultimately for supplying a conforming product. The compliance to the PRD is presented in an Implementation Documents (IDs), for example project plans (e.g. management, engineering and product assurance) and compliance matrix.

a. The supplier shall demonstrate compliance with the PRD requirements.

NOTE An "ECSS compliance matrix" (ECM) is a recommended method to document the demonstration of this compliance. The ECM is part of the project compliance matrix, addressing compliance to the applicable ECSS requirements (e.g. EARM).

b. The documentation identifying the compliance to ECSS requirements applicable to the project (e.g. the ECM) shall include following data:

1. The complete list of ECSS requirements applicable to the project (e.g. in the EARM).

2. For each requirement, the actual indication of compliance. When deviation is identified the justification is provided.

An example of template for an EARM for the requirements of ECSS-S-ST-00C is the following table:

Identifier	Requirement	Applicable (A/M/D/N)	Modified requirement

where:

applicable without change (A)

applicable with modification (M)

not applicable (D)

new generated requirements (N) including identification of origin if existing

According to ECSS-S-ST-00C, the customer provides an ECSS Applicability Requirements Matrix and the Supplier replies with an ECSS Compliance Matrix. It is important that the ECSS Compliance Matrix is done at the level of each requirements (as opposed to a global statement of compliance), in order to allow the Customer to detect early enough in the project the Non or Partial Compliance. It is essential to discuss them at the beginning of the project rather than discovering them at the end.

The attention of the Supplier is drawn to the risk taken by declaring systematically (global) compliance in the proposal. Although this might look like a way to increase the chances of being selected, it will likely backfire on the Supplier and the project when it will actually be discovered that this global compliance cannot be achieved.

A Partial Compliance needs to be detailed such that the Customer can assess the extent to which the objective of the ECSS requirement is covered, and whether a different way to achieve the objective might be acceptable.

A Non-Compliance needs also to be investigated in terms of feasibility and acceptability in the scope of the project.

There are historical records of PC/NC that ended up being acknowledged in an update of the ECSS-E-ST-40C Standard, such as the structural coverage of code achieved only by unit tests (version B) or through all the test campaign (version C, Note in 5.8.3.5b) or delivery of detailed design, not as a document (version B), but as an electronic file (version C, Ye in Annex R; accounting of models in 5.3.2.4).

When a Supplier has his own internal software engineering standard (e.g. as part of a company quality model), he can negotiate with the Customer the approval of a compliance matrix between the ECSS-E-ST-40C standard and the company standard, provided that all the information required by ECSS is actually delivered. This is in particular the objective of ECSS-E-ST-40C clause 5.3.9.2 (Documentation compliance) to allow a mapping of the ECSS DRDs on company documentation.

This is done today at each project level. However, it would be beneficial to put in place a customer/supplier company level agreement, although there is no today formalized organisational structure to do so.

4.2.3 Characterization of space software leading to various interpretations/applications of the standard

4.2.3.1 Introduction

Software is found in the space system at all levels, ranging from system functions down to firmware, including safety and mission critical functions.

ECSS-E-ST-40C is applicable for the development of any kind of software in the space domain, including software developed in R&D projects, in order to prove new technologies and novel concepts, or algorithms, prototype software that can be used in performing mission trade-offs, in designing the HCIs. or with the sole purpose of debugging an operational implementation.

Usually the space software is distinguished between flight and ground segment software.

In addition to the criticality levels, the following software characterisation is typically used for further tailoring of ECSS-E-ST-40C.

4.2.3.2 Flight software characterization

The overall aspects that may characterize flight software projects are:

a. Nature of carrier: Spacecraft, Launcher, Micro Gravity, Space station,

b. Nature of application: Command & Control, Data Processing, MMI, Real time, Security

c. Nature of development process: From scratch, Reuse, Prototyping or R&D

d. Criticality of Software: category A, B, C, D,

e. Programmatic: Governance, schedule, Risk, Cost

Various application families have been identified in the space flight software domain:

a. command and control software (on board of satellites, probes, spacecraft, but also command and control for payloads, control software of vehicles, hardware command and control, equipment, sensor, actuator software, up to the control software of robots and the central computers of space stations). They generally take care of control and data handling, and feature a high level of criticality. The focus is put on the integrity of data, which impacts on the design constraints and the technology used.

b. data processing software (on board satellites, probes, spacecraft and robots). The software processes the payload data and can be less critical. The focus is put on the performance (e.g. image processing).

c. micro-gravity facility and experiment software. It is generally a self-contained (software) system which includes command, control and data handling software for the check-out, error, FDIR, time, mode, TM/TC and resources management and local data processing. Its failure, by design, does not affect the whole space infrastructure.

d. control MMI software is generally running on a laptop for developing and running the experiment procedures and the various sub systems controllers, for managing concurrency and resources access. Although operating in space, its characteristics are closed to the commercial desktop software, and similar technologies and criteria can be used.

Launcher software (flight control software and equipment software) has a short operational life combined with high availability and reliability. Furthermore it runs once and cannot be tested in the real environment.

Flight software includes low level software such as boot, initialisation, board support package, that often cannot be modified during operation.

Flight software runs in a hostile environment. A failure of the flight software that results in the non-acceptance of commands can lead to the loss of the spacecraft. Software maintenance is more difficult on-board as it is difficult to access and is usually required continuing operation. In case also of a remaining fault within flight software, investigations are difficult to conduct due to less observability than on ground. This leads to specific dependability requirements.

Furthermore during operation, the spacecraft visibility can be limited and the spacecraft can be hidden from the ground control (by the earth itself or by a planet). This limits the access time and can lead to specific requirements for the autonomy of the spacecraft.

The operational lifetime of a spacecraft is generally long (in the range from 2 to 15 years). This puts specific constraints on the maintenance of the flight software development environment, as in most of the cases the on board software is the only spacecraft component that can be modified during in-orbit life. This allows correcting malfunctions in the software, but also to circumvent HW problems or to adapt to not-predicted situations in-orbit. The software development environment needs also to be working during all the operational lifetime of the spacecraft, this may require specific measures such as hardware support provisions or emulation on new hardware.

4.2.3.3 Ground segment software characterization

The overall aspects that may characterize ground software projects are:

a. Deployment environment: Dedicated systems, Cloud, Web-based system, Mobile & Embedded,

b. Nature of application: Command & Control, Data Processing, MMI, Real time, Security

c. Nature of development process: From scratch, Reuse, Prototyping or R&D

d. Criticality of Software: category A, B, C, D,

e. Programmatic: Governance, Schedule, Risk, Cost

The ground segment software is software that is part of the ground system and is therefore used for operating and exploiting the space segment. The ground segment software can be distributed across various subsystems depending on the ground segment architecture:

a. Mission operations system, including (not extensive list) the Mission Control System, Flight Dynamics Systems, Mission Planning, Operational Simulators and Data Distribution Systems.

b. Payload operations and data system;

c. Ground station system;

d. Ground communications system.

Ground segment software may be composed of critical, real-time, near real time (data driven payload data segment) and non-real-time (e.g. batch processing, mission planning, data processing and archiving) components. Some elements of the ground segment may stay operational even 5 to 10 years after the end of the satellite lifetime (e.g. the data archive systems), so it may have a long lifetime, including maintenance.

Many elements of ground segment software such as the Mission Control System and the operational simulator are often developed based on an existing, generic and mission independent infrastructure. Most projects build on the out-of-the-box functionality provided by the infrastructure and customise it for addressing mission specific needs. New software development projects (from scratch) are therefore quite unusual in the traditional ground segment domain. The heavy reuse of generic infrastructure in the ground segment, resolves many system engineering design aspects at a generic and mission

independent level (for a group of similar space missions with common characteristics). In fact the utilisation of the infrastructure software is not limited to reuse of software functionality but more importantly to re-use of a system-design at ground segment level. In other words the subject infrastructure can be seen in many cases as a reference ground segment system architecture, which defines the composing components of the ground segment software and establishes standard interfaces among them. Hence when developing the components of the ground segment for a particular mission (software systems such as the Mission Control System, Flight Dynamics software systems, operational simulator or the ground station software), the software development does not need to start from the system related software requirements engineering phase (the system context is established through the reuse of system design provided by the infrastructure).

The ground segment infrastructure is generic and covers the common requirements of many missions. For specific families of missions such as EO or deep-space missions, which share additional common requirements, special profiles of the infrastructure may be developed to increase the reuse even more. From this infrastructure perspective, many developments are not driven by the requirements of a single space mission and are not implemented as part of a particular space programme, but have their more or less independent and parallel lifecycle. These infrastructure developments are planned ahead of the missions (e.g. new versions of the mission control system or simulator infrastructure). Therefore some system requirements are not fully stable. Performance requirements, in particular, need to be forecasted or maximized;

The lifecycle of ground segment software elements is often more complicated than that of the flight software. It is not unusual that the different components of the ground infrastructure are developed by different Suppliers, maintained by yet other Suppliers and provided as Customer Furnished Items to the Suppliers who are responsible for the development of the mission specific software components (which again may be maintained by other Suppliers). The fact that more than one mission use at same time components of common infrastructure software, adds additional complexity to the governance of the infrastructure software lifecycle. To give an example the management of software defects on the infrastructure software can only be governed by configuration change boards, involving representatives of all software projects, reusing the subject software. These characteristics impact directly many aspects of software development life cycle, including the design, implementation, validation, maintenance and operation.

Components of ground software undergo an operational validation, which may be performed, depending on the schedule, before or after the individual software development project is finally accepted (AR), e.g. after the launch of the satellite or in the context of simulation campaigns;

The overall Ground Segment composes typically significantly larger number of complex systems in comparison to flight software (from the ground station to the mission control system to data distribution system and payload operations planning system, internet connectivity units such as routers and firewalls, server and PC operating systems, databases, …). Each of these systems is typically deployed in a geographically different location and operated by different entities. The process of end-to-end validation of such a distributed system can therefore usually be achieved only at system level and scenario based.

The Ground Segment (GS) often relies on commercial software such as operating systems and COTS. More-over with emerging software development technologies such as web-based applications, Java EE, .NET, SMP-2 based simulator model developments or the principles of Service Oriented Architectures, the GS software becomes more and more developed based on **frameworks** related to these technologies. The validation of these frameworks and COTS is often not a realistic approach. In particular as the COTS and framework are often not available at source code level and limited information regarding their test coverage is available.

It is worth noting that it is not unusual for ground software components developed based on such frameworks, that a significant percentage of the overall functionality of the "system" is provided by the framework (this can reach in some cases 90%). The flight software is in contrary often a custom development fully available at source code level.

The development of parts of the ground segment infrastructure (e.g. routers and gateways) is not directly managed by the ground data system developer and the operating entity (e.g. network) the level of quality of these units can only be assessed at contractual level (service level agreements SLA) or via system testing.

The human factor is an important factor in Ground Segment software, since it is very often manually or semi-automatically operated. A typical characteristic of the Ground Segment software is the provision of Man-Machine-Interfaces. Hence validating the software against all possible combinations of human interactions with it through the Man-Machine-Interfaces (flow of all possible inputs) is often unfeasible.

The Ground Segment software Error control mechanisms are above the level of software only. Examples: critical commands must be sent twice. CRC codes ensure that commands are not corrupted. In case of major problems the spacecraft will enter in safe mode, alerts are verified by humans conducting flight operation procedures;

Most ground segment software problems impact in the worst case the availability of the system. Ground segment software can however usually be corrected and redeployed much easier than flight software. This is for instance often related to much easier access to the system for debugging and redeployment (no need for satellite pass, change of hardware components, extend the resources, e.g. memory, storage). If the maintainability is high (i.e. short time is required to correct failures), there are margins to increase availability of the software i.e. the probability that the software system is available.

Suitability of Ground Segment software for a specific project may be claimed based on product service history, provided that conditions for product service history application are complied with. Most software routines are taken from previous versions of the system. It is particularly the case for complex systems such as flight dynamic software, which may have been developed over the years and are validated operationally (at system level) for a number of missions but not at unit and integration test level and there may provide limited test coverage information.

Ground software is normally

a. data driven (e.g. the behaviour depends on the specific TM/TC data being received/generated and the input entered by the user via the MMI);

b. configuration driven (IP addresses and ports, database adapters, WS endpoints, all options existing in the TM/TC database definitions);

c. human driven (with multiple users, multiple applications, multiple operational modes, etc.).

4.2.4 Software criticality categories

4.2.4.1 Software criticality analysis

The criticality of a software product is determined (ECSS-Q-ST-80C Annex D) at system level, based on the severity of the consequences of the system failures that the software can cause.

The software criticality category is the only criterion used to define a pre-tailoring of ECSS-E-ST-40C in its annex R.

Other characteristics of software, are not used to select the criticality for pre-tailoring. For example:

a. functional requirements for classes of missions or

b. non-functional requirements such as target platform (real-time or embedded vs. workstation)

c. role in a system (flight, operation, testing)

The process leading to the determination of the software criticality is described in ECSS-Q-HB-80-03A. Based on software-level analyses, the individual components of the software product, may be classified at a lower criticality category, provided that no failure propagation is possible from lower-criticality components to higher-criticality components.

The consideration of typical software failure modes (see ECSS-Q-HB-80-03A section 6.2.2.2: functionality not performed, wrongly performed, performed with wrong timing (sporadic or persistent) suggests that the most critical components are found in the ones allowing, for example:

a. to start the software (boot, initialization)

b. to command and observe the software

c. to put the system in a safe state in case of failure

d. to remotely modify the software.

or components whose design or role inside the architecture increases the risk of fault propagation (e.g. operating system, drivers, interrupt handlers, board support package)

To avoid failure propagation, techniques based on fault containment such as Time and Space Partitioning or On Board Control Procedure interpreter, can allow the coexistence of different criticality level components in the same software.

4.2.4.2 Critical software

Critical software is defined in both standards as software of criticality category A, B or C.

Therefore category D represents non-critical software.

Any software subject to ECSS-E-ST-40C is at least category D. Any further tailoring applied to the Category D pre-tailoring cannot be justified by criticality. It may be justified by other criteria, such as the risk that the project is prepared to take.

For software not directly involved in the operation of the system, but rather in its testing (such as simulators or software validation facilities), it is important that its criticality is also determined based on the indirect consequence of its failures on the system. In particular, it is assessed whether

a. a failure of a simulator can lead to a wrong system requirement having a system consequence, or

b. a software testing environment failure can lead to the non-detection of system non-compliance having a system consequence, or

c. these failures can be detected by other means in the system life cycle

For software developed iteratively in several contracts, with the aim of increasing progressively its maturity from prototype to a reusable product (e.g. R&D), the tailoring of the software engineering standard in each iteration depends both on its criticality (the expected criticality of the function for which the software will eventually be used, when the foreseen maturity is achieved) and on its level of Technology Readiness.

For example, early prototypes could be developed according to Cat D tailoring, but need to be re-engineered afterwards at the right criticality level of their intended use.

4.2.5 Tailoring

The contract perimeter is defining the extent to which the ECSS-E-ST-40C is applicable, because some activities such as operation, maintenance, retirement, or ISVV (for criticality category A and B), may not be included in the contract. But this is more related to "applicability" of the Standard rather than to its "tailoring".

In the same spirit, if activities of ECSS-E-ST-40C are done by another role than the one specified, this will also lead to tailoring, for example when system/customer tasks are delegated to software/supplier. i.e. not all of clause 5.2 may be done by the Customer, or the Supplier may be responsible for the acceptance test plan, or the Supplier may not be responsible for the definition of the external interfaces.

The ECSS-E-ST-40C introduces in Annex R a pre-tailoring based on software criticality.

These pre-tailoring may be further tailored according to criteria different from criticality, mainly according to the level of risk which is taken by not performing given engineering activities, or doing them differently than what is specified in the Standard.

It should be considered that taking a risk on high criticality software may have consequences much more important than taking a risk on non-critical software.

On the other hand, activities not required for non-critical software (e.g. higher code coverage, ISVV) may be performed for other reasons, such as intended reuse in a critical context. For example: (non-exhaustive)

a. merging the requirement baseline and the technical specification increases the risk of losing the customer or supplier standpoint, i.e. to miss a use case or to miss an implementation requirement.

b. skipping joint reviews, increases the risk to discover late disagreements between customer and supplier on the product capability or quality, causing substantial reengineering

c. not managing technical budgets and margins, increases the risk to discover unfeasibility late in the project

d. not using design methods, increases the risk to develop weak architectures and inconsistent designs

e. not defining interfaces, increases the risk of integration issues

f. not documenting the detailed design, increases the risk to lose control on the software development such as capability to anticipate implementation errors, to debug, to integrate, to maintain, to master safety and dependability, etc.

g. skipping unit tests, increases the risk to discover faults late in the process and to jeopardize the schedule.

h. not rerunning the full validation tests on the last version of the software, increases the risk to leave faults in the product

i. not performing full verification activities, increases the risk to affect the quality of the product

j. not complying with DRDs content, increases the risk of having non-complete documentation, and of missing information for maintenance

k. non-compliance with the DRDs structure, increases the effort of the reviewers

It is emphasized that the risk taken does not affect only the project management, but also the dependability and safety of the product, i.e. adequacy to its criticality level.

Each risk taken by tailoring out a software engineering requirement is analysed in the scope of the project, and its consequences are assessed to decide if it is acceptable or not.

Any further tailoring is therefore associated to a risk management, which is specific of each project. For this reason, it is not possible to propose other generic tailoring tables.

However, there are activities that can never be tailored out, e.g.:

a. agreeing on a development approach

b. specifying software and reviewing the specification

c. producing, validating and accepting software

d. managing the configuration

Examples of tailoring:

a. Prototypes and study software projects may come with minimal requirements for documentation and validation. These types of application are typically not re-used after having demonstrated the concept or their feasibility. Only a small proportion of prototypes are later used either for integration into other software products or as stand-alone applications. In the case where it is desirable to re-use a prototype, its reusability needs to be evaluated and the appropriate re-engineering will be performed.

b. The incremental development of a technology - following an improvement of its Technology readiness level (TRL) - could also lead to tailoring of the software engineering activities. The full list of activities that should be performed for a software "product" [TRL 6] will be performed incrementally along several contracts and development steps. This takes into account the foreseen criticality of the product in its intended operational environment. The tailoring addresses the extent of functions implemented in the software, the extent of its validation, verification and documentation, and the criticality level of each increment (see also 4.2.4 on software criticality).

NOTE Interpretation of the technology readiness levels for software is given in the chapter 7 of the ESA document “Guidelines for the use of TRLs in ESA programmes”.

c. For criticality category D software, the level of granularity of the detailed design, its depth, or in other words the concept of unit (raised to the level of function or even application), can be tuned to a level which takes into account both the needs in maintainability, or traceability, but also a realistic engineering approach related to the type of software (e.g. ground software). The internal interfaces [5.5.2.2] are defined at the same level of granularity, as well as the unit tests strategy.

d. For criticality category D software, the level of definition of the computational model can be adapted to the type of software and the criticality of its real-time aspects.

e. When ECSS-E-ST-40C requirements are tailored out, the associated Expected Output in the DRD is also tailored out. The DRDs propose a way to deliver the requested information, however 5.3.9.2 allows a different documentation structure, provided that all the needed information is there. This can be seen as a sort of tailoring of the DRDs.

f. Also, some documentation can remain available in Supplier premises without delivery, or can even be waived by the Customer.

4.2.6 Contractual and Organizational Special Arrangements

4.2.6.1 Contractual and Organisational Aspects of Software Development

Software development projects are usually realised in the frame of a contract between the supplier and the customer. The ECSS-E-ST-40C standard governs many aspects of the overall software development lifecycle and impacts the contractual agreement between the supplier and customer accordingly.

There are however some aspects such as the organisational structure of the customer and the supplier as well as the legal aspects of the contract, which may equally impact specific areas of software development lifecycle. Examples are software delivery and acceptance modalities or the agreed warranty periods between the customer and the supplier, copyright and Intellectual Property Rights (IPR) agreements, delivery schedule with associated definition of inputs and outputs.

It is neither in the scope of the ECSS-E-ST-40C nor its intention to address all possible scenarios of such aspects. This section gives an overview of some known issues, which are complementary to the specifications of the ECSS-E-ST-40C standard, and are typically specified in form of customised requirements in the Statement of the Work (SoW), or as part of the contract between the supplier and the Customer.

None of the following points in the sections below are mandatory and for inclusion in the contract between the customer and supplier. The objective is to raise awareness of the typical concerns which are not governed by the ECSS-E-ST-40C standard. For each of the mentioned aspects the customer/supplier analyses the need, and reflects the result in the contractual agreement if applicable.

4.2.6.2 Warranty, User support and Maintenance

Several activities happen towards the end of the development phase: warranty, user support, maintenance.

Warranty is a pure contractual issue included in the development contract, and is addressed in 4.2.6.6.2. User support belongs to the operation process section 5.9. The Software Maintenance phase is covered by section 5.10. The three activities may be performed within different contractual scope, and therefore may be performed by different actors.

Warranty is done by the software Supplier; User support is done by the SOS entity; the Maintenance can be done by the software Supplier or a third party.

An overview of the relationship between the different participants of the maintenance and support phases for a flight, or smaller ground system is shown in Figure 4-3. The Software supplier will be contracted by the System Integrator, or System Prime to develop, test and deliver the software in accordance with the Software Development Plan.

In some cases the System Supplier and Software Supplier will be the same company. In this case the arrangement may be similar but software warranty and support will be supplied by the software team rather than a separate company with a separate contract. For spacecraft there may not be system or software support, as authority may be transferred to a third party entity belonging to the operations organisation.

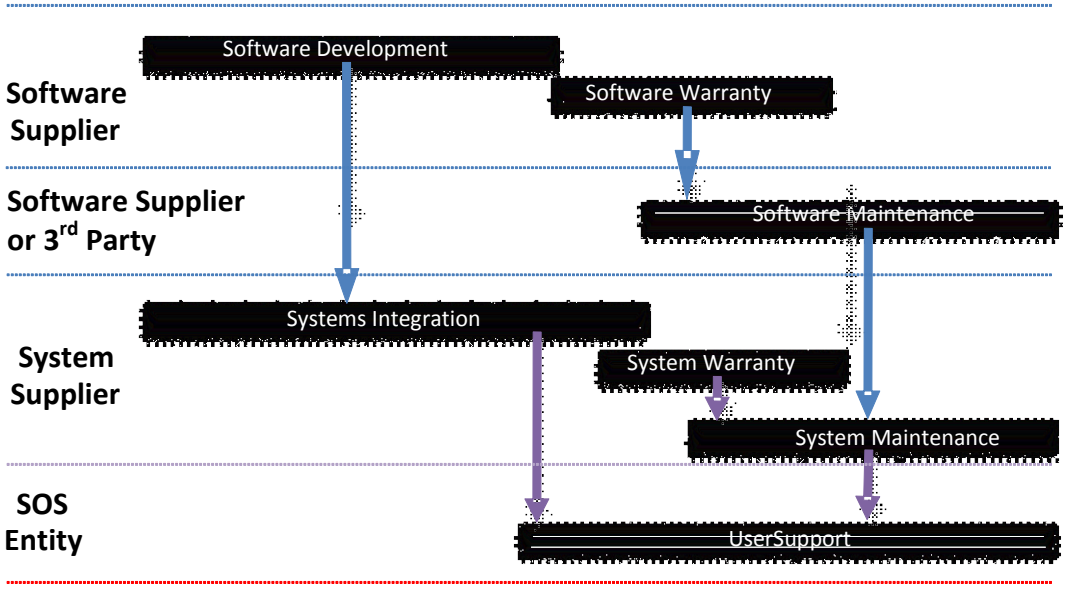


Figure 4-3 : Delivery of warranty and support between companies

Commonly now, larger ground system projects are being procured with the build and user support contracts linked very closely together (see Figure 4-4). This means that the contractor is asked to build a system, then maintain it and, in some cases, operate the system (including SOS entity role). In this case the service contract may imply that a team of people are employed during normal working hours, or on a 24/7 basis.

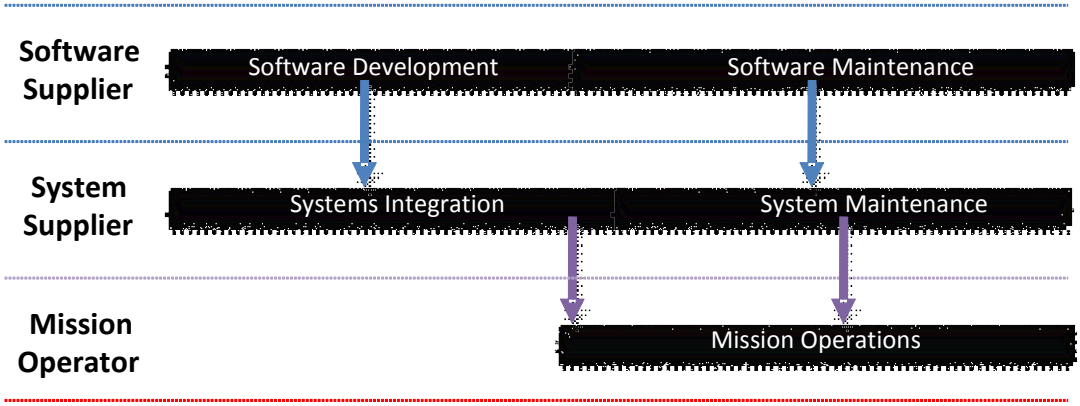


Figure 4-4 : Closely Coupled Build and Service Support Contract

For all types of contract it is usual to provision for the fix of defects, incidents or problems within an agreed period. Usually this period is different for the different criticality of defects. The contract may specify a Service Level Agreement (SLA) that defines the response time and couples them to contractual penalties or incentives.

4.2.6.3 Software Delivery Modalities

The **language**, **format** and **media** on which software and documentation is delivered is subject to agreement between the Customer and the Supplier, hence often specified through dedicated requirements in the Contractual agreement (e.g. often in the Statement of Work). Also the naming convention for the documents as well as the structure of the files and folders for software deliveries may be specified according to the Customers' requirements.

The Following requirements are provided as an example for such complementary specifications:

DELI-010	All the deliverables shall be written in British English.
DELI-020	All the deliverables documents shall be provided in Microsoft Word format in the version in use at the Agency and in PDF format. At the time of writing, the Agency uses Microsoft Office Suite 2003. . The latest version to be used shall be specified at KoM.
DELI-030	All the deliverables documents shall be produced based on the official document templates [RD1].
DELI-040	The deliverables for which the Customer owns the IPR shall not bear any information identifying the Contractor's Company or staff who produced the deliverable (e.g. Contractor's logo).
DELI-050	All deliverables shall be delivered in their original source format. Comment: For example in MS Word format for a document or in the source format of the UML tool for a UML document.
DELI-060	All deliverables documents shall in addition be delivered in PDF format.
DELI-070	All deliverables are subject to Customer review and approval.
DELI-080	All Documents shall be delivered in two original paper versions.
DELI-090	All documents delivered to the Customer for approval shall be signed electronically by the Contractor technical responsible.
DELI-100	All deliverable documents shall be named according to the rules defined in the Configuration Management Plan.
DELI-110	The deliverables shall be delivered to the Customer on duly labelled media as defined by the Configuration Management Plan.
DELI-120	The organisation of the files and directories on the media shall be approved by the Customer Technical representative.
DELI-130	Deliveries shall be done on DVD.
DELI-140	All documents shall be electronically uploaded on the document management portal, specified by the Customer.

4.2.6.4 Software Development Environment

The Selection of the supporting tools for performing the tasks in all stages of the software development lifecycle could be subject to agreement between the Customer and Supplier, hence often specified through dedicated contractual statements in the Statement of the Work or in the RB.

In particular the following tasks are often performed in specialised tooling, for example (non-exhaustive list):

a. Requirements Management

b. Software modelling and design

c. Software build and deployment management

d. Software Configuration Management

e. Software Testing, Verification and Validation

f. Software Defect Management

g. Software IPR and Asset Management

The application of ECSS-E-ST-40C mandates only the performance of the tasks as part of the Software Development Lifecycle activities. It does not mandate the usage of any tooling. However the utilisation of software development support tooling, facilitates appropriate execution of the corresponding tasks.

4.2.6.5 Software Development Methodologies

The ECSS-E-ST-40C does not mandate or exclude the adoption of a particular software development methodology, e.g. object oriented, service oriented, agile software development or adoption of model driven architecture paradigm.

The adoption of any software development methodology (and utilisation of related tooling) is therefore subject to contractual agreement between the Customer and the Supplier and is specified through dedicated requirements.

4.2.6.6 Software Acceptance Modalities and Warranty Period

4.2.6.6.1 Overview

The Acceptance testing period, also referred to as the check-out period starts with the delivery of the software by the Supplier to the Customer, and ends with the formal acceptance of the software by the Customer.

The duration of the acceptance period, i.e. the time granted to the Customer for performing its own validation of the delivered software is subject to contractual agreement and not specified by ECSS-E-ST-40C. It is therefore specified through dedicated requirements as part of the Contract (often in SoW).

Also the condition for acceptance is clearly specified as part of the contractual agreement between the Customer and Supplier.

The Customer may perform additional testing and validation at the top of the SVS specified by the Supplier during the acceptance period, and may report all identified software defects and non-compliances against the RB/TS to the Supplier. It is irrelevant how the Customer identifies software defects/non compliances, as long as they can be traced back to a clear non-compliance against the RB/TS. It is not subject to any contractual agreement between the Supplier and Contractor, which tests the Customer may execute during the acceptance period, since it's purely the Customers decision.

Since acceptances are often linked to payment milestones, the Customer and the Supplier may agree contractually on a stepwise acceptance approach, in which case the steps, their duration and the exact criteria for successful completion are contractually specified and agreed upon by both parties. This can for instance be realized through introduction of preliminary and final acceptance milestones.

In special cases, where the target operational environment of software is not easily accessible, e.g. in case of remote ground stations, the Customer and the Supplier may contractually arrange for additional acceptance milestones such as site and operational site acceptances.

Another case of additional contractually arranged acceptances is the so called Factory Acceptance, which is performed in the Supplier environment and aims at serving as a confident building step for the Customer before the software is delivered by the Supplier in the Customer's environment. (see section 5.7)

4.2.6.6.2 Warranty

The start and duration of the warranty is governed by contractual agreement between the Customer and the Supplier. It is also governed by the legal terms of the country/European/international law, applicable to the contract.

The following requirements are examples of contractual requirements for regulating the warranty start and duration:

The Contractor shall offer a rolling warranty starting with the first delivery and expiring 3 months after the Final Acceptance of the last delivery. Comment: "Rolling warranty" means that, at any point in time inside the warranty period defined in this requirement, the warranty covers the whole system.
During the warranty period, the Contractor shall be responsible to investigate and fix all errors found in the system and deliver corresponding corrections.
The Contractor shall perform an End of Warranty delivery, which shall reflect the latest status of the System both in terms of software and in terms of documentation.

The original Warranty period, which is provisioned for in the contractual agreement between the supplier and customer, often ends before the end of operational usage of the software (e.g. end of the space mission).

Software maintenance is in such cases established via the contractual agreements (typically in form of service level agreements) between the customer and the supplier (which can be the same supplier who developed the software or a different party who provides only the maintenance) in accordance to the operational needs of the customer on the subject software. The difference between warranty and maintenance is sometimes a source of confusion. The nature of the work to be performed under warranty and maintenance is similar (software defect management and removal). See also 5.10.2.1.2 for more details.

4.2.6.7 Intellectual Property Rights management

The assignment of Intellectual Property Rights for the software and other software related artefacts, e.g. documents, build and test scripts, etc. is not governed by the ECSS-E-ST-40 and is subject to contractual agreement between the Customer and the Supplier within the applicable legal frame.

4.2.6.8 Other development constraints / Customer Furnished Items

Specific constraints may be imposed by the Customer, which are not governed by ECSS-S-ST-40C. They need to be defined in the contractual agreement between the Customer and the Supplier. Examples are subcontracting scheme, target cost, particular risk sharing dispositions, the mandated extent of reuse, use of existing Customer Furnished Items, use of reference architecture, protection of secure material, artefacts relating to export or government control (e.g. EAR/ITAR).

5 Guidelines

5.1 Introduction

This chapter includes guidelines for the application of ECSS-E-ST-40C to space projects.

For the convenience of the reader, the note of section 1 is repeated here:

NOTE In order to improve the readability of the Handbook, the following logic has been selected for the sections 5, 6, and 7:

- section 5 follows the table of content of ECSS-E-ST-40C at least up to level 3 and generally up to level 4. For each sub clause of ECSS-E-ST-40C:
  + either information is given fully in section 5,
  + or there is a pointer into section 6 or section 7
  + or the paragraph has been left intentionally empty for consistency with the ECSS-E-ST-40C table of content, in this case, only " –" is mentioned.
- - section 6 expands selected parts of section 5 when:
  + either the volume of information was considered too large to stay in section 5,
  + or the topic is addressed in several places of section 5

  In any case, there is a pointer from section 5 to section 6, and section 6 mentions the various places in ECSS-E-ST-40C where the topic is addressed.
- section 7 follows the same principles as section 6, but gathers the topics related to margins and to real-time.

5.2 Software related system requirement process

5.2.1 Overview

5.2.1.1 Introduction

System-software level activities seem to be systematically underestimated or even misunderstood. Most software suppliers consider that these are not applicable to them, in particular in the case of avionics equipment or payloads. To illustrate the issue, some results are taken from a NASA study (*Final Report - NASA Study on Flight Software Complexity - Commissioned by the NASA Office of Chief Engineer, Technical Excellence Program, Adam West, Program Manager Editor: Daniel L. Dvorak, Systems and Software Division, Jet Propulsion Laboratory, California Institute of Technology*) . The study analysed the flow of engineering activities related to flight software and impacting its complexity. It came with

recommendations, in particular on system engineering. Some of them are quoted below from the Executive Summary of the Final report:

a. *"Engineers and scientists often do not realize the downstream complexity and cost-driving factors entailed by their local decisions. Overly stringent requirements and simplistic hardware interfaces can complicate software; flight software de-scoping decisions and ill-conceived autonomy can complicate operations; and a lack of consideration for testability can complicate verification efforts. It is therefore recommended to look at educational materials, such as a "complexity primer" and the addition of "complexity lessons" to NASA Lessons Learned.*[1]

b. Unsubstantiated requirements have caused unnecessary complexity in flight software, either because the requirement was unnecessary or overly stringent. Rationale statements have often been omitted or misused in spite of best practices that call for a rationale for every requirement. The NASA Systems Engineering Handbook[2] states that rationale is important, and it provides guidance on how to write a good rationale and check it. In situations where well-substantiated requirements entail significant software effort, software managers should proactively inform the project of the impact. In some cases this might stimulate new discussion to relax hard-to-achieve requirements.

c. Engineering trade studies involving multiple stakeholders (flight, ground, hardware, software, testing, and operations) can reduce overall complexity, but we found that trade studies were often not done or only done superficially. Whether due to schedule pressure or unclear ownership, the result is lost opportunities to reduce complexity. Project managers need to understand the value of multi-disciplinary trade studies in reducing downstream complexity, and project engineers should raise complexity concerns as they become apparent."

5.2.1.2 System engineering

System engineering is an interdisciplinary approach governing the total technical effort to transform requirements into a system solution. The relevant framework is contained in the "**Space engineering - System Engineering General Requirements**" (**ECSS-E-ST-10C**) **Standard** describing the ECSS relations for the software discipline.

The system engineering activities of a project are conducted by an entity within the project team of a supplier. This entity is called "system engineering organization". The system engineering process consists of activities to be performed by the system engineering organization within each project phase. **It is recursively applied by each system engineering organization of each supplier of the elements of the product decomposition.**

The system engineering process is intrinsically iterative across the whole life of a project, in particular in the initial phases (i.e. 0, A, and B) of the development of a complex system (e.g. a spacecraft), procured through a multi-layered set of suppliers. During these phases, the system engineering organization derives design oriented technical solutions using as an input the design-independent customer requirements.

Through this process the system engineering organization performs a multidisciplinary functional decomposition to obtain lower level products (both hardware and software). At the same time the system engineering organization decides on balanced allocations, throughout the system, of resources allocated by the customer and respects agreed margin philosophies as a function of the relevant technology readiness levels. The functional decomposition defines, for each level of the system, the technical requirements for the procurement of subassemblies or lower level products as well as the requirements for the verification of the final characteristics of each product.

[1] http://llis.nasa.gov/llis/search/home.jsp
[2] NASA/SP-2007-6105 Rev1

The system engineering process uses the results of these lower level verification activities to build bottom-up multi-layered evidence that the customer requirements have been met.

The system engineering process is applied with various degrees of depth depending on the level of maturity of the product (e.g. new development or off-the-shelf).

The system engineering process can be applied with a different level of tailoring as agreed between customer and supplier in their business agreement.

The system engineering organization has interfaces with organizations in charge of management, product assurance, engineering disciplines, production, and operations and logistics.

5.2.1.3 System level framework relevant for Software

The **context** of the space software engineering activities is the overall Space System Engineering process. The software related system requirement process, produces the information for input to the system requirements review (SRR). This establishes the functional and the performance requirements baseline (RB) of the software development. It constitutes the link between the space system processes, as defined in ECSS-E-ST-10C for flight systems, and in case of ground systems in ECSS-E-ST-70C, and the software processes.

According to the recursive customer-supplier model of ECSS, at each level, the customer is responsible for the delivery of a system in which the developed software is integrated. According to the recursive system-subsystem model of ECSS-E-ST-10C, he is responsible for the specification of the system requirements at lower level and, in particular, for any software comprised in the system.

From these perspectives major contributions from the "Software" discipline are absolutely necessary for the system level in order to provide a space system / product that will finally satisfy the customer's needs. A Software System Engineering organization or (depending on the hierarchical level of the customer-supplier relationship) at least function is therefore fundamental for the project setup on all supplier levels - independent from the value of contribution (e.g. Subsystem, Equipment's, Payloads) to the final space system or product. This organization or function serves as focal point / interface for all software aspects.

5.2.1.4 Software System Engineering

Software System Engineering comprises the work of software specialists (like analysts, designers, programmers, quality engineers and others).

When the software development is included in a complete space project, the software engineering processes have also a relationship with the project phases (0, A, B, C, D, E, F) and in particular with the system level reviews (e.g. some system-software activities are executed in phase B).

5.2.1.5 Co-engineering by means of an Integrated System / Software team

The idea is to transfer knowledge and responsibility between the system team and the software teams and vice versa. This is an important aspect to be considered during the development processes since misunderstandings and disparate responsibilities will prompt for failures. Furthermore this will shorten the loop back from RB to TS even down to software design and code if felt necessary in order to assess the feasibility and completeness of the software requirements. Therefore trade-offs may be conducted more efficiently than the standard Waterfall model.

In case the software supplier belongs to the same organization as the system provider, some activities may be performed jointly or by the same person when different tasks are assigned to it. In other words: the same person could perform tasks wearing different organizational hats (for customer + supplier / for system + for software levels).

NOTE This might also work for a software supplier from a different organisation, if a dedicated business agreement can be established early enough, e.g. in phase B.

5.2.1.6 Preconditions / Tailoring

The **System Requirement process** is a system process and initially a customer process. However the software development is an integral part of an overall space system product. The purpose of this process is to generate explicit requirements for the system from the view of a system engineer and / or a customer. The output is the Requirement Baseline (RB).

The System Requirement process is very often subject to tailoring, especially for Research & Development and Prototyping projects. For general tailoring refer to Appendix S of the ECSS-E-ST-40C or chapter 4.2.5 of this Handbook. See also the tailoring of the process in case of software or infrastructure reuse in 5.4.3.7.2 of this document.

a. The following points aim at clarifying the border between RB and TS:

b. ***A SSS requirement*** should define the software need in the system (the **WHAT**). This allows being independent (as far as possible) of the hardware and software implementation, i.e. several solutions may answer to the same need. Hence, for example for flight software, the RB includes IRDs or User Manuals for Space-to-Ground interfaces, Operations, Command / Control, Equipment's, HW/SW, OBC, HSIA, Electrical IRD describing the constraints of the system when there are several pieces of equipment, and the SSS explains what is the SW behaviour with respect to these interfaces.

c. ***A SRS requirement*** should answer to the needs through a functional implementation taking into account software and hardware constraints in the system (closer to the **HOW**, at least from a logical or functional point of view). This allows for development of the software detailed design. Hence, for example for flight software, the Technical Specification is including the SW TM/TC ICD.

The quality of the requirements in general and of the Requirement Baseline in particular has proven to directly impact the success of the software project. The tailoring of this process, which in a way is a self-assessment of the customer, needs to be performed carefully. In case of doubt, the elaboration of the Requirement Baseline can be delegated to the Supplier, but the responsibility and the approval of it needs to stay at the customer level. Any (initially) missing system requirements can be included through the System Requirement Review being the synchronization point. The SRR also allows the supplier to verify the Requirement Baseline, to adopt it and to suggest improvements.

If there are critical requirements identified at a later stage in the project, their implementation may induce high extra costs and major schedule delays.

If applicable, the software related system requirement process consists of the following 3 activities:

d. software related system requirements analysis

e. software related system verification

f. software related system integration and control

which are described in sufficient level of detail in the ECSS-E-ST-40C.

5.2.2 Software related system requirements analysis

5.2.2.1 Specification of system requirements allocated to software

The Use Case technique can support this activity and is described in 6.1

5.2.2.2 Identification of observability requirements

The operability of the Space Vehicle is a major objective of Monitoring & Control. It must meet not only the nominal operating needs, but also the anomaly processing and flight feedback information requirements. The induced requirements are directly applicable to the software (both ground and flight) and constraints its design.

Although belonging to the E70 "ground and operations" branch of the ECSS standards, the ECSS-E-ST-70-11C - "space segment operability" document defines in detail the requirements applicable to the spacecraft, covering both commandability and observability.

This document provides all the necessary inputs to the software for what relates to this discipline, in conjunction with ECSS-E-ST-70-41A "packet usage standard" and ECSS-E-ST-70-01C "OBCP".

The Requirement Baseline should therefore make applicable explicitly these standards, and possibly refine all the included requirements down to the software level during the system / software engineering phase.

5.2.2.3 Specification of Human Machine Interface requirements

In principle the ECSS-E-ST-10-11C standard is applicable to ground segment as its introduction states "This Standard defines requirements for the integration of the human in the loop for space system products". Many processes discussed in ECSS-E-ST-10-11C are general ones for any human system integration project.

However, it is clear that the ECSS-E-ST-10-11C standard puts emphasis on the Human Space Flight domain. The document could be tailored for Mission Control System needs, with the relevant stakeholder cooperation. Alternatively, upon approval by all stakeholders, another specific standard that meets or exceeds the ECSS-E-ST-10-11C standard, for this domain, could be made applicable.

5.2.3 Software related system verification

5.2.3.1 Verification and Validation process requirements

-

5.2.3.2 System input for software validation

The standard asserts that the customer specifies requirements for the validation of the software against the requirements baseline and technical specification, in particular mission representative data and scenarios, and operational procedures to be used (see 5.2.3.2 of ECSS-E-ST-40C). It also asserts that the supplier develops and documents, for each requirement of the software item in RB, a set of tests that validate the software against the mission data and scenario specified by the customer (see clause 5.6.4.1 of ECSS-E-ST-40C).

Scenarios are therefore essential to produce the software validation plan. It strengthens the suggestion that Use case technique (see 6.1) should be used for the definition of software requirements, at RB level and TS level. Thus the RB defines use cases that are refined in the TS. Once the RB and TS are written, the software validation plan is almost specified, because, if use cases are used all along the RB and TS elaboration, then they can be easily transformed into "test cases". This facilitates the preparation of the Software Validation Specification (SVS). However, the validation is based on requirements, and the validation plan must cover all the requirements. Achieving coverage requires complementing or enriching the scenario.

5.2.3.3 System input for software installation and acceptance

-

5.2.4 Software related system integration and control

5.2.4.1 Identification of software versions for software integration into the system

See 5.3.5.2

5.2.4.2 Supplier support to system integration

-

5.2.4.3 Interface requirement specification

-

5.2.4.4 System database

5.2.4.4.1 Introduction

This part aims at providing some details about the System Data Base (SDB), its interface with the other space software (particularly the central on-board software) and some of its development constraints.

All engineering processes involved in the space system life cycle are based on data management processes and tools. The SDB is the reference repository of all the system level data, e.g.:

a. System configuration data,

b. System element physical properties,

c. TM/TC data definitions,

d. On-board software parameters / monitoring / Safe Guard Memory / OBCP definitions,

e. On-board communication protocol data,

f. AIT specific data,

g. Ground calibration/decalibration functions,

h. Operation / FDIR specific data,

i. Electrical I/F data (at functional interface level),

j. Simulator data (modelling input).

The System team is responsible to define, configure, distribute and validate these system level data (see ECSS-E-ST-40C clause 5.2.4.4), with the support of the SDB.

The SDB is thus in the heart of the industrial process (i.e. development, validation, operations) of a complete system (including ground and flight segments). It interfaces with almost all engineering space subsystems (named SDB users below), e. g. the ground segment (including the control centre), the validation facilities, the simulators, the on-board software's, the operation tools…

The objective is to configure all software(s) and subsystems in a consistent way (see ECSS-E-ST-40C clause 5.4.3.6c).

One key point in order to ensure a proper exchange of information is to define, when possible, a well-defined name for all the described entities. It is of high added value for the whole project to ensure a consistent naming of e.g. telecommands, telemetry parameters, system parameters, from the Requirement Baseline to the software implementation and the SDB.

The SDB may be distributed or centralized. In any case, the important point is to ensure the consistency of the structure and of the provided data.

Data structures are defined at the SDB user (e.g. on-board software, Ground segments) PDR. Final content and values are defined at CDR.

The SDB tool itself is software designed through a conceptual Data Model that defines all data structures and external interfaces needed by SDB users, for software generation (including documentation) and/or operations. The SDB tool needs to be available in accordance to subsystems schedule; this creates strong dependencies among life-cycles of these subsystems and SDB tool. Hence the clause 5.2.8.4 provides some recommendations to properly tailor the ECSS-E-ST-40C.

5.2.4.4.2 On-board SW SDB user

The development of the on-board software of a space segment is based in particular on data:

a. provided by the system/operations team (e.g. numerical data, housekeeping packet definition, monitoring thresholds),

b. or produced by the on-board software itself (e.g. detailed description of telecommands, telemetry reports, and software and telemetry parameters).

The configuration data values produced at system level are necessary to build the on-board software binary image. Preliminary values are necessary to perform on-board software validation (especially closed loop tests) and the final values can be delivered later for final acceptance tests (see also ECSS-E-ST-40C clause 5.6.4.2b).

Due to the huge volume of system level configuration data, it is highly recommended that the introduction of the system level data into the on-board software binary image build process is automated. This will provide the additional benefit of being able to introduce late modifications of configuration data with a minimal cost.

Most of the data produced by the on-board software are specified at the time of the Technical Specification production. Defined in the Software Interface Control Document or SRS, they are then introduced in SDB. Their values are then defined by the software or the System teams.

Here also, it is highly recommended to put in place a computer assisted and even automated process, based on a formalized description in an electronic format (e.g. XML Schema, Eclipse Modelling Framework) of the corresponding section of the Technical documentation.

The data base parameters, that are used to configure the software, are defined and verified by the system team. They become requirements for the software, and as such they are validated (to some extent for the system level data) by the software team. They are fully validated in context of the system tests.

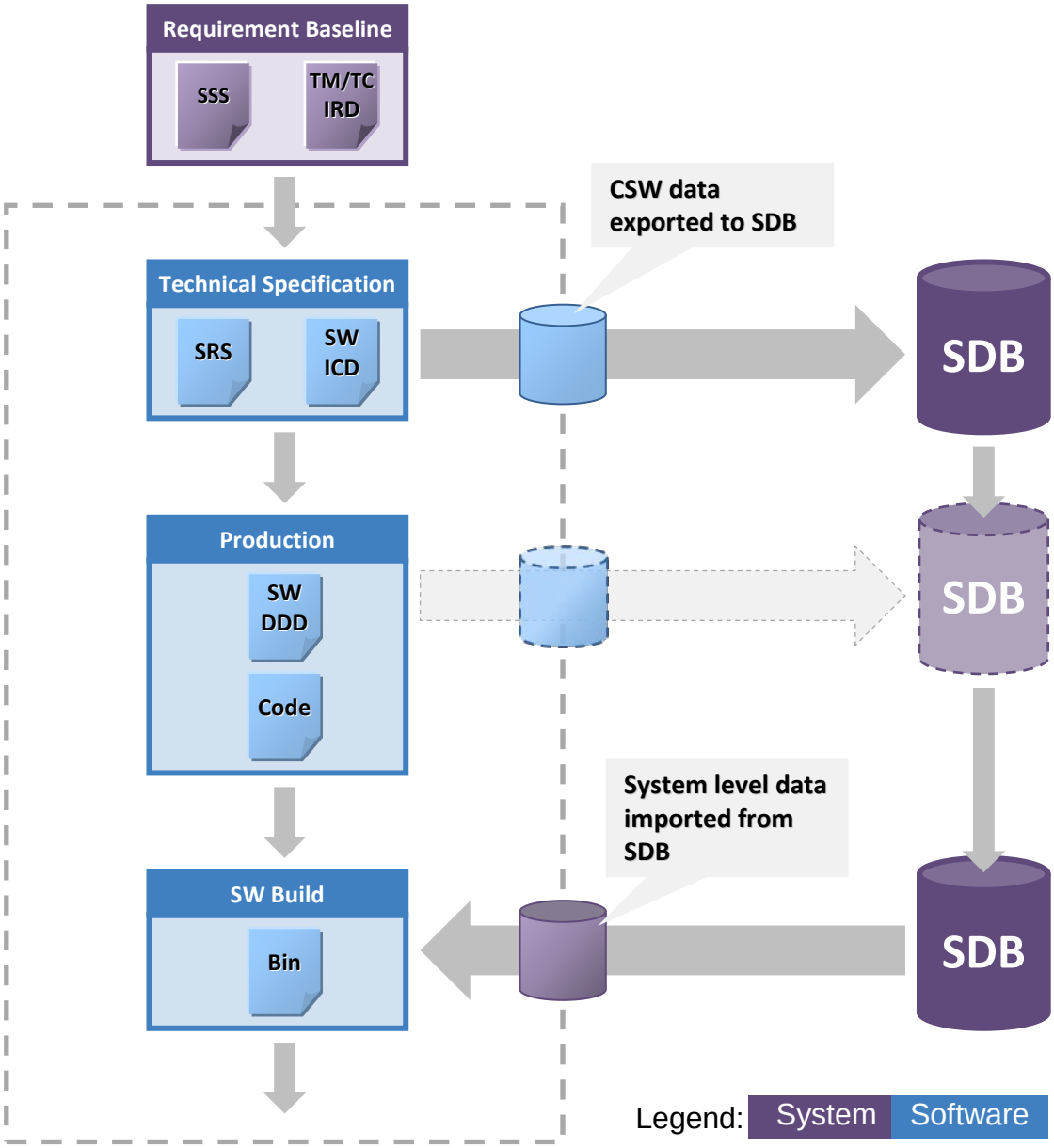


Figure 5-1: System database

NOTE CSW denote on-board software ("central software")

5.2.4.4.3 Ground segment SDB user

For the traditional ground segment software, only a part of the SDB is relevant which is limited to the information related to the TM/TC. For example, the "pure On Board information", like harness, are not used. Other data are specific for the ground segment, e.g. alarms, calibration/de-calibration functions, ground modes…

Since the ground segment is composed of many components, such as Mission Control System, Ground Station System, Data Distribution System, Network and Communication System, which are geographically distributed and developed, owned and operated by different entities, each subsystem may use a database of configurable items at subsystem level, and regularly synchronized from the SDB.

5.2.4.4.4 SDB Tool specific life-cycle

As a software product, ECSS-E-ST-40C applies to the SDB tool development. The SDB tool life cycle is an adaptation of the complete software life-cycle, taking into account:

a. the organizations, processes and tools set-up by industries for data management,

b. the dependencies with life-cycles of other space SDB users (e.g. early dependencies with central on-board software, late dependencies with Ground Segments),

c. the already existing SDB applications.

Apart from the infrastructure aspects, the main development of the SDB tool relies on the SDB conceptual Data Model, basis of the population, consultation, import and export tools. This Conceptual Data Model is therefore reviewed during the SDB PDR, especially to verify its compliance with SDB users' interfaces. Formally, these interfaces (e.g. SDB to central software ICD) are available at the PDRs of the SDB users. The SDB PDR should therefore be held after the SDB users PDR.

NOTE The process to generate the Conceptual Data Model is outside the scope of ECSS-E-ST-40C. It is addressed by specific other document ECSS-E-TM-10-23A (Space system data repository).

Reciprocally, the SDB user PDR authorizes starting the design and implementation of these components, where the SDB should be immediately available to start the data population! Moreover, the need dates for the various SDB users (e.g. on-board software, simulators, test benches, ground systems, operations) are different.

The SDB is thus available too late for several space SDB users.

Hence, the SDB development cannot wait for the complete SDB users' data specification readiness and the SDB PDR needs to be held early in the space system schedule, and before the end of all SDB Users PDRs.

Thus, the late arrival of some inputs needs to be anticipated to avoid late impacts on the SDB architecture.

The scheme in Figure 5-2 shows, in case of a new SDB tool development, the constraints among the SDB users life cycles and the consequent critical path.

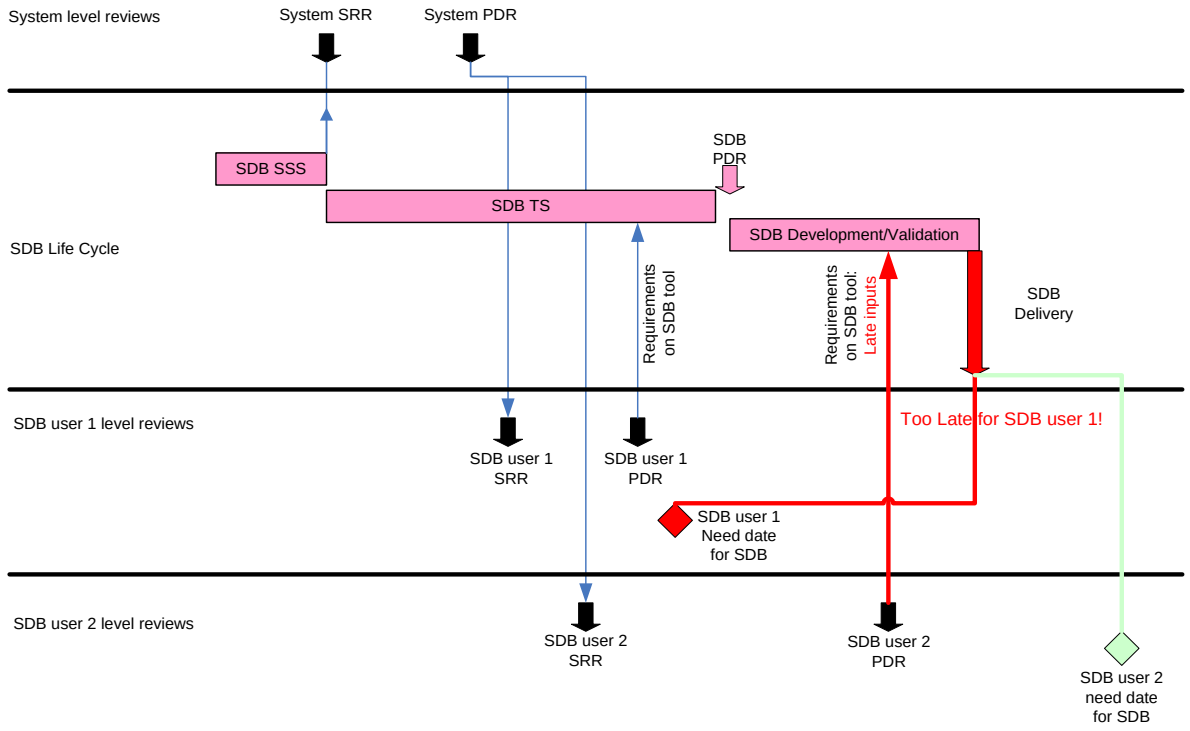


Figure 5-2: Constraints between life cycles

To answer all these constraints, the approach is to take benefits of already existing SDB applications, so as to be able to provide intermediate versions of the SDB as early as possible, and to put emphasis on the validation phase.

The proposed life-cycle and reviews for SDB is as follows:

a. System SRR: high level requirements on SDB are specified and reviewed. It is recommended to take benefit of already existing SDB applications: when applicable, a first version of the Reuse File is issued, identifying which application could be reused and the expected changes on the existing software(s).

b. The specification (SSS) then details the data management processes, SDB users and their roles, the scope of data to be managed, external interfaces for import and export, expected data volumes and performances, schedule of SDB tool itself (with incremental approach when needed) and preliminary schedule of SDB data deliveries.

c. This specification is reviewed during System PDR.

When the approach of reuse of an existing SDB tool, justified by a Reuse File, is agreed, a first version of SDB tool can be immediately delivered for initial population.

a. Each Change needed to comply the mission requirements is further implemented in new SDB tool versions,

b. Each SDB tool delivery is subject to an acceptance review, based on an acceptance plan prepared by the System team.

In case a complete or important development is initiated, the following reviews take place:

a. SDB PDR: the TS specifies the complete SDB tool, including the SDB conceptual data model, and mapping with external interfaces. The reuse of existing components (e.g. data models, data management tools) are identified,

b. SDB CDR: the SDB is validated against its TS and can be early delivered to users,

c. SDB TRR: recommended to get the agreement from the SDB users on the operational representativeness of the datasets that is used for the SDB QR

d. SDB QR: SDB qualification against the RB. The qualified SDB is delivered,

e. SDB AR: the SDB is accepted by the SDB users, represented by the SDB tool responsible in the System team.

5.2.4.5 Development constraints

-

5.2.4.6 On board control procedures

As mentioned in the ECSS-E-ST-70-01C "OBCP" standard, two categories of OBCP are considered:

a. "On-board Application Procedures (OBAP)" which implement part of the basic functionality of the spacecraft, i.e. which are an integral part of the spacecraft design. The overall qualification of the spacecraft requires the integration of the complete set of OBAPs.

b. "On-Board Operations Procedures (OBOP)" that are used to "operate" the spacecraft, but which are not involved in the qualification of the spacecraft.

NOTE In this context, "involved" has to be read as "included".

As a consequence, OBOPs are considered as designed and validated according to the engineering requirements defined in ECSS-E-ST-70C "ground and operations engineering" (therefore irrelevant for this handbook), whereas OBAPs are considered as designed and validated according to engineering requirements defined in ECSS-E-ST-40C, and ECSS-Q-ST-80C standards and qualified with the spacecraft.

The engineering process required for OBAPs derives from the level of isolation provided by the OBCP system, enabling the corresponding level of decoupling from OBSW. This level of isolation is determined according to the extent to which the capabilities of the system are protected against propagation of functional failures induced by the OBAP. This enables to decouple consequently the validation of the OBSW from the validation of the subsequent OBAPs.

According to these criteria, combined with the level of complexity of the OBCP itself, the OBAPs may be classified into three categories:

basic: are very sequential set of activities (at TC/TM interface) with limited use of conditional expressions. Access to the TC/TM functions is done by using the interface provided by the OBSW, without direct access to the OBSW itself. The category is of the same nature as the OBOPs. Therefore, the decoupling between those OBCPs and the OBSW is the same as the TC/TM. They could be developed, integrated, validated and maintained by the system team and transferred to the operations. Furthermore, they are qualified in the scope of the Spacecraft qualification.

advanced: are interacting directly with internal OBSW functions, with expected real time properties and/or complex conditional expressions, with also potential interactions the one with the others (including concurrency). However the OBCP engine should offer a strong level of granted segregation (in particular time and space partitioning capabilities), meaning that interactions between OBAPs and the OBSW are performed through a set of fully validated software functions in scope of the overall OBSW qualification. Under those conditions, those OBAPs could be developed, integrated and validated under system team responsibility.

NOTE Sending an "erroneous" advanced OBAP can have two effects (i) to cause spacecraft system misbehaviour or (ii) to cause a software crash. The OBCP engine cannot protect against

advanced OBAP that endangers the spacecraft system, but can protect the software in the scope of its qualification.

extended: are considered as of the same level of complexity as OBSW (in terms of algorithm as well as real time properties), interacting with the rest of the OBSW at a lower level. For this category of OBCPs, the level of containment and segregation of OBCP w.r.t. the OBSW cannot be granted. The complete OBSW and the extended OBAPs are qualified together. Therefore they are developed, integrated, validated and maintained within the scope of OBSW and under OBSW team responsibility.

The ECSS-E-ST-40C tailoring for basic OBAP, and the ECSS-E-ST-70C tailoring for OBOP, result in principle in the same level of engineering effort.

There is no expected additional tailoring of ECSS-E-ST-40C for the extended OBAP other than the one of the OBSW.

The following possible tailoring guidelines for basic and advanced OBAPs levels are identified, with proper justification:

documentation: could be reduced and optimised in order to gather all the relevant information within a reasonable number of documents. For example, there may be no need for RB and architectural design document in the case each individual OBAP is a set of self-standing basic and sequential operations. Behavioural and real time aspects are in principle encapsulated / hidden to the OBAP developers (this would be contradicted if it is expected to have many OBCPs and that they interact the one with the others). Nevertheless, a set of Requirements, even at basic level should be formalised. This allows to reflecting which OBSW functions, resources and interfaces will be used by a given OBAP. The associated documentation is used to consolidate the interlink between the family of OBAP to be operated on a given satellite, also in case of OBSW resources conflict analysis.

reviews and formalism: could be decreased down to the "technical review" level and their number may be restricted to the adequate level (e.g. SRR, PDR and CDR kind of reviews) enabling to review the required documentation. The objectives and organisation should be adapted accordingly. Due to interactions with the software reviews and system reviews, they should be also synchronised according to the scheme depicted in ECSS-E-ST-70-01A chapter 6.

validation, qualification and acceptance: could be performed as part of system activities without dedicated phases. In terms of facility, basic OBAPs would need to be validated at least with the spacecraft simulator, whereas the advanced level would require being integrated and validated together with the real OBSW, on the avionics test bench and exercised on the real spacecraft.

unit and integration testing: may be skipped or combined together with validation phase without full structural code coverage evidence.

5.2.4.7 Development of software to be reused

-

5.2.4.8 Software safety and dependability requirements

The methods and techniques to be used whenever software safety and dependability engineering activities are necessary are discussed in the ECSS-Q-HB-80-03A handbook.

5.2.4.9 Format and data medium

-

5.2.5 System requirement review

5.2.5.1 Relationship between software SRR and system SRR

Since system process and software process share the same review names, there is commonly confusion between them. In order to remove ambiguities in the frame of the software project, the review name could be systematically prefixed with "System" for system level reviews and with "Software" for software level reviews.

5.2.5.2 Software requirement reviews

According to the recursive customer-supplier model of ECSS, at each level, the customer is responsible for the delivery of a system in which the developed software is integrated. According to the recursive system-subsystem model of ECSS-E-ST-10C, he is responsible for the specification of the system requirements at lower level and, in particular, for any software comprised in the system.

The system level technical requirement specification for the software subsystem (system TS for software) in ECSS-E-ST-10C standard is the requirement baseline in the ECSS-E-ST-40C standard. The ECSS-E-ST-40C activity "evaluation of system baseline" verifies that the specific software activities at system level described in clause 5.2 are actually taken into account. This is formalized at the software SRR, the software being viewed now as a (lower-level) system. When it refers to SRR, the ECSS-E-ST-40C always refers to the Software SRR.

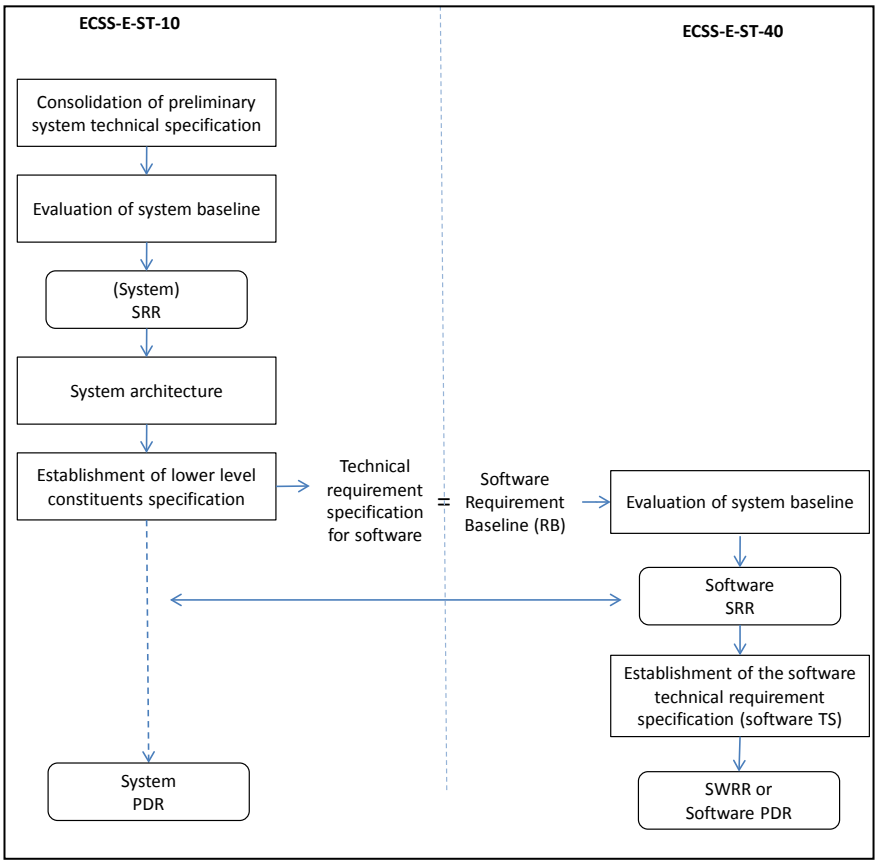


Figure 5-3: Software requirement reviews

The (System) SRR is a system review, held during the phase B of the project. The review main objectives are the assessment of the preliminary system design definition, the assessment of the preliminary verification program and the release of technical requirement specification for the

subsystems. The system requirements allocated to the software subsystem are thus identified at this point.

The Software SRR is a review of the requirements baseline (RB) that is the expression of the user needs for the software system – and which is customer documentation. It is thus a review focused on the customer activities related to the software system, as per ECSS-E-ST-40C clause 5.2. The objectives of this review are to ensure consistency with the system requirements (including interfaces) and system architecture, to check the feasibility of the software development and to reach the approval of the software requirements baseline by all stakeholders. When possible, the software supplier should be involved in the process in the frame of co-engineering activities, particularly since the requirements baseline is the main technical document initiating the software subsystem project.

The SWRR is a review of the technical specification – which is the supplier's reply to the requirement baseline. The objective of this review is to ensure that the requirement baseline is correctly understood and covered. The SWRR is a non-mandatory review anticipating the Software PDR for the part of the PDR addressing the software requirements specification and the interfaces specification.

5.3 Software management process

5.3.1 Overview

Although this is only implicit in ECSS-E-ST-40C, this clause is mainly about producing the software development plan, including life cycle, development methods, reviews, budgets, etc.

5.3.2 Software life cycle management

5.3.2.1 Software life cycle identification

This set of requirements in ECSS-E-ST-40C asks in particular for a development plan including a life cycle definition and some methods and tools. Among these subjects, this section focuses on three topics:

a. the life cycle selection, as required by ECSS-E-ST-40C requirement 5.3.2.1.a and b, is discussed in 6.2

b. the model driven approach, which is a possible technique replying to ECSS-E-ST-40C requirement 5.3.2.1.c, is discussed in 6.3

c. the testing methods and techniques, replying to the same ECSS-E-ST-40C requirement, is discussed in 6.4

5.3.2.2 Identification of interfaces between development and maintenance

-

5.3.2.3 Software procurement process implementation

-

5.3.2.4 Automatic code generation

Automatic code generation is discussed in 6.5

Model based engineering is discussed in 6.3

5.3.2.5 Change to baselines

-

5.3.3 Software project and technical reviews

5.3.3.1 Joint reviews

5.3.3.1.1 Action management

Actions created during reviews are usually associated to an (absolute) due date. Lessons learnt from projects indicate that these actions should also be associated to future milestones. For example, an action with a date will be also tagged as "blocking CDR" or "blocking flight". This allows (i) to put priorities on actions when they are generated and (ii) to check the closure of essential actions in the scope of a subsequent review.

This is in particular useful in case of iterative life cycle, where actions generated in a version affects other versions, and therefore should be analysed from several perspectives in order to associate a corresponding tag:

- actions affecting current version for the purpose of utilisation e.g. within AIV

- actions affecting subsequent versions but not the current one (e.g. a functionality was not used in the current version for the purpose of AIV but on which subsequent versions rely on)

- actions affecting e.g. overall quality of the code (i.e. cyclomatic complexity) which will matter only for the overall SW next major milestone.

These perspectives will help to identify tags "blocking XXX milestone for V version" which are recommended to use.

5.3.3.2 Software project reviews

-

5.3.3.3 Software technical reviews

-

5.3.4 Software project reviews description

ECSS-E-ST-40C project reviews are the same milestones as the one defined at project management level, in ECSS-M-ST-10C. These milestones intend to control the progress of the activities, stabilise baselines for the next stages and acknowledge completion of tasks until finalisation of the complete product. This means that, for instance, the flight software is subject to the same phasing and review logic as the spacecraft and its constituent subsystems. The purpose of a spacecraft or subsystem review is generally to accept the item in a given state and to authorize the continuation of the development to the next phase. For example, the PDR accepts the start of Phase C, the CDR releases the final design (and the software manufacturing), the QR analyses the status of the system verification and the AR judges the readiness of the product for delivery.

The ECSS-E-ST-40C series of reviews are intended for the customer to check the software development, and for the supplier to schedule the project progress, but are also used to properly phase the software development with the space system development. Generally, no phase is complete until the documentation for that phase has been completed and approved by the Customer through a dedicated review.

As stated in the ECSS-E-ST-40C sub clause 5.3.6, software reviews are synchronized with system ones that are mainly sequential because the system life cycle follows the V-Cycle model (see ECSS-M-ST-10C figure 4.4). Therefore, and whatever the selected software lifecycle model (see §5.3.1), there should be at the minimum the set of ECSS-E-ST-40C project reviews (i.e. SRR, PDR, CDR) synchronized with system level. Then, they can be completed by technical reviews where appropriate to map to the software lifecycle.

5.3.5 Software technical reviews description

5.3.5.1 Description

Technical reviews have been introduced in the ECSS-E-ST-40C standard, in order to account for the need of projects to handle reviews which may have a lower level of formalism, than the one foreseen by ECSS-M-ST-10-01C they are sometimes named "key points". Their formalism is agreed between customer and supplier, as the technical reviews are joint reviews, the Customer is involved and has visibility.

A Technical Review includes the same tasks as a Project Review (ECSS-M-ST-10-01C 4.2):

a. initiation of the review (members and procedure)

b. distribution of the data package (availability time of the data pack can be reduced w.r.t the time requested for a project review)

c. review of the documentation

d. review findings and conclusion

Flexibility with respect to ECSS-M-ST-10-01C (chapter 5) can be introduced for:

a. the independence

b. the objectives

c. the review bodies,

d. the review meetings

e. RID formalism

f. documentation DRDs (review procedure, RID, review team report, review authority report)

For example, a technical review may not have a chairman. The minutes of the collocation meeting with the attached disposed comments may be used as a report.

SWRR/DDR (Software Requirement Review and Detailed Design Review)

The SWRR and the DDR are NOT Technical Reviews. They are anticipation of the PDR and the CDR in case the Technical Specification or the Detailed Design is baselined before the project review. Therefore the formalism of the SWRR and the DDR is the one defined in ECSS-M-ST-10-01C.

SQSR (Software Qualification Status Review)

In case of reuse of existing software, the software reuse file can be reviewed in advance of the PDR, during a dedicated technical review. The reuse components/software and policy could have some impact of the software product design. This technical review enables to share with the customer the reuse approach sufficiently early in the development, and especially before the design activities.

The purpose of the review is to present the analysis that has been carried out on some already existing software components/software, in order to re-use them for the development of the current project/product. The analysis is based on assessment:

a. of all the information used to decide about the reuse (or not) of existing component/software,

b. of the qualify level of existing component/software,

c. that the reused software meets the project requirements and the domain use,

d. of the specific proposed actions, if they are required.

This review may be merged with SRR or SWRR.

TRR/TRB (Test Readiness Review and Test Review Board)

Two Technical Reviews are specifically mentioned in the software standard, the Test Readiness Review [TRR] and the Test Review Board [TRB]. They go together, one before a test execution (to ensure that the test specification and the test means are ready) and the other after the test activities.

Requirements 5.3.5.1a and 5.3.5.2.a are clear on the fact that these reviews need to be organized in relation to test activities, i.e. unit tests, integration tests and the various validations and acceptance testing. The tests activities subject to TRR/TRB are mentioned in the Software Development Plan.

The phrasing "as defined in the Software Development Plan" (actually duplicated in ECSS-Q-ST-80C 6.3.5.4) suggests that some testing activities might be considered as not subject to TRR/TRB. The common practice for unit testing or integration testing is indeed not to have specific TRR/TRB.

However, the TRR for Validation can be anticipated before the integration tests. It becomes a TRR for integration and validation where both Integration Test Plan and SVS are reviewed. Integration Test Report and Validation report are checked at the TRB for Validation (or at the CDR).

ECSS-Q-ST-80C requires (see requirement 6.1.5) that TRR/TRB are mandatory for any validation campaign, therefore before CDR, QR and AR. Common practice is to reflect it in a consistent way with ECSS-E-E40C (where requirement 5.3.3.3.a-b allows to define TRR/TRB in the SDP), according to the project needs. Typical examples are the following:

a. the TRB is very often handled together with/in the scope of formal reviews: TRB for validation together with the CDR, TRB for qualification together with the QR and TRB for acceptance together with the AR.

b. the TRR for qualification is sometimes handled together with/in the scope of the CDR

c. the TRR for acceptance is sometimes handled together with/in the scope of QR.

d. for larger software development, several TRR/TRB may be handled (e.g. per function) for validation w.r.t. TS.

e. for smaller software developments, TRR/TRB may be suppressed.

Typical objectives of the TRR are introduced in annex P.2.1<6>a. Note 5.

It should be highlighted that, as the TRR is supposed to confirm the readiness of test activities, this implies that the relevant SVS (against TS or RB) is available at the TRR, as well as sufficient documentation to achieve the objectives of the review stated in the Review Plan, in particular documentation of the test environment.

TRR Deliverables are summarized in annex Q.4

Typical objectives of the TRB are introduced in ECSS-E-ST-40C annex P.2.1<6>a. Note 6.

TRB Deliverables are summarized in annex Q.5.

DRB (Delivery Review Board)

The DRB is an additional Technical Review not mentioned in the software standard, but actually necessary to authorize the delivery of a software version to an external team aside to the project

reviews (usually software versions are released at CDR, QR and possibly AR). This review aims at clarifying the state of the version (according to the previous agreement between the provider and the user of the version) intended to be delivered in terms of perimeter, validation, verification and configuration (whereas project reviews aim at acknowledging completeness of the activities demonstrating the fulfilment of the requirements).

When validation has been run on the version, the DRB can be merged with the Test Review Board [TRB].

DRB Deliverables depends on the agreement between provider and user, but includes at least the software release document (see 5.7.2).

5.3.5.2 Use of Technical Reviews to accommodate various life cycles

5.3.5.2.1 Introduction

Technical Reviews are a useful tool for the implementation of iterative lifecycles (and not only incremental as mentioned in the ECSS-E-ST-40C sub clause 5.3.3.3.c), such as incremental, evolutionary, spiral or agile (see clause 5.3.1). These life cycles spread the development in smaller waterfall-like steps. Running all the needed reviews as Project Reviews is potentially overloading the schedule and cost. Running some of them as Technical Reviews may allow achieving the objective while making it feasible within programmatic constraints.

There are many practices in projects. The whole review cycle may be applied for each version, or one review cycle may be spread over the versions, or the review cycle is applied only on the last version. In addition, the objectives of the reviews are not seen the same way by all reviewers in all the projects. However, each practice intends to save cost and schedule by tailoring the full formal process, while maintaining efficiency with respect to the review objectives and ensuring that the final quality is met.

5.3.5.2.2 A specific approach for flight software

In this section, the variety of practices cannot be presented exhaustively. Instead, some generic guidelines are given, followed by an example among others of a possible review plan.

The flight software development schedule is constrained by the overall spacecraft schedule. System requirements are not all available at the start of the software project development, and the AIT environment is in need of a software version before the full qualification of the software. Therefore, the flight software is often developed following an iterative approach (as defined in 5.3.1). Each iteration is usually called a "version". Indeed, ECSS-E-ST-40C clause 5.2.4.1 a. calls for the identification of software versions for software integration into the system.

As explained in 5.3.1, several constraints and factors influence the "versioning", i.e. the definition of versions and the associated life cycle. For example:

a. the maturity of system design, which indicates the priority in the development

b. the strategy and requirements in terms of integration and validation at higher level (AIT)

In addition:

c. if a review is skipped, then the objectives of the skipped review are included in addition to the objectives of the next milestone.

d. configuration and quality activities outcome and assessments are considered at each review and deliveries.

Here is an example of versions definition and review setup in an evolutionary life cycle:

a. V1 is intended to allow the integration of all electrical units on the Electrical Functional Model (EFM). Therefore, V1 contains the basic Data Handling functions (typically a limited number of the functions). The architecture is defined, as well as some part of the detailed design. The technical budgets are still estimated. V1 must be tested enough such that it is usable on the hardware.

b. V2 is intended to allow the run of most close loop tests on the EFM using most of the AOCS involved equipment units. Therefore V2 contains the AOCS and Safe Mode (V2 is now typically a large number of the functions). The detailed design is more complete. Technical budgets are confirmed with high level of confidence. V2 is unit tested and validated on the stable functions.

c. V3 is intended to allow the satellite qualification. Therefore V3 contains the complete software. The technical budgets are measured and proven. V3 is fully verified and validated with 100% coverage of requirements.

d. V4 is the final flight software. V4 contains corrections of anomalies from system qualification. V4 undergoes regression tests and representative mission simulation.

NOTE The software located in non-erasable memory (PROM) should be ready at equipment CDR. Therefore its lifecycle should be shorter. The boot software should have at least passed a CDR for V1 delivery and should have its QR before the SW V2 TRR, or before the starting of the qualification of the first flight model.

A possible review setup is the following (this example assumes a single TRR before CDR)

Table 5-1: Possible review setup

	V1		V2		V3		V4	
	Project	Technical	Project	Technical	Project	Technical	Project	Technical
SRR	X		X		X			
PDR	X		X			X		
DDR		X	X			X		
TRR				X		X		
TRB/DRB		X		X				
CDR					X			
QR					X			
AR							X	

In an iterative/evolutionary life cycle, a way to reduce the overall review effort is to merge some reviews or to run some Project reviews as Technical reviews. The above scheme allows running half of the reviews as technical reviews. The rationale of their selection is:

a. Requirements Baselines are updated for the first three versions, they are important in the customer supplier relationship as they define the baseline for the validation and the system interaction. Therefore they deserve three formal SRRs (project review).

b. Technical Specifications are relatively complete for V2 (if V2 includes 80% of the functions). The Architecture is baselined for V2; no changes are expected for V3. Therefore formal PDRs apply to V1 and V2, V3 PDR is a technical review.

c. Detailed design is not mature in V1, and quite complete in V2, therefore a single formal DDR is done for V2, technical reviews only for V1 and V3.

d. TRR importance increases from V1 (no review) to V2 and V3 (technical review)

e. There is no CDR, but simply a TRB for V1 and V2, to review the results of the tests before delivery, but there is a single formal CDR (including the TRB objectives) and QR for V3.

f. The flight version V4 affords the formal acceptance.

NOTE All the documents are made consistent for the QR and AR. All the documents are available for QR (in the final DP) and updated for the AR.

Another possibility to inject flexibility in the review scheme, is to balance the review objectives according to the version. When the development is incremental, the objective of the reviews can also be incremental. For example, some objectives are reached in the scope of a particular version (denoted V in Table 5-2). Some objectives are fully reached for the complete software development, even if they are achieved at a preliminary version delivery (denoted F in Table 5-2). Some objectives, once achieved for the whole project at a preliminary version, do not need to be modified in terms of deliverable in the next versions (denoted NM in Table 5-2).

Table 5-2 is an example of review objectives and their applicability to each version. This table is an example only that is subject to review and adaptation for each project.

Table 5-2: Example of review objectives and their applicability to each version

SW-SRR Objectives	V1	V2	V3
Agree with the customer or their representatives that all requirements captured in the requirements baseline are commonly understood and agreed w.r.t. completeness and coherency.	V	V	F
The software versions to be delivered are identified and each requirement of the requirements baseline is allocated either to HW or SW, and for SW is associated to a version.	V	V	F
Suitability of the software development plans including the software planning elements with respect to the upper level planning	F	NM	NM
Suitability of the software product assurance activities to ensure the development and verification will be performed taking into account the criticality of the software	F	NM	NM
Suitability of the configuration management environment to ensure coherency and reproducibility of developed products	F	NM	NM
Adequacy and proof of verification performed by the developer	V	V	F
SW-PDR Objectives	V1	V2	V3
Agree with the customer or their representatives that all requirements of the requirements baseline (including interfaces requirements) are captured in the technical specification	V	V	F

Agree on the verification methods proposed to confirm conformance of the product with the requirements and technical baseline	V	V	F
Verify the capacity of the design (including potential re-use) to implement the technical requirements and their associated performance needs and constraints	V	V	F
Verify the assumptions taken for the budget and margins estimations in order to ensure schedulability and performances achievements	V	V	F
Decide on known unsolved issues way forward	V	V	F
Confirm suitability of plans and development standards	F	NM	NM
Adequate verification performed by the developer including on potential re-use	V	V	F
SW-DDR Objectives	V1	V2	V3
Confirm the stability of the technical baseline for the development	V	V	F
Confirm the detailed design is clear and correctly captures the technical requirements and will allow an eased maintenance of the product thanks to the application of the design standards or if none at least to a homogeneous way of designing	V	V	F
Verify that the software units and integration plans implement an adequate strategy and coverage to fulfil the expected test level according to the criticality	V	V	F
Confirm the development environment is in place and ad hoc to start the coding and testing	F	NM	NM
SW-TRR Objectives for Validation vs. TS Testing	V1	V2	V3
Confirm the Software behaviour through the Code analysis, the Unit Test and Integration Results and other verification performed for quality assurance with regard to the software criticality - justifications that can be agreed are provided when the results are not as expected*		V	F
Confirm the foreseen validation/regression test cases (and possibly procedures), analysis, inspection or review of design will ensure the full coverage of the technical Specification applicable to the version		V	F
Verify that software documentation, software code, procured software and support software and facilities are under proper configuration control		V	F
Check the TS validation facilities readiness with regard to the tests objectives		V	F
SW-TRB/SW-DRB Objectives	V1	V2	V3*
Assess the validation/regression tests results with regards to the objectives presented during the SW-TRR	V	V	F*
Any observed deficiency is duly explained with proposed temporary workaround and accepted by the receiver with regard to its need	V	V	F*

Any late fix implemented to correct a blocking deficiency is duly explained, cross checked, validated and configured. Appropriate regression has been performed			F*
The complete definition of the product is clarified and limitations of use are identified	V	V	F*
Confirm the way to operate the software is correctly described and does not have any unexpected side effect on the operations			F*
* : covered by the CDR			
SW-CDR Objectives	V1	V2	V3
Confirm that the implemented product is fully coherent with its definition			F
Confirm that all the agreed verification and tests have been successfully performed and duly reported in order to ensure the required quality level and the technical requirements coverage			F
Assess and agree on the deficiencies (RFW) or known unresolved issues (SPR-NCR) which can have an impact on following phases and identify a resolution plan			F
Confirm the way to operate the software is correctly described and does not have any unexpected side effect on the operations			F
Confirm the foreseen validation/regression test cases (and possibly procedures), analysis, inspection or review of design will ensure the full coverage of the User Specification			F
Review all the configuration item to baseline the technical configuration w.r.t. TS-TRR configuration and delta justification			F
SW-QR Objectives	V1	V2	V3
Confirm the applicable baseline is correct including RFDs			F
Verify through the RB-validation tests, analysis, inspection or review of design results that the software meets all its user requirements, and that verification and validation process outputs enable transition to “qualified state” for the software products			F
Verify that the complete set of test is run on the same software version otherwise justified;			F
Confirm the correct closure of major SPRs/NCRs and the acceptability of RFWs			F
Review and baseline the item configuration w.r.t. RB-TRR configuration and delta justification;			F
Assess and agree on known unresolved issues (SPR-NCR) which can have an impact on following phases and identify a resolution plan			F
Review of the maintenance plan			F

SW-AR Objectives	V4
Identical to the SW-QR objectives after application of the resolution plan and correction/validation of any other NCR raised on the software during satellite test campaign and successful run of the mission representative test.	F

There is a link between version control and life cycle:

a. the Change Control can start at the first iteration in order to trace the changes of requirements from version to version. The reason is that the content of the reviews is impacted. For example, if a feature at V2 DDR is moved to V3, its coverage in the reviews needs to be clear. As another example, if requirements of V1 are changed in the middle of V1 development while preparing V2, it should be traced. It is highlighted that if a requirement which was fulfilled by V1 is changed after the delivery of V1, there is generally no need – in the frame of an iterative life cycle - to go back and change V1 - it is sufficient to fulfil the new/modified requirement in the final flight software.

b. the Customer should follow the Change Control Boards that will be run to baseline the various items of the versions, in particular the requirements. Once a baseline is approved for a version, the supplier is not allowed to change it outside of a Change Control Board to which the customer is invited, because any change impacts testing, documentation and schedule. However, in the early phase, the process of Change Control Board might be lightened, provided that the Customer is made aware of the changes.

5.3.6 Review phasing

5.3.6.1 Review phasing for flight software

Figure 5-4 identifies the relations between the system reviews and the flight software reviews.

For flight software, the software requirements baseline is reviewed (during the Software SRR) before the System PDR. The software is part of the system technical solution that is verified at the System PDR. The software RB defines how the software participates to the system technical solution.

The SWRR is a non-mandatory review anticipating the Software PDR addressing the software technical specification. It takes place between the Software SRR and the Software PDR. If possible, it should take place before the System PDR, in order to have a common understanding of the software at the beginning of the project phase C/D.

The sub system design can only start after the complete system design has been preliminarily baselined at System PDR. For this reason, the Software PDR follows the System PDR.

Inversely, all the software design activities are performed and reviewed before the System CDR, in order to have a complete view of the complete system solution at that milestone. This includes the Software PDR and the Software DDR if it takes place. The Software CDR can be held before, jointly or after the System CDR. In case there is no DDR, the software CDR is held at latest at System CDR for the reason just explained.

For flight software, the Software QR is planned within the System QR, as the system and the software cannot be qualified independently. The Software AR takes place before the System AR, as the system cannot be accepted if all its sub systems have not been accepted.

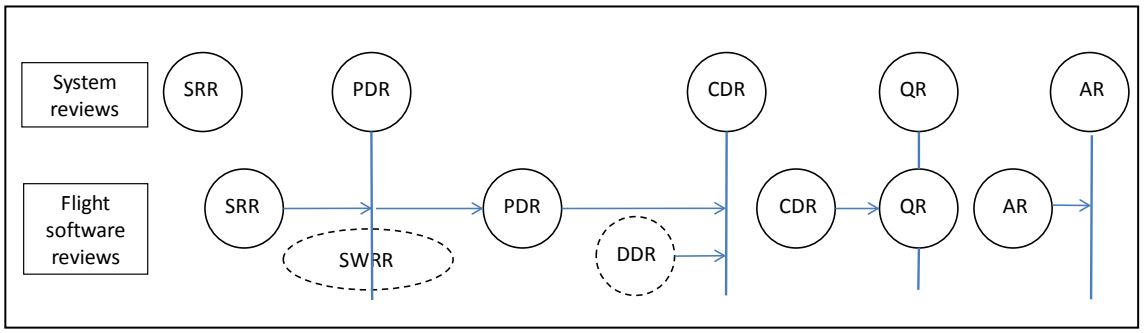


Figure 5-4: Phasing between system reviews and flight software reviews

5.3.6.2 Review phasing for ground software

Figure 5-5 identifies the relations between the ground segment reviews and the ground software reviews. The phasing between system and ground software reviews is very similar to the flight segment.

For ground software, it is required that the software technical specification be reviewed during the SWRR before the System PDR, if there is such a milestone. This would be eased for ground software as the overall Ground Segment high-level architecture (constituting elements, ICDs) is known in advance due to standardisation (e.g. SLE, PUS, etc.) and reuse of infrastructure (e.g. deep space stations).

All ground software reviews following the Software CDR are performed before ground segment QR.

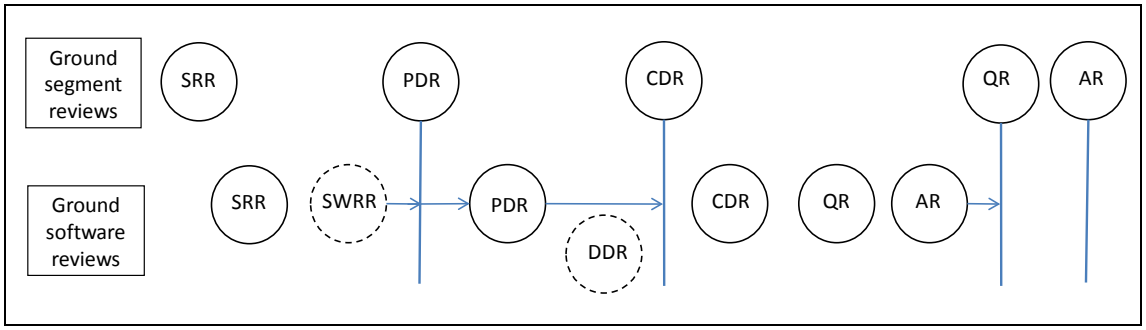


Figure 5-5: Phasing between ground segment reviews and ground software reviews

NOTE In case the ground segment software development is based on the reuse of an infrastructure based on a reference architecture (see 5.4.3.7.2), the SRR and the QR are not always conducted.

In addition, the operational phase devoted to the maintenance and evolution of existing infrastructure does not take place as such, but as part of the overall ground system.

5.3.7 Interface management

-

5.3.8 Technical budget and margin management

5.3.8.1 Software technical budget and margin philosophy definition

Technical budget and margin philosophy is discussed in 7.2

5.3.8.2 Technical budget and margins computation

Technical budget and margin philosophy is discussed in 7.3

5.3.9 Compliance to this Standard

See also 4.2.

5.4 Software requirements and architecture engineering process

5.4.1 Overview

The software requirements analysis is the first activity carried out during the software requirements and architecture engineering process. The main purpose of this activity is the establishment of the software requirements and their documentation in the SRS document as part of the technical specification (TS).

Software requirements are traced to the system requirements allocated to software from the requirement baseline established by the software related system engineering.

5.4.2 Software requirement analysis

5.4.2.1 Establishment and documentation of software requirements

5.4.2.1.1 Techniques

The Use Case technique can support this activity and is described in 6.1

5.4.2.1.2 Tools

In the last ten years requirements are increasingly specified and maintained in dedicated requirements engineering tools.

Requirement management tools usually contain some kind of database management facility. Expected capabilities of these tools are:

a. ensuring unique identifiers,

b. attach attributes (i.e. tool-defined or user-defined database fields) to requirements, such as status, importance, delivery version, verification method, creator, importance, risk, requirement category, creation date, last modification date, etc.

c. capturing, modifying, deleting and searching requirements and their attributes,

d. support cross referencing and traceability,

e. version control and revision history,

f. multi-user concurrent access, with access control

g. interface to documentation tools for document generation to produce requirement specifications,

h. view and multimedia facilities

For medium to large projects and especially for generic projects, a tool to manage requirements is invaluable. This especially applies for complex space systems where several different levels of decomposition are identified, and where requirements are specified and traced accordingly. In these circumstances keeping traces and relations manually becomes unacceptably error-prone and time-consuming.

5.4.2.2 Definition of functional and performance requirements for in flight modification

-

5.4.2.3 Construction of a software logical model

The ECSS-E-ST-40-C defines the logical model as reported in the following:

a. The logical model is an implementation independent model of software items used to analyse and document software requirements (3.2.13)

b. The logical model is a representation of the technical specification, independent of the implementation, describing the functional behaviour of the software product. It supports the requirements capture, documents and formalizes the software requirements. It also can support an iterative verification process with the customer.

c. The logical model allows in particular to verify that a technical specification is complete (i.e. by checking a software requirement exists for each logical model element), and consistent (because of the model checking).

d. The logical model should be defined using a language with well-defined semantics, which could possibly be executable. Formal methods can then be used to prove properties of the logical model itself and therefore of the technical specification. This does not preclude using any specific method or tool.

e. The logical model can be completed by specific feasibility analyses such as benchmarks, in order to check the technical budgets (e.g. memory size and computer throughput). In case the modelling technique allows for it, preliminary automatic code generation can be used to define the contents of the software validation test specification.

f. If software system co-engineering activities are considered, the logical model could be a refinement of the following system models: data, application function, event and failure

The ECSS-E-ST-40C clause 5.4.2.3 requires that the supplier construct the logical model.

The software logical model is a simplified model of the software application. It makes the software requirements understandable as a whole and not just individually. Depending on the language that expresses it, it can also be used for verification of completeness, consistency, proof or formal verification of some high level properties and generation of test scenarios.

A software logical model should be initially a simplified description that shows the high level essentials, that should show what the software system do, and avoid using implementation details. It should be defined more in depth for new complex functions, for instance, in order to assess the feasibility of requirements, than for reused functions or simple ones. The effort to increase the level of detail of the logical model should be balanced with the expected added value.

The model should be a communication bridge between the software and the system teams, enabling them to understand and agree requirements, and to anticipate the software requirements early within the system definition in the systems engineering processes related to software. This is of primary importance for software. Preserving the adopted system modelling method for software modelling, or assuring the continuity through appropriate model transformations with preservation of properties, represents an asset for the efficiency and correctness of the flow from requirements to implementation.

A model is simpler to understand and maintain when it is hierarchical, with consistent decomposition criteria, progressing through level of abstractions. It should be organized around a defined software component model approach (see 6.3.2.4).

The model shows the software functionalities and includes behavioural views. State transitions diagrams support the behavioural view representation that can be used also during the design activities. Alternatively Petri-Nets, SDL or other methods can be used.

The logical model can be built using different methods depending on the software functions (e.g. FDIR) or other characteristics (e.g. automata, modes). Possible methods and/or languages are:

a. functional decomposition, structured analysis, mathematics

b. object-oriented analysis (OOA);

c. formal methods;

d. use case and scenarios, e.g. as introduced in UML

See the Annex B for detailed descriptions of such languages and methods.

Although the authors of any particular method will argue for its general applicability, all of the methods appear to have been developed with a particular type of system in mind. Looking at the examples and case histories supports the decision whether a method is suitable.

The experience gives indication on which function benefits better of which method:

a. the control and guidance of satellites, producing cyclically actuation data from sensor data, are best modelled with mathematics, data flows or functional models

b. the mode management of a spacecraft, or the reconfiguration after failure, or some parts of autonomy management, are best modelled with state transitions diagrams.

c. data management systems, in their parts where data flow or state machines alone cannot naturally represent them, are better modelled with UML.

d. the high criticality of the software under development may suggest the application of formal methods.

NOTE Model based engineering is discussed in 6.3.

5.4.2.4 Conduction a software requirement review

-

5.4.3 Software architectural design

5.4.3.1 Transformation of software requirements into a software architecture

In the software architectural design process, the software requirements are transformed into an architecture that describes its top–level structure, identifies the software components, that ensure all the requirements for the software item are allocated to its software components, and later refined to facilitate detailed design. It covers as a minimum hierarchy, dependency, interfaces and operational usage for the software components, documents the process, data and control aspects of the product,

describing the architecture, static decomposition into software elements such as packages, classes or units, describes the dynamic architecture, which involves the identification of active objects such as threads, tasks and processes, describes the software behaviour (see ECSS-E-ST-40C clause 5.4.3.1).

The architecture describes the solution in concrete implementation terms. As the *logical model*, produced in the requirements analysis, structures the problem (what) and makes it manageable, the architecture defines the solution (how).

The architecture defines a structured set of software component specifications that are consistent, coherent and complete. A design method provides a systematic way of defining the software components.

The logical model is the starting point for the construction of the architecture. The design method sometimes provides a technique for transforming the logical model into the architectural design model. The designer's goal is to produce an efficient design that meets all the software requirements, as well as specific design requirements such as portability, degree of reuse, RAMS, etc.

In particular, the Use Case technique can support this activity to help identify software architectural components and trace to them. It is described in 6.1. More generally, model based engineering is discussed in 6.3.

The software architectural design describes:

a. the static architecture,

b. the dynamic architecture including the software behaviour.

5.4.3.2 Software design method

5.4.3.2.1 The static architecture

The design method is expected to outline the static view of the design, i.e. decomposing the software into lower level software components, in a top-down approach, by describing also the external and internal interfaces, the relationships, and the information held by each component.

It starts by defining the top-level software components architecture and then proceeds to define the lower-level ones, down to the software units. This view is static because it does not describe the time-dependent behaviour of the software, which is described in the dynamic view.

The top-down approach is vital for controlling complexity. Even if a design method allows other approaches (e.g. bottom-up), the presentation of the design is always top-down. In fact a hierarchical presentation makes the model simpler to understand (i.e., fewer elements are disclosed at a time, from generic to specific).

The way components are decomposed into lower elements is what differentiates the various design methods. A number of design methods are described in Annex B.

5.4.3.2.2 The dynamic architecture

The software architecture and design also describe the dynamic aspects of the design. This is particularly important when concurrency is an issue or for event-driven software (both typical of reactive software).

The dynamic aspects involve the identification of:

a. active components, such as threads, task and processes,

b. shared resources and associated protection mechanisms,

c. the communication patterns of the connected active components (e.g. synchronization),

d. the way active components interact to implement behaviour and the evolution of each of them as it responds to events based on its current state (e. g. protocols, state transitions).

The dynamic design is based on the selected computational model and provides also the temporal attributes for real-time active objects, which enable schedulability analysis to be undertaken. Some temporal attributes may be concerned with mapping timing budget and performance requirements onto the design (e.g. the deadline), other are attributes that are defined as design parameters (e.g. the period of execution), or set as schedulability analysis parameters (e.g. the worst case execution time). Further details are provided in section 7.4.3.

The software behaviour is intended as the (sequential) description of the internal functions or operations that are provided by the software components or units, identified in the static architecture.

To describe the software behaviour, it is possible to use the same behavioural description techniques that are used in the analysis phase for the logical model, such as automata, Petri-Nets, State Charts or interaction scenarios.

The behavioural term can be intended to describe the observable behaviour of the components that execute concurrently in the dynamic design. This meaning is part of the dynamic architecture.

5.4.3.3 Selection of a computational model for real-time software

The computational model is discussed in 7.4.

5.4.3.4 Description of software behaviour

-

5.4.3.5 Development and documentation of the software interface

-

5.4.3.6 Definition of methods and tools for software intended for reuse

5.4.3.6.1 NASA study on flight software complexity

The NASA Complexity Memo (Final Report - NASA Study on Flight Software Complexity - Commissioned by the NASA Office of Chief Engineer, Technical Excellence Program, Adam West, Program Manager Editor: Daniel L. Dvorak, Systems and Software Division, Jet Propulsion Laboratory, California Institute of Technology) reports on several studies, analysis, enquiries, that have been done on software complexity, which are worth quotes and summary in this handbook.

First, some quotes about software complexity:

Complexity is a measure of understandability, and lack of understandability leads to errors. A system that is more complex may be harder to specify, harder to design, harder to implement, harder to verify, harder to operate, risky to change, and/or harder to predict its behaviour. […] It's important to distinguish between "essential complexity" and "incidental complexity". Essential complexity arises from the problem domain and mission requirements, and can only be reduced by descoping. Incidental complexity arises from choices made about hardware, architecture, design and implementation, and can be reduced by making wise choices.

Essential complexity can be moved but not removed (except by descoping). For example, a decision to keep flight software simple may simply move some complexity to operations, along with attendant costs and risks. […] Architectures, which are specifically designed to enable flight/ground capability trades, are critical to moderating unnecessary growth in FSW complexity.

A source of complexity is due to the fact that software can be changed easily therefore it absorbs complexity, it is a complexity sponge.

Complex system have been defined succinctly as follows: A system is classified as complex if its design is unsuitable for the application of exhaustive simulation and test, and therefore its behaviour cannot be verified by exhaustive testing (Defence Standard 00-54, Requirements for safety related electronic hardware in defence equipment, UK Ministry of Defence, 1999).

The avionics hardware can be source of complexity, in particular:

a. *Interfaces that do not provide the option of interrupt-driven, forcing polling. You can always turn off the interrupts if you want polling, but if they're not there in the first place the bridge is burned.*

b. Interfaces that do not provide atomicity, that is to say several ports need to be manipulated separately to get the desired result. Especially bad if timing tight (forcing critical sections) or no way to insure the operation actually worked.

c. Distribute functionality to remote controllers to reduce real-time demands on the central processor.

Second, some quotes about the importance of architecture in complexity:

Architecture is about managing complexity. Good architecture—for both software and hardware—provides helpful abstractions and patterns that promote understanding, solve general domain problems, and thereby reduce design defects.

Unnecessary growth in complexity can be curtailed in particular by:

a. *Modular software architecture to isolate concerns and allow composition of individually tested elements.*

b. Good architecture rooted in the fundamental mission objectives.

c. Design fault protection early, otherwise it is added on piecemeal, distorting the architecture a little more with each addition.

d. Use design patterns to capture sound solutions to recurring engineering needs, so less time spent reinventing wheel, freeing resources to work on more important things.

Necessary growth in complexity can be better engineered and managed through architecture:

a. *Build a flight software architecture that minimizes the incremental cost of adding functionality, or modifying functionality.*

b. Spend up-front time getting the architecture right. The key properties of the architecture are modularity (isolating pieces of the system from other pieces of the system), layering (to provide common ways to do common things), and abstractions (to provide simple ways to use to capabilities with complex implementations).

c. Do architecture up front. The architect needs real authority to protect the architecture, otherwise short-term cost and schedule pressure will lead to decisions that corrupt the architecture, to the detriment of lifecycle costs. According to one definition, architecture is the set of design decisions that are made early in a project, and for which the consequences of a bad decision only appear much later.

The report includes several advises related to architecture:

a. *Use component-based architecture, with components as the element of reuse, and connectors. Configuration manages the component specifications and API's to manage costs of evolution, testing, etc.*

b. Model the behaviour of key aspects of the system (state transitions, timing) to avoid finding out what doesn't work after it's built.

c. Don't dictate architecture to designers. Focus on the bottom line: delivery of an independently testable, verifiable product.

d. Have one architectural leader.

e. Establish and stick to patterns.

f. Invest in reference architecture

g. Separation of concerns

As a conclusion:

"Good software architecture is the most important defence against incidental complexity in software designs, but good architecting skills are not common. From this observation we made three recommendations:

(1) allocate a larger percentage of project funds to up-front architectural analysis in order to save in downstream efforts;

(2) create a professional architecture review board to provide constructive early feedback to projects; and

(3) increase the ranks of software architects and put them in positions of authority. "

5.4.3.6.2 Complexity assessment

One way to assess the complexity of architecture is to identify and to analyse the dependencies between the elements of the architecture. The analysis should then highlight architectural or structural patterns and potential flaws, weaknesses or critical elements in the system architecture. Metrics related to the dependencies can also help measuring the architecture complexity, provided they are correctly interpreted.

Analysis of dependencies

The analysis of system dependencies is also referred to as structural analysis. The system architecture is decomposed in an ordered, hierarchical way. The dependencies between the system components are identified. This system decomposition and dependency analysis can be done at model level during design, and then confirmed at code level after implementation.

There are two classical representations of the dependencies of a system.

a. One is a graph representation where the architecture elements are displayed with relationships between them. The UML equivalents are the component and composite structure diagrams. Graphs are very intuitive, and hierarchical decomposition can be used to master scalability.

b. The other representation is a matrix representation called DSM (for Dependency Structure Matrix or Design Structure Matrix). It has two main advantages. First, it can represent a large number of system elements and their relationships in a compact way that highlights important patterns in the architecture. Second, it allows matrix-based analysis techniques, which can be used to improve the structure of the system.

Structural patterns

The analysis of dependencies between architecture components should lead to the identification of structural patterns and anti-patterns. Structural patterns are design patterns that make the design more understandable and less complex by identifying a simple way to realize relationships between the architecture components. Design patterns are widely described in literature. Their counterparts, the anti-patterns, are either ineffective or counterproductive patterns.

Examples of structural patterns are:

a. Layered structure: there is no dependency cycle in the model.

b. High cohesion and low coupling structure: this refers to a set of components that have a high cohesion between them and low coupling with the other components of the architecture. They are good candidate to be grouped in a parent artefact.

Examples of structural anti-patterns are:

a. Cyclic dependency: there is a dependency cycle in the model. A cyclic dependency between components is considered a major architectural flaw. Such a dependency makes the code difficult to understand and maintain. More importantly, cyclic dependencies undermine testability, parallel development, and reuse.

b. Popular element/Butterfly: many other components depend on the popular element. It is not necessarily a flaw, but the component should then be a low-level system interface or utility component.

c. Global breakable: a component that depends on many other components, and is thus affected when these components are changed. They are undesirable because they indicate fragility and lack of modularity in the architecture.

d. Local breakable: a component that depends immediately on many other components. Such a component carries excessive responsibility and is typically identified by many long methods. They make the system difficult to understand, to maintain, and to reuse.

e. Global hub: a component that has both many dependencies and many dependents. It is often affected when any other component is modified, and it affects a significant percentage of the system when it changes. They are undesirable because they indicate fragility and lack of modularity in the architecture.

f. Local hub: a component that has both many immediate dependencies and many immediate dependents. Such a component carries excessive responsibility and also serves as a utility or commonly used component. They make the code difficult to understand, to maintain, and to reuse. They also make the code fragile and unstable.

Architecture Complexity Metrics

Coupling and Cohesion are the two terms which very frequently occur together. They relate to interactions and interdependencies between software design components. Together they talk about the quality a design should have.

Coupling is based, for a given software component, on the measure of the components on which it “depends” (this number is the used components, the "fan-out" of the module). Low Coupling is advised. It refers to a relationship in which one component interacts (e.g. "uses") with a few number of other component(s) (e.g. only one) through a simple and stable interface and does not need to be concerned with the other component’s internal implementation and is very slightly impacted by their evolution. Low coupling is often a sign of well-structured software, a good design which finally increases maintainability and reusability. In the contrary, a high coupling is indicative of a high degree of interdependencies. In general, the higher the coupling, the poorer is the overall design.

Cohesion refers to the degree to which the elements of a module belong together, it is a measure of how strongly-related each piece of functionality expressed by the source code of a software component is. High Cohesion implies simplicity and high maintainability and reusability. This generally contributes to a better design. The designer should strive for high cohesion at the lower levels of the hierarchy. For example, there could be common low-level functions that are identified and made into common components to reduce redundant code and increase maintainability. High

Cohesion can also increase portability if, for example, all I/O handling is done in common components. In the contrary, low cohesion implies that components perform tasks which are not very related to each other and hence can create problems as the component becomes large. The utility of such component could be reconsidered.

Coupling is usually paired with cohesion: low coupling often correlates with high cohesion, and vice versa. Low coupling, when combined with high cohesion, supports the general goals of high readability and maintainability. Such principles are reinforced by object-oriented principles.

5.4.3.6.3 Reference architecture

In accordance with the NASA report, there is a clear trend in software systems to give more importance to (reference) software architecture. The emergence of actual reference architectures (Automotive AUTOSAR, Space SAVOIR, ISIS and EGS-CC for ground software) highlights the need to base software developments on sound, long term architecture, in particular to:

a. favour reuse of components

b. facilitate future evolution

c. standardize their interface

d. facilitate integration and testing by separation of concerns, in particular complete separation of the functional and the non-functional aspects (in particular real-time and dependability). The ease of composition is obtained by the composability and compositionality properties.

NOTE Separation of concerns is the process of breaking a computer program into distinct features that overlap in functionality as little as possible. Coined by W. Dijkstra in his 1974 paper "On the role of scientific thought", the advent of MDE however eases the definition of concern-specific views and helps prove the benefit that their separation may bring.

e. support the concept of correct by construction by providing architectural design whose properties can be verified formally. In particular support the concepts of composability (the components keep their local properties when assembled) and compositionality (the local properties of assembled components provide a system global property) [see section 5.4.2.4.]

NOTE Correctness by Construction (C-by-C) is a software production method that fosters the early detection and removal of development errors to build safer, cheaper and more reliable software.

The motivation is the reduction of the project risks linked to its software architecture. Even if an architecture is not aiming at reuse, it should however implement the concepts of "*separation of concerns*", "*composability*", "*compositionality*" and "*correct by construction*", in order to facilitate the integration, to structure the tests (per concerns) and to replace some tests campaigns by earlier verification of design properties (such as schedulability analysis).

Architecture is the fundamental conception of a system in its environment embodied in elements, their relationships to each other and to the environment, and principles guiding system design and evolution. Reference architecture is a single, agreed and common solution for the definition of the software architecture of a set of software systems whose domain of variation is identified.

The reference architecture is produced from the application of the domain engineering processes [ISO12207]. The domain of reuse is defined by analysing various standpoint of the future system (e.g. existing technology, expected future needs, trends). An important result of the domain engineering is the variability factors. They are the elements that changes from a system to another system in the

domain of reuse. They are important design driver for the architecture, as the impact of their change should be minimized and localized in the architecture.

Reference specifications (at the level of a requirement baseline) are derived within this domain. They include the variability that the products should feature to be reusable in the domain. They constitute the Generic RB of the system. They address the architecture and the generic functions of the system. From the architectural part of the generic RB, the architecture is established, which also takes into consideration the variability of the domain of reuse. The variability is reflected in the system data base (see 5.2.4.4). This architecture becomes the reference for the domain. It includes in particular reusable elements (within the domain) called building blocks. The building blocks implement the functional part of the generic RB.

A specific part of the reference architecture addresses system functionalities which are common to all the systems belonging to the domain of reuse, for example the communications, the scheduling or the operating system. Therefore this part will be subject to a common part of the reference RB extracted from the existing RBs of the domain of reuse.

For examples potential flight software reference architecture would implements, in a part named “execution platform”, the avionics system communications and the real-time scheduling. The mission specific application software would be expressed as components calling the services of the execution platform.

As a conclusion, the major design drivers of architecture with respect to complexity are:

a. containment of the effect of a change of one of the variability factors of the domain of reuse,

b. systematic implementation of the "separation of concerns", in view of decoupling as well the verification and testing concerns,

c. identification of design patterns allowing the definition of design properties that can be verified early, aiming at a "correct by construction" design.

Reducing complexity may have higher priority than system performance. Seen from a pure software engineering or project management standpoint, it is beneficial to trade performance against complexity, and to suggest hardware resources increase in order to allow for software complexity decrease.

5.4.3.7 Reuse of existing software

5.4.3.7.1 Reference

Reuse of existing software is address in the ECSS-Q-HB-80-01A handbook.

5.4.3.7.2 Reuse in the Scope of Generic Reference Architectures

Context

The notion of reference architecture has been introduced in 5.4.3.3, with the viewpoint of architecture complexity reduction. It is addressed here with the viewpoint of the reuse process and the adaptation of the review process to reused architecture.

One of the underlying examples of Reference Architectures addressed in this section is ground data systems of many missions, these are typically composed of the same set of components with the same standardized interfaces.

Another example is the reference architecture for an operational simulator, this defines what simulation models an operational space mission simulator is composed of, and which interfaces each

model provides/consumes. The experience in building software from flight reference architecture is more limited than for ground.

A Reference Architecture, in short RA, is a reusable system design pattern, which assigns the required functionalities of a complex system to a predefined set of composing components.

In other words the RA specifies which components a system is composed of, and specifies standardized interfaces between those components. The implementation aspects of the functionalities assigned to each component of the composed system are however not in the scope of a RA.

It is highlighted that Reference Architecture can define so called RA profiles, which allow choosing from more than one fixed set of components depending on the profiles of the shared system requirements.

Impact on ECSS-E-ST-40C

In case of instantiation of a system from a Reference Architecture, the ECSS-E-ST-40C RB and TS elicitation processes is translated in the following way:

The RB of the new system will be composed of the reference Generic RB, plus a mission Specific RB. The mission Specific RB will be implemented either as instantiation of existing building blocks (further described in a TS), or as new elements.

There are a number of architectural activities that are performed once in the scope of the generic reference architecture elaboration. These activities are reused at the time of mission deployment. Therefore part of the RB, TS and architectural process are also reused.

The life cycle of the reference architecture and of the specific instances can ultimately be separated, and there may even be different development teams. The reference architecture team delivers an architecture baseline to the specific development team.

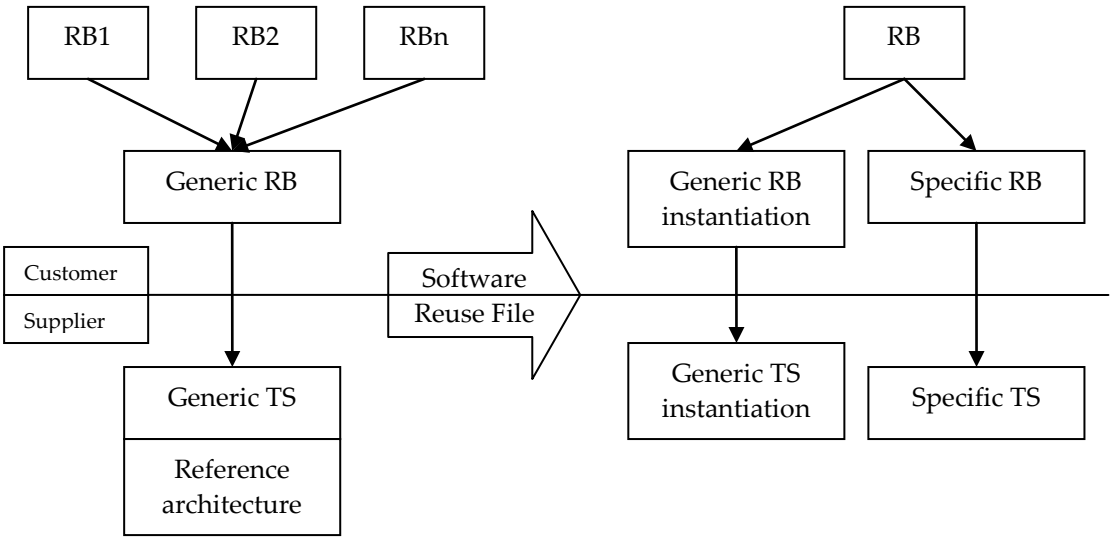


Figure 5-6 : Reuse in case of reference architecture

The ECSS-E-ST-40C process applies both for the generic and the specific parts. The Customer of the specific system is expected to place reuse constraints on the design and development coming from the reference architecture.

Considering that SW restarts from existing documentation for RB, TS and ADD, the SRF (ECSS-E-ST-40C Annex N) is important to explain the overall reuse approach at SRR and PDR. Moreover, the concept of SQSR (Software Qualification Status Review) can be used to support the specific review of the SRF.

Specific tailoring

It is highlighted however that in some cases, the Specific RB could be relatively limited, and could be merged with the Specific TS. In this case, it is important to handle a SWRR. The SWRR is merged with the SRR. As a consequence of merging Specific RB and Specific TS, the QR is merged with the CDR.

This is in particular the case for some mission control systems of the ground segment development. The system design can be fixed once (in form of a RA) for a group of similar systems, which share a set of system requirements.

Actually, an infrastructure for ground segment is developed from the common parts of Ground System Requirement Documents (considered as RB). A separate team implements the mission specific features which are added to the configured infrastructure directly from dedicated SRS (TS).

5.4.3.8 Definition and documentation of the software integration requirements and plan

-

5.4.4 Conducting a preliminary design review

-

5.5 Software design and implementation engineering process

5.5.1 Overview

-

5.5.2 Design of software items

5.5.2.1 Detailed design of each software components

-

5.5.2.2 Development and documentation of the software interfaces detailed design

-

5.5.2.3 Production of the detailed design model

NOTE Model based engineering is discussed in 6.3.

5.5.2.4 Software detailed design method

-

5.5.2.5 Detailed design of real-time software

The dynamic design describes the behaviour and the interactions of the different processes or tasks of the software system and the ways shared resources are accessed.

At software level, the following real-time aspects should be identified in the dynamic design:

a. High-level interaction schemes (client-server, asynchronous/synchronous communication, scheduling scheme and priorities between tasks).

b. Shared resources (shared memory, hardware device, communication bus).

c. Real-time constraints (maximum allowed jitter, maximum CPU usage, maximum response time).

d. Dynamically allocated resources (memory, task).

The real-time software dynamic design model should identify the design elements related to the real-time aspects in order to enable the verification of the timing requirements and to assure the predictability of the system. Analysis of these real-time elements (following the chosen computational model) should permit to determine whether the timing constraints on the system can be satisfied. It should also be demonstrable by analysis that there are neither race conditions, nor deadlocks.

The real-time software dynamic design model identifies:

a. All system tasks. For each task, the activation characteristics (frequency or minimum inter-arrival time), the priority and the accessed shared resources should be defined. The task could have additional attributes, such as the activation phase.

b. Cyclic activation signal (external signal, processor clock).

c. Task communication means, such as shared memory, mailboxes/message queues, signals.

d. All handled and non-handled interrupts. For each interrupt, the frequency (or minimum inter-arrival time), the priority and the accessed shared resources should be defined.

e. The scheduling type (e.g. sequential or multi-task), the scheduling model (e. g. cyclic or pre-emptive, fixed or dynamic priority based), and the scheduling algorithm (e.g. Fixed-priority pre-emptive), under which the system is executed.

f. Task synchronization mechanisms if any – both inter-task synchronization mechanisms and mechanisms for synchronization of tasks with external signals.

   EXAMPLE: a task dealing with an external signal waiting on a semaphore could be woken up by releasing the semaphore from an interrupt handler.

g. All mutual exclusion mechanisms to manage access to the shared resources (e.g. semaphores, task locking, interrupt locking).

h. All protection mechanisms against problems that can be induced by the use of dynamically allocated resources, e.g. memory leaks, non-deterministic response time…

i. Means of (processing) distribution and inter-node communication (e.g. virtual nodes, Remote Procedure Call, ORB).

The dynamic design may also include specification of:

a. the means of providing timing facilities (e.g. real-time clock, with or without interrupt, multiple interrupting countdown counters, relative or absolute delays, timers, time-out).

b. the means of providing asynchronous transfer of control (e.g. watchdog to transfer control from anywhere to the reset sequence, software service of the underlying run-time system to cause transfer of control within the local scope of the task)

Several methodologies enforce the consideration of real-time aspects within the architecture (see annex B).

Real-time software is discussed in section 7.

5.5.2.6 Utilization of description techniques for the software behaviour

-

5.5.2.7 Determination of design method consistency for real-time software

The computational model allows to consistently relate entities of the software architecture to abstractions of the computational model. This also requires an adoption of design and coding rules (i.e. a programming model) which is compliant with the computational model and which does not allow the use of language and architecture idioms, style and patterns (e.g. interactions with RTOS) that do not conform to the computational model. In the opposite case, incompliance of the programming model with the execution and concurrency semantics defined in the computational model would undermine the assumptions made by the schedulability analysis.

5.5.2.8 Development and documentation of the software user manual

-

5.5.2.9 Definition and documentation of the software unit test requirements and plan

-

5.5.2.10 Conducting a detailed design review

-

5.5.3 Coding and testing

5.5.3.1 Development and documentation of the software units

-

5.5.3.2 Software unit testing

5.5.3.2.1 Introduction and common understanding

NOTE Testing methods and techniques are also addressed in 6.4

This chapter recalls the major principles of the unit tests and extends the basic approach in the frame of the overall software development and validation logic.

The initial objective of Unit Testing is to check the correctness of the implementation of a software unit (at source code level) with respect to its definition into the corresponding SDD-DDD, i.e. the correct behaviour of a function and related data in the domain of its input parameters.

It consists at specifying tests in the SUITP (and then developing appropriate test code) which exercise the function to be tested in its data domain, and monitor that its behaviour and results are compliant with the expectations, as defined in SDD-DDD. When necessary, the lower level interactions need to be stubbed to simulate their behaviour, or replaced by already tested functions.

Unit tests are run as early as possible in order to gain sufficient confidence in the source code to allow starting integration then validation against the TS while minimising further debug effort.

Unit test is sometimes the only way to perform some specific validations, e.g.:

a. testing interfaces with the computer or the BIOS (e.g. HW drivers) on a facility representative of the HW/SW interfaces;

b. validating some TS requirements that are not reachable with a complete SW. In this case, the unit test facility includes an OBC emulator. Such requirements are tested on a specific test executable needing the knowledge of the design;

c. validating a SW function with a very large combination of data, which is hardly feasible on complete software. In this case, the unit tests are performed on internal mechanisms in a simpler way.

or some low level verifications, e.g.:

a. testing some mechanisms that contribute to the real-time behaviour of the software (e.g. semaphore, interrupts, buffer management);

b. testing the boundaries of configuration / mission data: when the definition of some data is not included in the source code (e.g. for mission data, family of spacecraft), a database is used but the database values are not used by the tests. Instead, test values are defined to cover the functional range of these mission data.

5.5.3.2.2 Unit Testing and Code Coverage

The code structural coverage technique (e.g. statements, decisions) is commonly associated to Unit Testing. It is recognised that achieving the code structural coverage objectives, largely contributes to verify one of the Unit Testing objectives (i.e. exercise the complete code). Moreover, the source code structural coverage measurement, fulfils an additional verification objective on the overall software, providing a higher confidence in the software reliability.

However, the code structural coverage objectives can be achieved by other means other than Unit Testing. Thus, as highlighted in the note of ECSS-E-ST-40C clause 5.8.3.5 b, this source code structural coverage analysis can be performed by running any kind of tests (e.g. validation tests), measuring the code coverage, and achieving the code coverage by additional (requirements based) tests, inspection or analysis. This approach is particularly efficient because the functional testing approach generally allows reaching quickly a high level of code coverage for a reduced effort. This strategy is recommended when using adequate validation facilities (i.e. allowing code instrumentation) and leads to largely reconsider the basic and systematic Unit Testing approach.

Moreover, achieving the code structural coverage contributes to find unused or unreachable code.

5.5.3.2.3 Areas for optimization or adaptation

Overview

The systematic application of unit tests or the very deep unit tests (fine grain) require a huge effort, potentially not compatible with the project class, the project schedule, or project needs. Sometimes, the level of the unit tests needs to be balanced according to the criticality of a function, of a component or of the software itself: the granularity of the unit tests contributes to the expected software reliability level. Sometimes still, tests at unit level are fully redundant with other validation tests or it can be considered as more efficient to reduce the unit tests effort and to start validation earlier, while accepting discovered faults during the validation phase. Hence, the standard Unit Testing approach needs to be adapted or optimized; the overall objectives of Unit Testing introduced above, can be achieved through various combinations according to the project context and the software characteristics.

This part presents different areas for adapting or optimising the systematic Unit Testing approach, highlighting their strengths and weaknesses in order helping the assessment of the risks with regard to the project context. The adaptation or optimization of the Unit Testing approach need to be assessed early in the project according to the criticality level, the development schedule, or the deliveries expectations (i.e. the validation or reliability level of a delivered version). On the basis of this risk analysis, the Unit Testing strategy can be built and hence the way to achieve the Unit Testing objectives can be optimised. The defined strategy as well as the appropriate rationale, need to be documented (e.g. in the Software Development Plan or in the SUITP) and agreed together with the Customer and Quality representatives.

Unit testing adaptation according to the Software Criticality level

The unit test technique is based on the direct exercising of the source code, and particularly can use the opportunity of a "white box" approach. Hence, it allows an in depth assessment of the correct behaviour of a function. However, this requires a huge effort which may be balanced according to the overall project real expectations.

ECSS-E-ST-40C requires unit test for all levels of criticality. If a project considers, as a matter of tailoring, that the unit test effort needs to be focussed, then the criticality criteria may be used in the following way. The criticality level of software units is evaluated (e.g. according to ECSS-Q-HB-80-03A). The unit tests required by ECSS-E-ST-40C 5.5.3.2c are applied to a different extent (e.g. a partial application of the requirement w.r.t e.g. testing depth) depending on the unit's criticality or its contribution to critical functions. It is highlighted that ECSS-Q-ST-80C defines requirements relevant to software criticality classification; in particular, in order to classify software components at different criticality levels, software failure propagation between components of different criticality must be prevented (ECSS-Q-ST-80C, sub clause 6.2.3.1.).

The full application of the ECSS-E-ST-40C 5.5.3.2c requirement is particularly important for:

a. highly critical software (e.g. overall criticality A),

b. a limited set of highly critical software units within a critical software,

c. software units contributing to highly critical functions.

Specific testing methods and techniques that can contribute for the assessment of critical software products are presented in Annex B.

Unit testing optimization

Unit testing is one of the means at developer level to reduce the risk to discover faults during validation. Other means are introduced here.

A first example is to reconsider the unit test objectives in the frame of the overall software verification and validation. Some unit tests are performed on selected units. Then functional tests are run, and structural coverage is measured. At this point, most of the source code functions are checked on a representative set of their input parameters. The weakness is that evidence is not provided that the set of input parameters is completely covered. To bring this evidence, and to achieve the complete structural coverage, complementary unit tests are performed, or this approach is completed with e.g. the use of "abstract interpretation" tools, run in a systematic way, including the subsequent correction of the potential errors.

This approach intends to achieve unit testing objectives of a part of the units by functional tests instead of classical "unit tests". This is equivalent to performing test of a unit, not stubbing the external units, but using the real components.

Nevertheless, tricky behaviours are easier to test out of the functional context and typically, unit tests remain particularly efficient to check robustness code or error cases, which are difficult to exercise through functional tests. Unit tests are also performed for units having high measured complexity metrics.

A second example is, in order to limit the risk of discovering faults during the validation, to improve the maturity of the source code by early source code peer reviews.

As a third example, it is also recommended to use powerful simulators during validation, including high level investigation capabilities at software implementation level.

Another example consists in applying the tests on a group of software units. The principle is to perform unit testing with regard to the software design elements (i.e. components or set of components) on the produced software units or a set of software units. Unit test cases are not systematically defined at each software unit level. This optimization level is generally reached combining the functional view of the TS and the design view of the SDD. The functional view helps defining test scenarios and the design view helps defining the set of units to be tested together. Both views (functional and design) also allow covering integration tests objectives (see next chapter) by checking that information exchanged between the design items set up through the functional tests is correct. The 'function' defined in this testing approach is often close to functions defined in the SRS, e.g. for an on-board central software a PUS service, an equipment management, an AOCS attitude estimation function or a thermal regulation loop.

With this approach, the external interfaces of the function are tested in order to ease the further integration with the rest of the software. The internal operations of the function are implicitly tested during the functional tests contributing to the integration tests objectives.

5.5.4 Integration

5.5.4.1 Software integration test plan development

NOTE Testing methods and techniques are also addressed in 6.4.

5.5.4.2 Software units and software component integration and testing

5.5.4.2.1 Introduction and common understanding

This chapter recalls the major principles of the integration and extends the basic approach in the frame of the overall software development and validation logic.

The external interfaces are specified in IRDs, their implementation is defined in dedicated ICDs. Internal interfaces are in the SDD only.

The objectives of the integration are:

a. to build the overall software from software items (building logic),

b. to ensure that the interactions (e.g. interfaces, real-time behaviour) between software items and with external interfaces (e.g. TM/TC interfaces, HW/SW interfaces), are consistent together and also with respect to their definition.

The integration consists in the physical and progressive assembly of software items, corresponding to design items defined in the SDD-ADD, up to the complete software. These software items are software modules or packages, groups of software modules, building blocks, or software executable. The building logic describes the order and priorities of assembly of the software items in a structured way, in the frame of the project to build one or several software executable.

In a life-cycle, the integration is the stage in which individual software items are combined and tested as a group. In principle, this stage occurs after unit testing and before validation testing. Integration takes as input the software items that have been unit tested, groups them in larger assemblies, applies tests to those assemblies, and delivers as output the integrated software ready for validation testing.

The integration strategy is made consistently with the static architecture of the software. When the software is made of items with "use" dependencies from user items to used items, "bottom-up" assembly logic can be applied, from the lowest level items until the top of the hierarchy. Each of the intermediate items can be tested as an item with stubs for the used items when necessary (e.g. for error cases) or with the actual item when the actual item behaviour is needed. Generally, all the upper level items are used integrated with the actual intermediate one.

5.5.4.2.2 Areas for optimization or adaptation

This standard integration approach generally needs to be adapted or optimized and, in some circumstances, the overall integration objectives introduced above can be achieved through various combinations according to the project context, and the software characteristics. The project development and validation strategy can completely drive the integration strategy.

The building logic and the objective to cover the interfaces can be independent. Moreover, the assembly and the integration testing can be split throughout the validation stages of the overall life-cycle.

The integration may largely overlap the different increments, i.e. starting after the unit testing of the first increment, and finishing before the validation testing of the complete software. In particular for an evolutionary software development, each intermediate software version corresponds to an intermediate step in the integration process. This implies that the definition of the intermediate versions and the overall software integration strategy need to be consistent.

The building logic also depends on the type of software, the use of existing building blocks, the type of software items (e.g. HW-related components, applicative items, automatically generated code), if items are a group of modules or a set of executable, etc... For complex software systems (e.g. mission centres, distributed software), the building logic consists in the assembly of already validated software tools, building blocks, database and HW and computers. The complexity of such software requires a deep description of the building logic (e.g. in the Software Development Plan). In the opposite, for more simple software reduced to one executable and using high level language or the interfaces typing, the building logic may be highly simplified, sometimes reduced to compilation and links editing (makefile).

The objective to ensure the internal and external consistency of software interactions, usually based on dedicated integration tests, can be partly or completely achieved in other way and particularly through verifications, validation or other existing tests (e.g. unit, functional, HW/SW, real-time), while it can be demonstrated that existing tests allow the interfaces check and coverage. Thus, the common unit and functional tests objectives can be extended to achieve the integration testing objectives.

The way to achieve this objective is described in the Software Validation Plan, and ECSS-E-ST-40C can be tailored according to the different criticality levels of the software parts (e.g. systematic and complete test of interfaces for a boot, an autonomous or highly critical function, simple coverage of function calls for less critical and patchable parts).

Sometimes, when the considered software only consists in an executable developed with a strong typing-language or through dedicated coding rules, the static interfaces consistency is warranted by the compiler and then, the interfaces exercised by the unit and functional tests can be collected, then checked to assess the coverage.

Moreover, some tools are able to automatically check the interfaces consistency (functions, parameters types, names or number) that also contributes to this objective.

When the definition of the interfaces is done by mean of tools (e.g. centralised interfaces/data allowing the generation of both interfaces code and ICD), interfaces inconsistencies are very limited and the interfaces verification is easier, or even completely proven when using formal languages (e.g. ASN1, Corba) for data description (see Annex B). In these cases, it is recommended to use editors, checkers and document producers:

a. A specific editor to support the user describing the interfaces without any specific knowledge of the syntax required by the language. The result consists in a formal ICD.

b. A document producer that translates the formal definition into a readable format, e.g. HTML, Word or Excel. The result consists in a readable ICD.

c. A data checker that controls the data compliance with the formal ICD. It is used both by the data producer software and by the data consumer software.

Finally, according to the software items purpose, integration testing can be made through various means (including inspection) and facilities when test is chosen. For example, for a component managing the input/output of the software, a Hardware bench can be used together with the lower level software (i.e. BIOS, O/S).

5.6 Software validation process

5.6.1 Overview

The ECSS-E-ST-40 Standard on Software requires that software be verified during the development life cycle and validated before delivery.

Validation is defined as the confirmation, through the provision of objective evidence that the requirements for a specific intended use or application have been fulfilled. The validation process (for software) is the process to confirm that the requirements are correctly and completely implemented in the final product.

Software validation is often associated to software verification: these two separate processes aim at ensuring that the software manufacturer is building respectively the product right and the right product (i.e. compliant to its requirements).

Software testing is an important software engineering activity widely used to find faults in programs. Testing is the preferred validation method; however other techniques (such as static analysis, model-checking, design properties verification) may contribute to the validation objectives. Due to the growing complexity of space software, and as a consequence of the combinatory explosion of the product behaviour, validation by testing alone can become impossible or unfeasible.

In order to achieve an effective testing approach from functional standpoint, it is preferable that validation test cases are designed to cover a consistent set of requirements rather than a single requirement. On the opposite, several test cases may be needed to cover a single requirement in different context.

The validation activity demonstrates compliance with requirements. This may be done by showing the software performs as specified and contains no defect that prevents it performing as specified. Demonstration that a piece of software conforms to functional requirements may be achieved by exercising the function. However, demonstration that a piece of software contains no defects that prevent it from meeting its requirements requires expert knowledge of what the system does, and the

technology the system uses. This expertise is needed if non-functional requirements are to be met. Since exhaustive testing is usually not tractable on complex software, it is practically almost impossible to demonstrate that a piece of software is fault-free, unless huge resources are used to perform, for instance, formal proofs.

The validation activities are split over the one against RB and against TS. The chapter focuses on the one against RB, highlighting the specific aspects of this validation compared to the one against TS.

5.6.2 Validation process implementation

5.6.2.1 Establishment of a software validation process

5.6.2.1.1 Process

Software validation against TS and RB is made up of the 3 main steps:

a. Software validation tests specification,

b. Software validation procedures implementation,

c. Software validation tests execution and reports.

The software validation tests specification with respect to the TS and RB is written by the responsible of the software validation and is provided respectively at a dedicated TRR and at critical design review (CDR), see also 5.3.5. Due to supplier's environment limitations, it is possible that validation comprises a set of tests that check the software product against only a subset of the RB requirements.

For each test case, the software validation tests specification provides the identification of the test case, the aim of the test, and the list of TS or RB requirements covered by the test. It indicates the environmental needs: configuration of the facility, configuration of the support software (e.g. simulation) and special equipment needed (e.g. bus analyser). It lists the test inputs or describes its initial context, and gives the test steps and their associated expected outputs or success criteria.

For the critical design review (CDR) and qualification review (QR) the responsible of the software validation provides the software validation tests procedures, which describe how the test cases are implemented in one or several tests programs. They describe the treatments performed by these tests and how to launch them, drive their execution and take their results.

Even if the simplest layout, one test case per procedure, is often chosen, it is possible to have several tests cases implemented in a single procedure and a test case distributed among several procedures. Means or product configuration constraints can lead to such distributions.

The responsible of the software validation provides a software verification report (SVR) containing the synthesis of the validation test campaign and the reference to the analysis, if some requirements are not covered by testing.

The VCD (Verification Control Document) is a system document. It often uses the traceability between the SVS, the TS and the RB.

5.6.2.1.2 Additional considerations about validation test

NOTE Testing methods and techniques are also addressed in 6.4

The validation process takes into account the software development phase until the AR, including an incremental development scheme if any, and also the evolving / corrective maintenance activities during the system test with actual equipment and environment that:

a. May induce evolutions of the software,

b. May highlight remaining faults in the software,

c. May lead to the evolution of some mission data.

For all software modifications or evolutions after a first delivery, a generic non-regression strategy is set-up, which can be tuned on a case by case basis.

The validation tests do use intrusive tools to conduct the test and to observe its results. The (re-)execution of validation tests without instrumentation is mandated by the ECSS-Q-ST-80C Annex D tailoring of sub clause 6.2.3.8 only for software criticality categories A & B, but could be extended as a best practice to others categories.

When a requirement expresses a design constraint or a performance (e.g. TC buffer min size, max length for a command), additional tests are developed to cover the product behaviour at the boundaries or even beyond the limit:

a. Stress tests to verify the combinatory of the boundaries

b. Robustness tests to verify beyond the constraints and the error propagation

c. Performance tests to verify the SW behaviour in operational cases; this concerns mainly the CPU load, the long duration

5.6.2.1.3 Test environment

Introduction

The test environment addressed in this chapter concerns flight software and ground segment.

Flight software

In order to perform testing, there is a need for adequate validation support, dedicated to each validation phase, in terms of representativeness in particular w.r.t. the real execution environment but also in terms of capabilities. The name SVF (software validation facility) is typically used for calling the facilities for on board software validation. Sometimes, this SVF is incrementally developed and enriched across the validation phases in order to meet the validation objectives depicted here below.

Representativeness

Several levels of representativeness may be considered for flight software at:

a. processor level: from real processor module, processor emulators, up to host machine executing native code,

b. IO level: from full physical-electrical ISO layers to pure functional protocols,

c. equipment: from physical models, simulated ones, to restricted interface data buffering,

d. in-orbit environmental conditions: from real operational conditions with hardware in the loop [HIL, HWIL] to simulated equipment and subsystem.

The expected level of representativeness is increasing throughout the different test phases:

a. for unit and integration testing: in principle, at that stage, the representativeness is expected only at processor level. The execution of native code on a host machine may be sufficient, possibly complemented by processor emulators, depending on the expectations in terms of object code coverage.

b. for software validation testing: at that stage, the full representativeness is expected at processor and IO levels. For the equipment it may be restricted to interface data buffering without functional simulation. This representativeness at processor and IO levels may be refined depending on the two categories of validation tests:

1. For functional requirements testing: processor and IO emulators may be sufficient,
2. For timing and HW/SW interface testing: real physical processor and IO modules are expected. Those ones may be breadboards or EM which behaviour should not deviate from the FM.

c. for software qualification: full representativeness of processor and IO is expected, but real equipment is not yet needed and could be simulated at the adequate level.

d. for software acceptance and system testing: as far as possible, real HW is used in the loop, but however simulated equipment at the best level of representativeness for anomaly test cases. Environmental conditions should be simulated, approaching real operational conditions.

e. for in-flight acceptance: real spacecraft with real operational conditions.

f. for in-flight maintenance, the current trend is to use a Software Validation Facility (SVF), which is built around an on board computer numerical simulator.

For some small, simple and/or deeply embedded software (e. g. equipment unit or data processing), dedicated SVF may not be developed. It may be acceptable to perform software validation against RB directly on the system or equipment test environment during a dedicated phase or combined together with system level integration/qualification.

Capabilities

The required level of capabilities of the SVF depends tightly on the subsequent validation phase and its associated constraints:

a. debugging capabilities (breakpoints, patching and dumping memory addresses, setting/dumping IO registers) are expected in order to support investigations during low level testing (in particular during unit and integration testing but potentially also during validation testing when problems occur).

b. code execution tracing capability is expected to measure the structural coverage (for unit and integration testing).

c. higher level capabilities in order command and control at the TC/TM interface expected during validation tests. Therefore the provided tools should enable, thanks to a close connexion to the SDB to elaborate the subsequent telecommands, and retrieve the content of the telemetries.

d. timing measurements and real time behaviour observation capabilities are required, the less intrusive they can (e.g. LICE or JTAG real time probe), in order to observe and measure the software tasks execution (with a precise time stamping), granting a strictly same execution when the observation is not made (in particular on-board during operational life).

e. processor, memory, IOs, buses, equipment simulated range of values and failures capabilities (with the adequate timing accuracy) is required for all the test cases “at the limits” and anomaly test cases during all validation phases: it should be possible for example to tune the response time of an equipment on the communication bus within a specified range of values (minimum, maximum, typical).

f. checks automation and reproducibility are key capabilities for the SVF in order to be able to rerun test with the lower impact.

g. direct access to all memories (with adequate performance) is necessary in order to reload (respectively dump) all memories in particular to ensure an efficient flight software reloading.

h. context saving and restoring capability, in order to be able to bring the system under test into a pre-defined state.

Ground segment software

Two categories of ground software can be distinguished regarding software validation facility:

a. software interacting with the space segment (e.g. a Control Centre),

b. software interacting only with other ground software.

For the validation of software interacting with the space segment, a number of simulators and mock-up systems are typically used. A software-in-the-loop operational simulator is the most common facility adopted for the end-to-end validation of the overall ground segment. It typically simulates the space segment (including the S/G RF link) as well as the functionalities of some of the composing ground data systems, e.g. the ground stations network and the control centre to ground station communications (e.g. via the CCSDS Space Link Extension (SLE) services).

Several levels of representativeness may be considered for the simulator: the simulator models are enhanced gradually to expose more and more realistic interfaces (e.g. power, thermal, TM/TC and on-board communication interfaces). The highest level of representatively of the software-in-the-loop operational simulator incorporates an emulated version of the same OBSW as those deployed on the OBC of the space segment.

In the absence of a software-in-the-loop operational simulator, some mission may utilise a hardware model (engineering) model of a satellite combined with other dedicated simulators for validation of the ground data system. The possibilities of error injection and contingency simulations is however limited in such scenarios, which may result in partial validation of the ground systems.

For the validation of software interacting only with other ground software, some specific facilities are used to support the 3 kinds of activities involved in validation:

a. Context update (incl. provide expected input data),

b. Test execution (incl. use SW to SW communication means),

c. Result analysis (incl. compare with expected output data).

To support these activities, simulators are used:

a. Data Simulator, to generate expected data according Interface Control Document (for the syntax specification) and Software Validation Plan (for the various scenarios and associated contents),

b. Protocol Simulator, to simulate e.g. a Web Service provider or FTP-based communication tool.

Data Simulator and Protocol Simulator, combined together are able to replace the software that interacts with the to-be-validated software. This makes possible an early validation (when interacting subsystems are not yet available) or an independent validation (when different suppliers are involved in the development and maintenance of the ground segment).

Other tools are used to support the validation activity, in particular a supervisor, to run the tests according to an expected schedule, to monitor the execution and to store the results.

Test environment limitations

In the preliminary phase of the project particular attention is given to the definition of the validation test environment with the objective to be as much as possible representative of the real target HW, as well as to provide the necessary observability of the SW.

In case of test environment limitation, the validation is completed using other methods than testing as:

a. analysis,

b. estimations and structural study of the design,

c. inspection of parts of code that cannot be exercised in the configuration required by the requirements,

d. review of unitary tests.

Otherwise, clause 5.8.3.9 of ECSS-E-ST-40C is applicable. This clause specifies that the supplier has to identify the requirements that cannot be validated in the supplier's environment and forward them to the customer so that they can be validated in the customer's environment such as avionics, platform or satellite tests.

5.6.2.2 Selection of an ISVV organization

5.6.2.2.1 Introduction

The key technical objective of ISVV is to find faults and to increase the confidence in the software, which should therefore reduce the development risks. This objective can be reached by performing additional and complementary verification and validation of a software product (i.e. the software and all corresponding documentation) by an organisation that is independent from the software developer.

The "independence" notion of the ISVV is valuable and should permit to have a "fresh viewpoint" on the product and on the applied process. But the fact to have a different view does not imply to detect faults or defects if the activities and the method used are the same than the one used by the SW supplier within the nominal life cycle.

In order to be efficient, the ISVV activities should not start too early so that their inputs are mature enough. The experience shows that starting ISVV activities too early, results in a long loop and inefficient "questions / answers" process between the SW supplier and ISVV supplier via the customer (since there are no direct bridge between the SW supplier and the ISVV supplier, to grant the independence notion) that could result quickly in parallel activities (both nominal and ISVV) on the same items (most of discrepancies being raised in parallel by V&V and ISVV suppliers).

Currently, the SW V&V process is improved in order to be more and more integrated in the development process, as the earliest possible moment, in particular for verification activities (e.g. introduce verification tools inside the development environment to enable each developer to verify daily the quality of the produced code). This coupling between development and verification activities will induce more difficulties in order to schedule ISVV tasks in the future.

In the frame of ESA project, the ISVV process is supported by a dedicated ESA ISVV Guide [ESA ISVV Guide]. The following chapters are written in accordance and consistently with this ESA ISVV Guide to avoid duplication.

The ISVV should be requested and the detailed objectives defined by the customer. The ISVV guidelines should be therefore tailored according to the criteria defined in the guide such as the supplier knowledge/method level, or even customer knowledge/method level as example. In case of cost limitation, the activities should be tailored according to the expected benefits taking into account the level of risk the project is ready to take.

5.6.2.2.2 ISVV tailoring

a. ISVV levels

The ISVV guide [ESA ISVV Guide] introduces ISVV levels, in order to ease the tailoring process.

The following ISVV Levels are defined:

1. ISVVL 0: No ISVV activities are required.
2. ISVVL 1: Basic ISVV is required.

3. ISVVL 2: Full ISVV is required.

b. Factors

The ISVV Level is depending on two factors (ref ISVV guide annex E for detail):

1. the software criticality category
2. the error potential which depends on the characteristics of the development organisation, the development process or the software itself which may affect the quality of the software.

c. Activities

Annex D of the ISVV guide discusses important concepts of ISVV level definition. Each project tailors the applicability of the different tasks and activities to the different software products, components, following Annex D for the ISVV level definition.

This section introduces some tailoring which are based on project lessons learnt completing and clarifying the tailoring proposed by the ISVV guide.

Some emerging techniques and tools tend to be introduced in the nominal V&V process (ref. 5.6.2) by the SW supplier e.g. use formal methods, abstract interpretation or model engineering. Some of them overlap the ISVV activities that become potentially useless. The detailed impact on the ISVV process has not been reflected so far in the ESA ISVV guide. The ISVV effort could therefore be refocused on the following activities:

1. Analysis of the robustness w.r.t. to the external dependencies. The main faults, problems come from this area: lack of interface definition, lack of data ranging, lack of interface accuracy, lack of robustness rule w.r.t. these external interfaces
2. Verification of analysis report when some requirements are not covered by test.
3. Validation completeness: focus on the requirements that are partially covered by several tests and verify that the tests set actually cover the requirement.
4. Reliability: according to specific criteria to be defined (external defect propagation, verify what the system and the SW “should not do”.)
5. Focus on the User Manual. User Manual is an SW supplier output and it is a major document for the SW product operation, but it is not “validated” by test.
6. Focus on part of test specification for software that has been largely modified since the last baseline.
7. Focus on SW parts presenting deviations w.r.t. to the rules, metrics, others. In this case, the role of the ISVV process is not to verify the justification of the deviation but to define which kind of tests or analysis is needed to verify the potential weaknesses of the SW product.

d. Additional criteria

In addition to the ISVV levels driving the tailoring, some other criteria may be considered:

1. ensure that V&V and ISVV activities are complementary the one with the others
2. estimate ISVV activities added values with respect to the project context (reuse , autocoding) in order to establish priorities among the activities proposed by the ISVV guide
3. target some ISVV activities on the specific part of the product or based on the experience of found defects

4. take into account synchronisation constraints between the V&V supplier and ISVV supplier
5. define the adequate level of independence (separation of concern).

5.6.2.2.3 ISVV activities schedule

Scheduling the ISVV activity is key for its success. There is a need for proper synchronisation between the SW and ISVV supplier. Therefore the ISVV contract should not be managed as a single and continuous task but as dedicated and separated set of work packages triggered by completion of SW supplier activities (WP per WP once the inputs are stable and mature). This would enable to accommodate the schedule changes all along the project. This would also minimise the disturbance induced on the SW supplier and improve the ISVV efficiency.

This should not preclude providing the adequate level of visibility to the ISVV supplier on SW supplier up-to-date schedule in order to organise the ISVV activities accordingly.

In others words, the schedule of the ISVV activities should be enough flexible to support the followings examples:

a. unsuccessful review, delays or major pending issues , which could suspend the ISVV activities until the subsequent input is consolidated
b. incremental or complex function available in advance, which could request to anticipate some subsequent ISVV activities

5.6.2.2.4 Independent validation activities and related facilities

The independent validation activity aims at demonstrating that the major and critical requirements applicable to the SW are met in a consistent, complete, efficient and robust way.

By definition of the validation objectives, independent validation should be based on the RB / TS. It consists of defining and running dedicated validation tests or analyses. This activity requires a dedicated and representative SVF.

In order to be cost effective, the following alternative approaches can be applied:

a. The additional ISVV validation tests should be specified without knowledge of the SW supplier validation test specification.
b. The additional tests are based on a detailed analysis of the SW supplier validation test specifications, taking into account the suggestions proposed into the ISVV guide tailoring chapter.

To be noted in both cases, is the additional ISVV validation tests can be executed by the SW supplier team on their SVF. This would avoid procurement of dedicated SVF and training of the ISVV team in its use. The SW supplier should be contractually requested to ensure availability of their SVF and provide support to these activities.

The ISVV validation tests should be performed without modifying the SW binary image delivered by the SW supplier. The expected SW version should be a formal version delivered with a full configuration status and associated release documentation.

5.6.3 Validation activities with respect to the technical specification

5.6.3.1 Development and documentation of a software validation specification with respect to the technical specification

See also 5.6.4.1.

5.6.3.2 Conducting the validation with respect to the technical specification

-

5.6.3.3 Updating the software user manual

-

5.6.3.4 Conducting a critical design review

-

5.6.4 Validation activities with respect to the requirement baseline

5.6.4.1 Development and documentation of a software validation specification with respect to the requirement baseline

This chapter focus on software validation with regard to the RB, and provides recommendations on how to do it.

RB validation aims at validating the software against the requirements of the requirement baseline. The ECSS-E-ST-40C clause 5.6.4.1 requires the 100 % validation of the RB.

The requirements of validation w.r.t. the TS and the RB are the same, except:

a. the 5.6.4.1 / a / 1 requirement: “testing against the mission data and scenario specified by the customer in 5.2.3.1”.

b. the 5.6.4.1 / a / 4: validation w.r.t. RB requires a "non-intrusive environment", meaning that the interface used is the real operational one (e.g. TC/TM for on-board software) without use of any intrusive means to exercise software functions.

c. the 5.6.4.2 / b: validation w.r.t. RB shall test software as unmodified "black-box" (also without patches/workaround).

The RB validation test definition is an activity on its own. The RB requirements are written from the Customer standpoint and must be validated with this spirit, whereas the TS requirements are written by the Supplier and define the detailed behaviour of the software. As RB requirements are fully covered by (a set of) TS requirements that are themselves fully validated by a set of tests, this traceability may contribute to the identification of the validation tests against the RB. This requires that the traceability RB/TS is checked from an engineering standpoint (not being granted by a table or a traceability tool).

Validation activities w.r.t. to RB should complement the ones w.r.t. to TS in particular in the following cases:

a. when many TS requirements cover a single RB requirement, and each test case w.r.t. TS test individual TS requirements, thus not testing the parent RB requirement as a whole.

b. when operational scenarios have been defined in the RB (taking into account expert knowledge of what the system does, and the technology the system uses).

c. or when operational data is not available for the TS validation , and the set of operational data requires a rerun of a subset of the validation tests.

d. or when the TS validation test environment is not representative enough w.r.t. the qualification objectives (computer target representativeness or open/closed loop tests).

Except for the case (b) which should be clearly identified inside the RB, the two others cases should be specified inside the SOW as a validation logic scheme.

For case (b), the objective will be reached in exercising operational scenarios, elaborated jointly between the supplier and the customer with the aim of anticipating system level tests in the supplier's environment. The operational scenarios should be defined in the RB.

For case (d), ECSS-E-ST-E40C §5.8.3.9 specifies that the supplier is in charge to alert the customer. In this case, the activity will be performed on another test environment, either by the supplier or by another team (avionic integration team, system team, etc. …) under the responsibility of the customer. According to the alternative chosen, the validation w.r.t. to RB can be based on a subset of TS validation tests (same validation team using same test specifications , updating the program test according to the test mean configuration) or can be a complete new set of tests (new validation team introducing new test scenarios).

NOTE If the customer performs these tests, there could be a potential overlapping with the acceptance tests that is described in 5.7.3 software acceptance.

5.6.4.2 Conducting the validation with respect to the requirement baseline

-

5.6.4.3 Updating the software user manual

-

5.6.4.4 Conducting a qualification review

The qualification of the product is pronounced at the QR review, after the complete development and validation process has been conducted successfully by the supplier.

The qualification status is the evidence of a complete and compliant product to the requirement baseline (as per 5.6.4.1).

In case the supplier uses one single representative SVF (including closed loop) enabling to cover both RB and TS validation testing phases, and there is an adequate mapping of requirements between RB and TS (thanks to traceability), the supplier can conduct a combined phase test in order to validate the product simultaneously against TS and RB. In that context, it may be decided to have a single CDR/QR covering both objectives.

If the risk of finding errors late in the system testing is accepted, representative enough SVF may not be developed. It may be acceptable to perform software validation against RB directly on the real HW (BB, EM or even FM) during a dedicated phase, or combined together with system level integration / qualification. In that context, SW QR may be considered as part of the system QR.

Sometimes, the Customer defines himself the TS in form of detailed software requirements specifications SRS, which is then provided as part of the tendering material (ITT) as the base line for software development. In this case the software development starts directly from SRS, i.e. no RB exists. The supplier often amends and extends the SRS (TS) which is then reviewed by the Customer and finalised in SWRR. All verifications are then performed against this SRS (TS) and no RB is defined/maintained in this case. The QR is accordingly tailored out, with appropriate justification.

In the case of ground software developments, e.g. Mission Control System MCS and Operational Simulator developments, it is typical that the software is additionally validated by the software operator (Flight Control Team, FCT) in the context of so called Simulation Campaigns, using the Ground Operation Procedures and the Flight Operation Procedures. During these simulation campaigns all elements of the ground segment are either present as real software/hardware or simulated as part of the operational simulator, including the ground stations network and the constituting elements of each ground station as well as the communication between the Mission Control System and the ground stations. The Space to Ground link as well as the complete Space Segment are replaced (simulated) by the operational simulator, respecting the applicable space to ground ICD.

5.7 Software delivery and acceptance process

5.7.1 Overview

-

5.7.2 Software delivery and installation

The delivery is the action which consists in transferring any software version from the software supplier to a third party. This covers both formal versions subject to acceptance and intermediate versions which are not (section 5.7.3).

Contractually speaking, the delivery is traced by a delivery sheet, dated and signed by the customer and the software project manager. The delivery sheet identifies the software version number and refers to the corresponding software CIDL. In addition, when the delivery is performed on a media (tape, CD), this is identified.

The delivery can be performed even if the acceptance is not pronounced yet (key point not already held or unacceptable state at the acceptance key point), and the customer/end-user will then use the delivered software version at his risk. After formal acceptance, the software version can be used according to the delivered associated documentation.

At that stage, the product/software version is appropriately configured, and software configuration item list produced and reflecting the delivered version.

According to the agreed software development plan, the delivery may include all or part of the software documentation reflecting the delivered version.

The software development plan will address in detail:

a. The software versions to be delivered and accepted, with associated schedule.

b. The software delivery content: including media and associated documentation.

c. The potential meetings dedicated to the deliveries with associated level of formalism.

In the case of instruments or fully embedded software (firmware), formal software delivery will not occur solely but together with the hardware.

5.7.3 Software acceptance

5.7.3.1 Acceptance test planning

NOTE Testing methods and techniques are also addressed in 6.4

The software acceptance process is the review process of the software in order for the customer to check that it corresponds to his needs and to the contractual agreement.

Each time a software version is delivered, a formal and contractual acceptance is pronounced between the software project manager on one hand, and the "customer" or his delegate on the other hand. This concerns the case of incremental development. However, only the final version enters into a formal upper level validation process, at system level. For the delivery of intermediate versions, which do not enter formally into a higher level process (e.g. version delivered to AIT for preparation of their tests), the level of formalism may be adapted.

Especially in case of Agile software development, the frequent software deliveries (nightly builds and two weeks Scrum products) are not subject to formal validation/acceptance. The customer and the user of the delivered version may be either internal within the same company or external in the scope of a subcontract, e.g. AIT team within the satellite prime company. Nevertheless formal acceptance is pronounced on the latest version.

With an incremental process, if several software versions are issued, each of them is accepted by its customer. For example: TM/TC version, basic software version, AOCS. It may correspond to a certain content of the specification but also to different level of validation (partial, total).

Final acceptance triggers the software operations and software maintenance processes (as well as the guarantee).

When the software is embedded within an instrument, equipment, a piece of hardware, it is not necessarily accepted as a stand-alone product. In that case, the delivery/acceptance may be pronounced together with the hardware. The here-above described software acceptance process remains valid.

In case of the ground software, the acceptance process may be divided into more multiple steps. The software may be preliminary accepted (PA) conditional on fixing identified issues before it is finally accepted (FA) This incremental acceptance process is usually adopted in order to allow faster availability of the software in particular in cases that the delivered software is used for validation of other software projects, e.g. Operational Simulator is used for validation of the Mission Control System. In such cases the PA may still have minor faults but can be partially accepted in order to facilitate the progress on the overall ground software development. Also contractual and payment aspects are considered in adopting such and incremental acceptance.

For some software the operational environment in which the software will be deployed is geographically remote and cannot easily be used for acceptance purposes. This is for instance often the case for ground stations. In such cases the acceptance process may introduce intermediate steps such as Site Acceptance (acceptance on the reference environment at Customer site) and Operational Site Acceptance (final deployment and acceptance at the operational site, e.g. the ground station). The modalities and the process of acceptance is a contractual aspect that is specified and agreed as part of the contract between the Customer and the Supplier.

In the particular case, the customer and supplier belong to the same organisation, the acceptance tests may not exist as a dedicated standalone test phase:

a. acceptance tests may be part of system level testing,

b. software acceptance review may not be conducted separately but together with system level acceptance review.

5.7.3.2 Acceptance test execution

Formally acceptance tests can start after dedicated TRR where the acceptance test plan is approved.

The technical interest of the customer is to cover 100 % of the RB during acceptance.

The main difference of the acceptance compared to qualification tests resides in the representatively of the scenarios, and the execution environment that are the customer ones.

The acceptance test plan describes the scenario to be exercised (in which order and according to which criteria the tests cases are run), including breakpoints in the case an anomaly occurs (acceptance abortion or continuation allowed).

As a consequence, two situations are considered:

a. The acceptance is effectively pronounced when the customer has run all the tests contributing to the full validation of the RB (e.g. when the customer and the SW supplier are belonging to the same company). In that context, some optimisation may exist in order to combine/perform jointly validation against RB by the SW supplier, acceptance testing, and further more system testing.

b. In the opposite, when the SW is sub-contracted to an external entity, and in the same spirit as it is performed for HW, an acceptance is performed on a "qualitative" subset of RB (considered adequate by the customer) in order to be pronounced as early as possible after the end of the supplier's activities.

The tests may cover the rebuild of the executable from source code, to ensure that the build process can be performed.

As a general scheme, two categories of acceptance tests are considered:

c. Supplier site acceptance tests: these tests are performed by the customer (or at least witnessed by the customer), at the supplier's premises, on their environment. They comprise a set of tests that globally check the software product against its requirements baseline. Usually, these are a subset of tests contributing to the qualification of the product or anticipation of tests to be performed during final site acceptance tests (ref. hereunder).

d. Final site acceptance tests: these tests are performed by the customer, at the customer's premises, on the target environment. They comprise a set of tests that globally check the software product against its requirements baseline.

5.7.3.3 Executable code generation and installation

The software delivery and acceptance process of ECSS-E-ST-40C leads to the acceptance of the software based on a subset of the requirements baseline validation tests. Once the software is (re-) generated, the next step of the acceptance is to install the software according to the dedicated procedure.

5.7.3.4 Supplier's support to customer's acceptance

The supplier supports the acceptance testing activity. The main responsibilities of the supplier during the acceptance phase are:

a. conduct investigations whenever necessary on problems/anomalies occurrence (assess the "normal" and "abnormal" observed behaviours),

b. train/teach/transfer knowledge of the SW up to the customer level (therefore the support is directly an active participation),

c. update the user's manual whenever necessary in order to reflect all the constraints, "features" and associated behaviour induced by the SW design choices observed during acceptance tests on the run scenarios,

d. Depending on the contractual agreement, the supplier may run the acceptance tests of behalf of the customer.

Execution of acceptance tests are logged /archived. A logbook is established in real time during the execution of tests, based on the acceptance test plan, identifying the status OK/NOK at each step of the procedure.

5.7.3.5 Evaluation of acceptance testing

The supplier is expected to review and comment on the acceptance test plan, in order to identify potential inconsistencies of the test scenarios w.r.t. the software product capabilities.

5.7.3.6 Conducting an acceptance review

For each version, the acceptance is pronounced during a review (which could be very light in terms of formalism and combined with the delivery). The acceptance data package is defined in the acceptance review procedure.

The results of the tests are reported at the acceptance review, which is convened by the customer.

Formally acceptance tests are declared finished after the acceptance review where the list of anomalies is analysed by all the parties:

a. blocking anomalies: preventing the success of the acceptance and requesting a delta / full acceptance again,

b. minor anomalies: the acceptance may be then declared successful given a certain number of actions/corrections to be performed by the supplier (leading to a new delivery if needed).

For both type of anomalies, workarounds and patches may be delivered by the supplier.

During the acceptance review, the agreement of the two parties will be traced on:

c. The exact content of the inputs/applicable documents together with potential change requests, and also waivers if any,

d. The agreed/applied process, in particular if all the life cycle has not been applied (missing tests), including the list of reviews held if any (and reference to the corresponding minutes of meeting),

e. The tests results and list of identified anomalies including potential impacts/workarounds if any,

f. User's manual/installation procedure, where relevant.

At AR, the supplier proposes and commits on a detailed schedule for the open actions.

5.8 Software verification process

5.8.1 Overview

As defined by the ECSS-E-ST-40, Verification is the confirmation, through the provision of objective evidence, that specified requirements have been fulfilled. The verification process (for software) is the process to confirm that adequate specifications and inputs exist for any activity, and that the outputs of the activities are correct and consistent with the specifications and inputs.

If the validation process aims at verifying that "you are building the right product", the verification process allows checking that the "product is right" which also contributes providing evidence that the "right product" has been developed.

The Software Verification Plan (SVerP) describes the approach, defines all the verification activities of all the expected outputs of the development activities, and the organizational aspects to implement the software verification activities, i.e. focusing on the organization, schedule, resources, responsibilities, methods and tools. The ECSS-E-ST-40C Annex I provides a SVerP template. Although the SVerP is delivered for the Preliminary Design Review (PDR), the Verification process should often be set-up before the PDR, i.e. during the requirements analysis phase.

Verifications may be performed through different means (e.g. reviews, inspections, analysis, simulations, prototyping, audits, specific tooling).

Activities developed hereafter (not exhaustive with regard to the complete list of verification activities required by the standard) consolidate some topics of the standard or highlight some activities, which require a specific attention. They are generally considered as best practices to consolidate earliest the path to a "right product".

5.8.2 Verification process implementation

5.8.2.1 Establishment of the software verification process

5.8.2.1.1 Overview

The ECSS-E-ST-40C requirement 5.8.2.1.c asks for the methods and tools to support verification activities. This section addresses (i) the traceability, which is common to several verification activities, and which is supported by tools, and (ii) requirement engineering, which is common to RB and TS elaboration.

5.8.2.1.2 Traceability

Introduction

The ultimate aim of traceability is to support or facilitate some activities such as:

a. verification, e.g. completeness, consistency,

b. change analysis, i.e. the so called impact analysis,

at a fine level of the software product.

In this aim, traceability activity consists in establishing and maintaining the necessary links between detailed elements of the software product (e.g. a requirement, a test case) that allow "navigation" between these detailed elements. Links may be of different types (e.g. implementation, refinement, verification, reference) according to the traceability needs.

However, due to the quick increase of the traceability complexity and effort, the basic traceability need consists in establishing links allowing checking that the upper level requirements are taken into account (or justified) by the lower level software elements throughout the life-cycle.

This is the approach of the ECSS-E-ST-40C which identifies the following detailed elements:

a. system requirements of the RB,

b. software requirements of the TS,

c. software components of the SDD,

d. software units of the SDD,

e. software code,

f. test/analysis/inspection cases of the SUITP, SVS-TS and SVS-RB.

The ECSS-E-ST-40C requires the following coverage verification (See clause 5.8.3):

a. between system requirements and software requirements,

b. between the software requirements and the software components,

c. between the software components and the software units,

d. between the software code and the software units,

e. between the tests cases (TS and RB) and the software requirements, and the system requirements.

Moreover, ECSS-E-ST-40C also requires an overall traceability:

a. code traceable to design and requirements,

b. unit tests traceable to code, design and requirements,

c. integration test traceable to architectural design.

Direct links between all detailed elements are hopefully not necessary and some techniques allow minimising the traceability effort:

a. Establish a direct link between two types of detailed elements only when the coverage measurement is required: e.g. between system requirement and software requirement. Such links are usually established between detailed elements of "adjacent" steps of the life-cycle (e.g. requirements analysis and architecture). For others, transitivity mean is used: for example, if an element A is traced to B, and B to C, transitivity can be used to assess the traceability between A and C. Complete the direct links accordingly,

b. Moreover, if most of the time explicit links (i.e. manually established and often explicitly formalised within documents or tables) are necessary, the use of implicit links (i.e. through clear criteria or rules) is recommended as much as possible for efficiency reasons. Naming rules (e.g. a software unit has the same name than the related ADD component) are commonly used for implicit traceability.

Generally, for the ECSS-E-ST-40C, direct links are established for each couple of elements here after (most corresponding to the ECSS coverage needs):

a. between system requirements and software requirements,

b. between the software requirements and the software components (could be implicit when software requirements are made together with a generic architecture, the reference of the requirement may include the component name),

c. between the software components and the software units (could be implicit through dotted naming rules),

d. between the software code and the software units (usually implicit through naming rules),

e. between unit tests and the software units (usually implicit through naming rules),

f. between the integration tests and the software components (usually implicit through naming rules),

g. between the TS tests cases and the software requirements,

h. between the RB tests cases and the system requirements.

Other ECSS-E-ST-40C traceability needs do not require any additional effort since they can be obtained by transitivity.

Once the traceability is established, traceability links are assessed in both directions. For completeness verification, the covered and un-covered elements are identified and appropriate actions are taken to improve the coverage (i.e. modification of the traceability, modification of the elements, or justification). The most efficient way is to use a traceability tool. A matrix may also be used to support this verification.

When coverage proofs are requested, the most popular way is to provide a traceability matrix showing the elements in relation (next upper level elements on the left part of the table) and the potential justifications for lacks of coverage. Bi-directional matrixes are generally useless.

The effort on justification on completeness and granularity of tracing could be decreased according to the software criticality level, e.g.:

a. for C-level software criticality, justification could be limited between RB and TS requirements, TS requirements and test case,

b. for D-level software criticality, justifications could be skipped.

Traceability using Tools

Through numerous other features, Requirement Management tools usually also allow requirement traceability through the creation and maintenance of explicit links from and/or to requirements (or other traceable elements). This allows to fully managing the requirements, from their formalization, their configuration management up to their traceability.

When requirements are formalized through editing tools, a Requirement Traceability tool may be used instead of a Requirement Management tool strictly dedicated to the traceability management (e.g. coverage analysis, matrixes reporting). Such tools (based on pattern matching principles) require a clear and systematic formalisation of the traceability elements and links within documents.

When model based engineering is adopted, integrated use of requirement management and modelling tools should enable to extend the set of explicit links or relationships from the requirements to external entities of the logical model, the design or the validation specification.

Modelling of such external links has the following advantages for software development process:

a. Links from the requirements to the logical model enable to better verify that the technical specification is complete (i.e. by checking a software requirement exists for each logical model element).

b. Links enable the verification of design completeness: no requirements are discarded, no components are non-justified.

c. Links enable the verification of test completeness: each requirement has at least one test case associated, and no test case is redundant. The links help determining the regression test.

As an extensive example, links supported by SysML language are provided in Annex B.

5.8.2.1.3 Guidelines for requirements definition

The following list enumerates desired characteristics of the requirements:

a. The RB set expresses what is needed by the stakeholders (users, customers, suppliers, regulating bodies) and should avoid stating "how" to achieve this. Requirements that constrain the possible solution space may be included in the RB.

b. Each RB requirement is attainable, which means it is achievable within reasonable cost, time and technology. Of course this aspect is subject to negotiation and adequate risk management.

c. The TS set should be grouped according to the different categorization. The ECSS-ST-E-40C suggests a list of categories, such as functional, and performance, reliability, operational, etc.

d. Each requirement is expressed in clear language, and in case a clear notation, so that it can be readily understood and reviewed by the customer. Recommendations on the wording of requirements, as well as other requirements for formulating technical requirements, are contained in the ECSS-E-ST-10-06C, the standard for technical requirements specification.

e. When the number of functional requirements gets large the set should be properly decomposed into subsets that contain requirements “of the same kind”. The grouping may be done according to different categorizations (e.g. functional decomposition).

f. Requirement grouping can be done at different levels by establishing hierarchical relationship between (sets of) requirements: from higher level, more general requirements to lower level, more detailed requirements.

g. When necessary (e.g. in case of system constraints, performance,) requirements are accompanied by a proper rationale or justification.

h. Each requirement is unambiguous, explicit, verifiable and traced.

i. The set is complete and consistent.

j. For each requirement one or more verification methods are identified and substantiated.

k. Complete traceability between a requirement set and the upper level requirements is maintained.

l. The revision history of each requirement is kept and made accessible.

5.8.2.2 Selection of the organization responsible for conducting the verification

-

5.8.3 Verification activities

5.8.3.1 Verification of requirements baseline

Unclear responsibilities assignment between system and software teams may result into software failures. In particular the interpretation of the “system needs” by the software teams is the cornerstone of the system and software processes.

A key principle of the ECSS-E-ST-40C to achieve as far as possible the common understanding of the software requirements, is to build the requirements in two steps (RB for needs then TS for solution)

and to perform a cross-check through two separate requirements documents (i.e. RB and TS) and reviews (i.e. SRR to check the needs completeness, and the PDR (potentially anticipated by the SWRR) to allow the Customer checking that the proposed software solution answers the needs and the software supplier checking that his understanding of the customer needs are correct, complete and results into a feasible software solution.

This handbook also recommends an early integration of the software engineering process in the system engineering phase (co-engineering) to facilitate the common understanding of the requirements: the supplier can early understand the software needs and checks their feasibility, before the SRR.

In the situation where the SW supplier is not yet selected when the SRR is conducted, the SVerP may be only available at the PDR, consequently after the SRR. This would prevent co-engineering process to be set up and would therefore delay the verification activities to be performed at System-Software engineering level.

In order to verify the completeness of the Requirements Baseline (RB), the following list provides a set of topics which could strongly affect the software during its development process or which are shared by different elements (e.g. interface, failure mechanism, and timing performance). These topics need to be checked carefully at system level (for completeness) and at software level (for feasibility):

a. the RB describes clearly the environment in which the software will operate. The description focuses on the operational context and the interfaces with external systems,

b. the RB specifies the characteristics of all external systems (e.g. bus, computer, ground interface) interacting with the software product. The above-mentioned characteristics include, but are not limited to, the communication protocol, the concurrency and real-time model,

c. the RB specifies the control and observation points for each function,

d. the RB specifies the fault detection, isolation, and recovery , in relation with the dependability and safety analysis outputs,

e. the RB specifies the modes/sub modes and transition between modes (modes automaton). For each mode/sub mode and transition, the associated activities (functional description) are specified,

f. the RB specifies how telemetries / packets TM are dated and the precision required,

g. the RB identifies the requested configurable data of the software, that are usually defined through a software database,

h. the RB identifies and justifies the margins policy in terms of memory and CPU allocation,

i. the RB defines the operational scenarios (e.g. traffic scenarios),

j. the RB specifies the list of external events that are produced or used for execution by the software product, as well as their occurrence model (e.g. periodic, spontaneous),

k. the RB specifies, for each application function or hardware component, the type of failure (no failure, accidental failure on value or time, Byzantine or not, intentional or not [security]) and their occurrence model (e.g. periodic, a periodic),

l. the RB specifies the timeliness properties (i.e. constraints on time to start or terminate application functions), periodicity, and jitter when they are mandatory with their associated justification (e.g. AOCS commanding loop between sensors acquisitions and actuations),

m. the upper level project development plan (system, subsystem, equipment, instrument, platform development plan) specifies the schedule allocation for the software development, including margins consistent with respect to the development effort,

n. the upper level project development plan defines all inputs necessary for the development activities, such as the test facilities, input documentation, CFI (Customer Furnished Items), environmental models if necessary and their availability,

o. the upper level project development plan identifies all needs in term of software product deliveries and their dates.

5.8.3.2 Verification of the technical specification

The "design feasibility" verification, required by the ECSS-E-ST-40C requirement 5.8.3.2, may be anticipated by activities such as:

a. when the requirements engineering is based on a model checking approach (Annex B), some properties of the model may be verified at the earliest stages of the life cycle, and from which the design model may be derived afterwards,

b. a strong system software co-engineering primarily established to consolidate the needs, also allows an early consolidation of the design feasibility when supported by fine SW experts analysis or prototyping,

c. the need for an evolutionary robust design approved by a peer review in particular if an iterative life cycle is selected (See the section 5.3.1on life cycles that recalls the risks of each life cycle models),

d. the static and dynamic design are well defined and supported by a monitoring of the resources budget,

e. the reused generic architecture or building blocks are able to support the specific requirements.

5.8.3.3 Verification of the software architectural design

The ECSS-E-ST-40C requirement 5.8.3.3 requires developing "feasible operations and maintenance ". In fact these two notions can be anticipated individually.

The operation feasibility can be covered through prior verification activities as:

a. the environment (the external interfaces definition completeness) in which the software operates is clearly described in the RB and correctly refines in the TS,

b. the controllability and observability definition of each function describes respectively the telecommand services and telemetry services used,

c. the fault detection, isolation, and recovery strategy is coherent,

d. the operational scenarios are defined in the RB as use cases.

The future maintenance can be assured through prior verification activities on the following points:

a. up to date complete QR/AR data package,

b. a static and dynamic design sufficiently documented,

c. the Software Development Environment and Software Validation Facility are, as much as possible, decoupled from potential complex infrastructures existing in the companies,

d. availability under adequate configuration management control of all non-deliverables contributing to the software product e.g. models, source code, makefile, build procedures, test scripts,

e. the non-obsolescence of development, validation facility and production chains (potential obsolescence to be mitigated by potential migration),

f. training of skilled people e.g. involvement of maintenance team before the end of the software development activities,

g. the completeness of the maintenance plan

5.8.3.4 Verification of software detailed design

-

5.8.3.5 Verification of code

The purpose of code coverage measurement is to measure the confidence one may have in a software product and to assess the risk that potential faults remain hidden in part of the software which has never been run.

ECSS-Q-ST-80C 6.3.5.2 and ECSS-E-ST-40C 5.8.3.5.c requires code coverage to be measured.

For Category A and B there is an requirement [ECSS-E-ST-40C 5.8.3.5.b] for 100 % code coverage (100% MCDC only for Cat A). For Category C and D, the standard requires that the value is agreed with the customer.

ECSS-Q-ST-80C requires a quality model. One of the metrics of the software quality model can be the targeted number for code coverage for category C and D which can be any number according to the project needs and risk analysis.

The effort of performing code coverage and reaching the targeted number has to be considered in a context of risk analysis and reuse process: The **risk** is evaluated according to (i) the **probability** of a fault in the software and (ii) **consequences** in the system (i.e. criticality). The probability of a fault in code that has not been tested is the same for flight software and for ground software. The consequences are the same if they have the same criticality level. The risk is therefore the same for flight and for ground software. It is sometimes considered more acceptable to take higher risk on ground software than on flight software, because recovery actions are perceived easier on ground systems.

The complexity of a module may directly affect the risk of remaining faults. Therefore, complex units may deserve a higher code coverage target supported by proper unit testing.

In addition, traditional ground software usually relies on significant reuse of (reaching sometimes more than 90 %):

1. COTS and operating system for which the source code is usually not available,
2. reuse of open source or proprietary software for which the confidence has been proven,
3. firmware of hardware components such as firewall, routers.

For this part of the software, code coverage either cannot be conducted or requires a substantial effort.

In addition, the amount of unused code which is not relevant, but simply deployed as part of the reused components, can be significantly high.

More generally, the suitability of reusing the software in the project has to be evaluated in accordance with ECSS-Q-ST-80C 6.2.7.

5.8.3.6 Verification of software unit testing (plan and results)

-

5.8.3.7 Verification of software integration

-

5.8.3.8 Verification of software validation with respect to the technical specification and the requirements baseline

-

5.8.3.9 Evaluation of validation: complementary system level validation

-

5.8.3.10 Verification of software documentation

-

5.8.3.11 Schedulability analysis for real-time software

Schedulability analysis is discussed in 7.5.

5.8.3.12 Technical budget management

See also 7.3.

5.8.3.13 Behaviour modelling verification

NOTE Model based engineering is discussed in 6.3.

The ECSS-E-ST-40C requirement 5.8.3.13 requires the verification of the behavioural model. The following topics should be verified to anticipate the correct behaviour of the final SW product, during different steps of the life cycle:

a. the software dynamic aspects are well defined at design level: each function is properly described (e.g. with finite state machines or Statecharts), the way the different services are used (e.g. frequencies, task context, call order / dependencies, external activations by interrupts…). Sequence diagrams / time lines is an efficiency way to show the dynamic behaviour of the services and the interactions between them;

b. the proposed design provides SW behavioural observability and debug features (e.g. time stamping of tasks starting and ending, semaphores changing);

c. the software design is safe and robust, in particular with regard to the external error propagation (e.g. defensive design, implementation analysis, test coverage).

5.9 Software operation process

5.9.1 Overview

5.9.1.1 Introduction

The purpose of this section is to explain the process related to the operation of space software. The following types of software are covered:

a. Flight software embedded in non-accessible space objects

b. Flight software embedded in serviceable vehicles which could be accessed at least in a limited way, through a human machine interface

c. Ground software which can be accessed at any time

All 3 categories of software are operated by the entity combining operations organizations and associated ground systems for which ECSS-E-ST-70C is applicable. For bullet a) flight software the software operation process described in ECSS-E-ST-40C (chapter 5.9) is to be considered in a wider scope of space vehicle operation.

The software operation process is linked together with the software maintenance process.

It covers the following activities (as far as applicable for a space project):

a. Operational management encompasses activities like:

   1. managing and coordination of Ground Segment service requirements,
   2. provision and support for all mission activities,
   3. coordination of Missions operations management and Engineering,
   4. management guidance,
   5. conflicts resolution, and
   6. on console during critical phases.

b. Operational planning involving producing the procedures for operating the product, training operators and users, operational testing, problem reporting, system operation and user support. This encompasses tasks like

   1. planning of Ground Segment resources for all preparation activities and mission activities (also with other projects and missions where applicable) with respect to Ground Segment usage,
   2. technical and operational inputs for Simulation and Mission Timeline,
   3. review of Mission Requirements Documentation,
   4. resources scheduling / conflict resolution for all activities,
   5. resource usage tracking,
   6. Ground Communications Schedule generation,
   7. IGS Work Plan generation,
   8. Coordination of all participating facilities for On-board System and PL operation,
   9. Preparation of Ground Segment operations (including simulation and testing activities), and
   10. Voice Loop Management.

c. Operational testing of new releases: the software product is released for operational use when the operational testing criteria have been satisfied, in accordance with the operational plan.

   NOTE The team responsible for software maintenance process is not always in a position to carry out operational testing, since this cannot be done without the operational environment and possibly specific operations expertise.

d. User support, implemented in practice using the organisation set up for the software maintenance process, including

   1. assistance and consultation to the users as requested,
   2. provision of workaround solutions,

3. advise on software use and documentation, and
4. handling user requests for software maintenance, including recording problem reports.

These activities are undertaken by the "System Maintainer" and the "System Operator" (administration part of the role): these two roles are attributed to a SOS entity in the standard. The SOS entity is a link between the system operator and the software maintainer team. The SOS entity assumes a coordinator role during operations.

The phasing and management of the operations process is determined by the system level requirement to operate the software product at a given time, i.e. it is not always connected to overall project Phase E. Operations engineering can start earlier or later than the phase E. In principle, operations engineering starts at the time of the operational readiness review (ORR, see ECSS-E-ST-70C). In practice, it starts earlier, because ground segment software products are in extensive use to qualify the ground segment and interfaces to the space segment well before operations occur. On the other hand some software may not be operational until long after launch, for example in a deep-space mission.

5.9.1.2 Incident Management

During operation of the software, incidents can occur. An incident is an unplanned interruption to a software system or service, or a reduction in quality of a system or service. Failure of a configuration item that has not yet impacted a service is also classed as an incident.

The purpose of incident management is to restore a normal service as quickly as possible. Usually an incident is resolved by applying a workaround to the system and a new release of software would not be applied at this point since developing a new release would be a much slower activity.

Incidents would normally be channelled through a single system wide Service Desk; however there may be a maintenance organisation (such as the system prime) that has a single point of contact through which to channel incidents. The relevant tracking tools are required in order to track incidents from when they are raised to their closure and agreement of this closure with the originator.

The design phase is carried out using the same tools and processes as were used to build the software in the first place. In maintenance the software will be changed in order to resolve problems, or to implement changes or improvements.

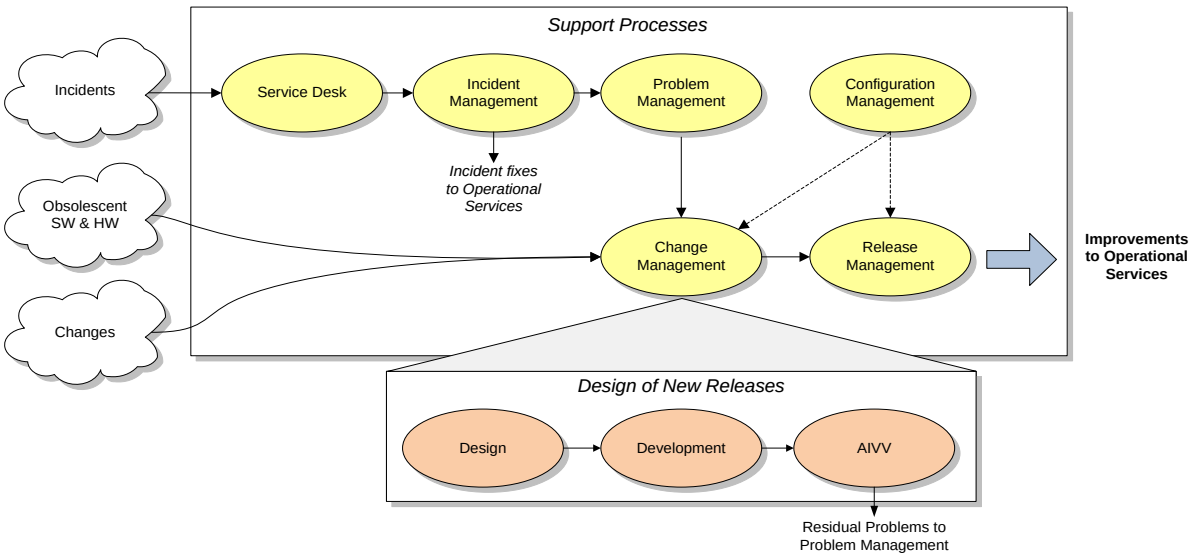


Figure 5-7 : Example ITIL processes

5.9.1.3 Problem Management

A problem is a cause of one or more incidents. The cause is not usually known at the time a problem is created. The problem management process is responsible for further investigation and the recommendation of the fix that will resolve the root cause.

One or more problem fixes may drive the production on a new release of software. How problems are grouped into releases will depend on the priority and severity of the problem.

Decisions not to implement some functions and not to fix some defects in flight software are often made on the basis of whether or not an 'operational workaround' exists. This amounts to moving complexity from flight software to mission operations where it is a continuing cost rather than a one-time cost and where it increases the risk of operational error, especially as the numbers of such workarounds accumulate. Operators need to be consulted early and a considered trade-off analysis performed taking into account the costs and risks. Only then should flight software be de-scoped by retaining long term workarounds.

5.9.1.4 Release Management

Release management is the process used in plan releases from determining their content to putting them through transition into operations.

Release management will determine when releases are required in order to meet the customers or system's needs. It is usual to conduct a review meeting between e.g. the operators of the system, the owner, the maintainer to agree the changes (including problem fixes) that are grouped into releases and when these releases will be rolled out.

The release management process will ensure that the release is on target to meet it roll out date. A test review meeting will be held to make sure that the results of the testing has been agreed and the release is ready to be used. A release Board will then be held. This discusses the suitability of the release and when it can be rolled out onto the system. The Release Board is usually attended by representatives of the operators, maintainers, integrators, safety engineers, trainers. Its purpose is to make sure that all parties have made preparations for the release, the release is ready to be rolled out, and the operators are trained up and ready for its deployment.

5.9.2 Process implementation

-

5.9.3 Operational testing

-

5.9.4 Software operation support

-

5.9.5 User support

-

5.10 Software maintenance process

5.10.1 Overview

-

5.10.2 Process implementation

5.10.2.1 Establishment of the software maintenance process

5.10.2.1.1 Introduction

The software maintenance process is activated when the software product undergoes any modification to code or associated documentation as a result of correcting an error, a problem or implementing an improvement or adaptation. The objective is to modify an existing software product while preserving its integrity. This process includes the migration and ends with the retirement of the software product or of the system itself.

Its starting point is dependant from the contractual set-up of the project (e.g. warranty period of the supplier, which organization is later-on in charge of performing the maintenance), but in most cases the start is coupled with a project review (like QR, AR, or FAR).

This section provides guidance in particular about:

a. How the maintenance is usually structured and how it relates to the other phases;

b. Best practice processes for managing software maintenance;

c. The different types of maintenance activities and how these can be handled;

d. The Software Maintenance Plan.

5.10.2.1.2 Maintenance Support Processes

Change Management Process

Change management ensures that changes to the system are managed to ensure that their impact on the design is fully understood, and they can be incorporated in a controlled manner. Changes may come from two sources. They may be internally generated as a result of a problem resolution, or externally generated e.g. as an improvement to the system. Ultimately both these types of change need to be tracked in the same way although they may have different contractual implications.

Configuration Management Process

Software Configuration Management is the process of ensuring that the software is complete and forms a set of consistent elements during its development phase. The purpose of Configuration Management during maintenance, is to identify control and account for service assets and configuration items (CIs) to ensure their integrity across the maintenance lifecycle as well. This will usually require the use of a supporting Configuration Management System in order to ensure that the assets and CIs are controlled correctly.

Effective Configuration Management is an essential part of the maintenance process, and needs to be maintained in order to ensure that every asset in the test and development systems, is held at a known state with a known version of the software build.

5.10.2.1.3 Maintenance Categories

The overall software maintenance concept covers all types of maintenance, including the handling of modifications that are initiated by a change request. The following sections identify the different categories of maintenance, and the type of activities that are performed within each:

Planned Operational Maintenance

In this maintenance category special software updates are planned and scheduled in order to prepare the software system for the next mission increment, especially for operations. As such the operational maintenance is pre-planned and driven by the user and mission increment needs. It is not related to anomalies.

The following activities are typically related to operational maintenance:

a. Modification of operational products as requested for the mission increment;

b. Adjustment of other flight software products;

c. As side effect of the mission increment specifics (operational products), as far as they are not of the type corrective maintenance;

d. Adjustments to improve the mission operation;

e. Regression tests appropriate for the mission increment;

f. Preparation of software system deliveries;

g. Documentation updates.

Flight software changes required for operational maintenance are:

a. Adjustment of flight data (non-operational products) , if no flight software is impacted ;

b. Adjustment of error messages to provide more information or avoid misunderstandings during mission operation;

c. Adjustments of flight software internal buffers to cope with mission increment specifics;

d. Adjustment or re-definition of On-Board Application Programs (OBAPs) in order to improve mission operation or to cope with hardware problems;

e. Adjustment or re-definition of Simulation Models in order to improve mission operation or align the models with the real system (Hardware and Software).

Unplanned Corrective Maintenance

All unscheduled activities are categorised as Corrective Maintenance. This is the case for all faults and failures that causes a space software system to be inoperable. The objective is to restore the space software system to operational conditions as required. Corrective maintenance might lead to instant patches, or corrections of the problem in later mission increments. This is the case for all faults and failures that causes a facility to be inoperable. The objective is to restore the infrastructure to operational conditions.

Corrective maintenance will be performed by a field service when hardware and software problems are detected and caused by a malfunction of an infrastructure item (depending on an agreement of spares concept).

Therefore following activities are part of the corrective maintenance:

a. Removal of faults and failures;

b. Repair and replacement of faulty hardware items;

c. Repair if feasible and not time consuming at the facility (e.g. for cables and connectors);

d. Establishment of temporary workarounds if spares are not available in time or not cost effective;

e. Troubleshooting to isolate and detect software failures;

f. Software patches;

g. Regression tests;

h. Documentation updates;

i. Problem handling to describe the complete problem process over the problem life cycle.

The corrective maintenance activities are supported by:

a. local and centralized spare stocks;

b. contracts with local vendors for maintenance.

Once a fault or a SW fault has been identified the correction process is initiated according to agreed project needs. The problem will be corrected either by installation of a SW patch or the HW will be repaired or replaced. This includes the ability to correct faulty processes or threads by replacement using other memory space when necessary.

After installation of an update regression tests need to be executed on the reference facilities prior to installing the new item at target site. If necessary at the target site selected regression tests will be performed and results reported to ensure that the installation was successful.

In case of a hardware failure it is investigated if the hardware item can be repaired or has to be replaced. In case of a hardware replacement the procurement or repair needs to be initiated.

Preventive Maintenance

Preventive maintenance means planned and scheduled activities that are performed to keep the Infrastructure in operational conditions. It is performed to ensure the long term availability of the facilities hardware and software to support the day by day operation. Typical tasks are:

a. SW Back-Ups;

b. Periodic inspections to monitor the condition of the equipment;

c. Inspection of the HW with the emphasis on the detection of faulty mechanical / electrical parts;

d. Calibration of equipment;

e. Replacement of consumable items (toner, paper, light bulbs etc.);

f. General cleaning and maintenance;

g. Maintenance of the logs and error statistics.

Adaptive Maintenance

Adaptive maintenance means that the software can be upgraded in such a manner that they get over a maintenance phase without changing the specified hardware or software functionality. Commercial

software licenses need to be available for the complete duration of the maintenance phase, i.e. planning has to take into account that during a maintenance phase no sudden hardware and/or commercial software upgrades are necessary due to obsolescence of the items.

Adaptive maintenance might also be induced by hardware obsolescence. As the operations facilities are usually in operation for several years and most of these facilities or parts of them are used day by day without major technology upgrades since development ended. On the other hand, technology is moving fast in the commercial markets in particular in the computer, communications and video area, so that it is becoming very difficult, time consuming and costly to maintain outdated equipment.

As a countermeasure the Ground Segment service introduces an obsolescence management to have a controlled process to replace equipment in due time before it becomes outdated and thereby avoid future additional unforeseen cost. In the past replacement of equipment and systems was mostly driven by external event, i.e. future operations cost, no more commercial maintenance available, etc. In an operational environment with an expected lifetime reaching possibly beyond the current decade, the replacement of ground equipment is planned and controlled to keep the equipment and systems operational in a cost effective manner.

Continuous monitoring of the existing equipment and review of the commercial technology developments and trends as part of the ground segment engineering is one input to the obsolescence management. Another important contribution is the planned yearly configuration review. Out of both information sources a replacement strategy is established and maintained for the individual systems and equipment. The overall obsolescence strategy will be to replace equipment as needed dependent on technology improvements and commercial developments but as seldom as possible before the equipment becomes obsolete and causes negative impact to operations.

Evolutionary Maintenance

Evolutionary maintenance is initiated by external demands to improve the functionality or system reaction. It will be initiated by end user problem reporting, but only implemented on the basis of a customer change request or it is covered in the business agreement as explicit system enhancement.

The objective is to prevent failures and optimize the system for the operational use. This might be done, for example, redesign a component that has a high problem rate, or modify a component in order to improve its operability.

The evolutionary Software Maintenance Process starts when a need is identified to change and allocate requirements, i.e. operational requirements, quality requirements, design requirements, documentation requirements, and implementation requirements.

These maintenance actions might include major re-design of the software.

Improvement maintenance is considered in the overall maintenance concept; however improvement maintenance is always triggered by a Change Request Process, or as part of the proposal for the next phase.

The analysis will be performed on the basis of problems that have been found during the recent phase, but not implemented for several reasons. Problems that would improve the usability or maintainability will be proposed as subject of improvement maintenance before entering the next regular maintenance phase.

5.10.2.1.4 Maintenance Environment

The development and testing environment will need to be retained and readied for the maintenance phase of a project. In addition, the maintenance staff may need to be trained in order to use the equipment if they are different from the development team. This includes not only the development, but also the test environment.

Obsolescence of development equipment, test environments and software will need to be considered as part of the maintenance plan. This is just as important as the operational software and hardware as without it the software will become un-maintainable.

Throughout the duration of the maintenance activity the test environment and simulators may need to be modified in order to keep them realistic. This may mean simulating degradations in spacecraft performance.

5.10.2.1.5 The Maintenance Plan

The maintenance plan defines the engineering processes undertaken to maintain the software. Since the plan describes the development and testing activities required to be undertaken on each new release of software, the maintenance plan can be derived from the Software Development Plan (SDP) and uses the same standards including ECSS-E-ST-40C/ECSS-Q-ST-80C tailoring.

The logistics to manage the facility and make it available to spacecraft operations and to the supplier in charge of the maintenance should be addressed in the maintenance plan.

For higher criticality software, the development and testing that needs to be performed will be at the same level as the initial build defined in the SDP. ECSS-E-ST-40C also states that the same tools and test environments should be used to test the maintenance releases as the original software. Basing the maintenance plan on the SDP will help to ensure that this occurs.

A maintenance plan is needed even when the supplier is in charge of the development and maintenance process. A suggestion for the content of the Maintenance Plan is provided in Annex C of this Handbook, this is not provided in the standard ECSS-E-ST-40C.

5.10.2.2 Long term maintenance for flight software

5.10.2.2.1 Characterisation of in-flight modifications

NOTE More information on software maintenance in flight is given in the SAVOIR document [Flight Computer Initialisation Sequence]

An in-flight modification is the mean to correct either the code or the data of an application embedded in a spacecraft. This application is either the central application software of the spacecraft, or an application running on equipment or a payload. Usually it is out-of-scope to change boot code which commonly contains the reloading mechanism itself. This part (and similar software), being considered as the nerve centre that allows any other data processing, are basically developed according to the stringent requirements of highly critical software (normally defined as cat B software after proper software criticality analysis).

The in-flight modification process relates to the software maintenance process depicted in section 5.10.

The definitions related to "in-flight modifications" are as follows:

a. Software reload: consists of reloading the full memory image of the on-board software regardless whether the memory addresses are changed or not, and then proceed with restart.

b. Patch: A patch consists of reloading specific parts of an existing binary only, possibly while the on-board software is currently running. A patch is well suited to correct or recover from critical problems, but can also be applied to implement a limited set of new requirements. It might cover additional aspects like urgent Operations product updates, on-board computer reboot, data pool changes, data base updates as requested by the system authority in charge (Change Control Board / CCB or other boards). A patch could be used until the final problem solution becomes available, i.e. as temporary solution to a problem, but could also be used as permanent solution if complete software reloading is not expected.

c. If the flight software is used to cope with degraded or malfunctioning hardware a patch can be used as permanent solution that will become unnecessary after hardware replacement has happened in the flight unit in orbit (when accessible, e.g. in a manned space segment). Then the patch needs to be de-installed to assure correct system handling.

d. Patch cluster: (best practice of COLUMBUS) up to 15 critical Non-Conformance Reports (NCRs) / System Problem Reports, are to be resolved by patching within following conditions: one single on-board computer reboot, but no data pool changes, no planned data base updates, only critical problems or urgent Operations product updates are covered by the maximum number of 15 NCRs / System Problem Reports.

e. Operation Patch cluster: (best practice of COLUMBUS) up to 50 NCRs / System problem Reports of Operations Data updates, are to be fixed temporarily by patching. Following conditions apply: one single on-board computer reboot only, no data pool changes, data base upgrade as needed, and, limited regression testing tailored to OPS product's needs in order to validate them.

f. OBCP load/reload : when OBCP capability is supported on board, loading or reloading of one or several OBCP enables in a safe and flexible manner to perform the expected changes without the drawbacks of patches or complete software reload.

NOTE OBCP are further discussed in 5.2.4.6.

Different patch capabilities can be used, like the patch that directly replaces a section of code (fix size sections are implemented and patched in a whole) or a new section that is uploaded in a specific part of the memory, with a derivation to this new part to be executed in replacement of the area of the one under modification. Other capabilities may also be proposed directly in RAM or in EEPROM that can be loaded on ground command.

There is no standard approach today, but the following criteria are used to define the appropriate capabilities in the context of the project. The trade-offs are conducted during the early phases of the project in order to derive the subsequent requirements applicable to the software (some of them having potential major impact on the software design itself, e.g. OBCP engine implementation):

a. availability of the system: the acceptable mission interruption duration is compared with the time taken to load a patch (or load a whole memory image), apply it and exercise before the system is back to mission mode.

b. TM/TC bandwidth: in conjunction with the previous criteria, this parameter will influence the duration of patching / reloading operations

c. available memory size

The two first criteria will drive whether it is acceptable to reload complete binary or if only specific modified code is reloaded in conjunction with the third one (storage of complete binaries need enhanced memory size).

According to those criteria, temporary and urgent workarounds to identified in-flight problems are often implemented as patches whereas planned set of "clean" modifications are implemented as full software binary load. Therefore it is very likely to implement both capabilities on-board for the Central Flight Software (high availability). In between these two capabilities, OBCP offers an intermediate flexible way having the advantages of a local and limited patch without the induced drawbacks and risks in terms of complexity of patch design and patch operations.

Furthermore in the case of high availability, it is worth to manage several binaries on board (at least two), with the associated capabilities:

a. to reload one of the binary of the software while the another one is under execution,

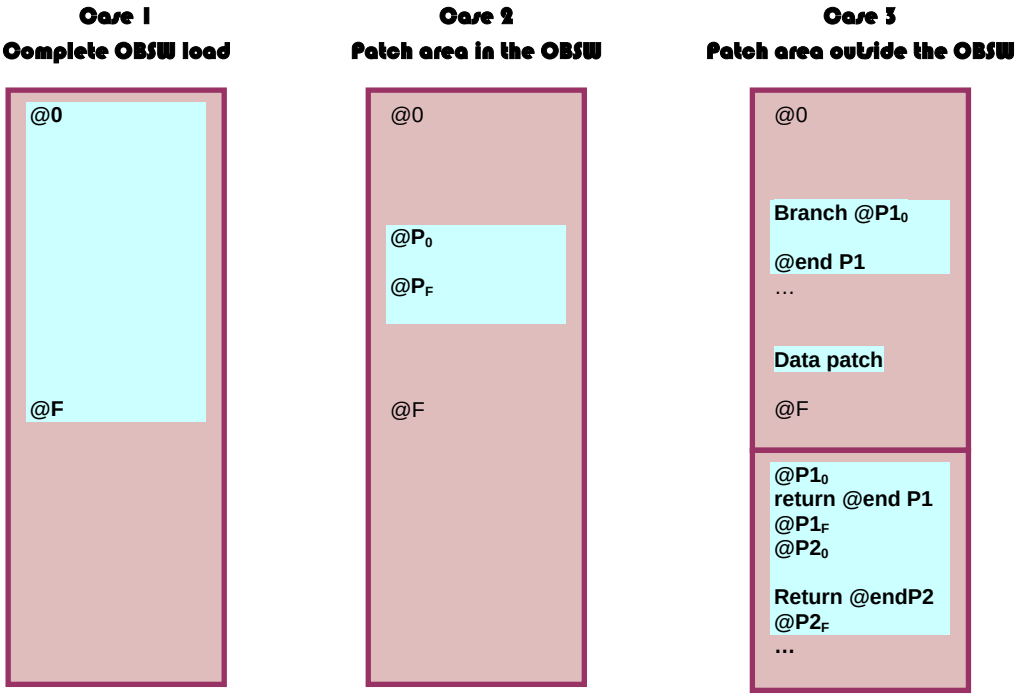
b. to switch the execution from one binary to another with the minimum interruption of the mission.

In the case partial portions of binary are required, dedicated mechanism may be implemented in order to load, store, activate, and inhibit safely patches in dedicated memory areas.

In case of low availability (e.g. secondary payload or experiment of lower importance), it may be acceptable to completely interrupt the subsequent software, completely reload and restart afterwards.

5.10.2.2.2 Patches scenarios and operations

A **Patch** is a SW correction / modification to be uploaded by the operations team through dedicated "patching" procedures using TC. Patch definition is a complex process which requires potential specific Coding directives (compilation and linking), assembler code analysis, and comparison with the original assembler code to identify the most efficient way to perform the correction. Different kinds of patches are possible:



In the case there is only one binary image of the OBSW on board, the patch can obviously not be applied on a part of the OBSW (On-Board Software = BOOT Software plus Application Software / ASW) which is currently under execution. It is therefore applied in a dedicated mode where the piece of code to be modified is currently not executed (some process may be inhibited specifically for that purpose). Specific operational care (if no dedicated protection mechanism is available) should be taken in order to prevent any corruption / modification of the patch / dump code itself. In particular, dumps may be performed to ensure that the memory content corresponds to the expected software binary code. Then after the modifications have been performed, the Application Software is switched back to normal operational modes.

It is highly recommended during validation of the modification to exercise the patch procedure itself to verify its robustness and efficiency.

The patch mechanism design and development is part of the initial software development process and can be validated as such during the qualification process.

The problematic part in the patch itself is to ensure the correct mapping in the patch application i.e. not patched zone needs to have an ISO-mapping demonstrated. The patch needs to be correctly linked in the hosting application.

5.10.3 Problem and modification analysis

-

5.10.4 Modification implementation

-

5.10.5 Conducting maintenance review

-

5.10.6 Software migration

-

5.10.7 Software retirement

-

6
Selected topics

The intention of this whole chapter is to address several selected subjects as described in the Note of 1. They are referenced in chapter 5 from the relevant ECSS-E-ST-40C requirements.

6.1 Use Cases and Scenarios

6.1.1 Relation to the Standard

Use cases and scenarios are applied in the requirement analysis processes, both for software related system requirements (clause 5.2.2 of the ECSS-E-ST-40C standard, and 5.2.3.2 for the scenario) and software requirements (clause 5.4.2 of the ECSS-E-ST-40C standard) to express functional, operational and HMI requirements, and in the software architectural design (clause 5.4.3 of the ECSS-E-ST-40C standard) to help identify software architectural components and trace to them.

The logical model mentioned in clause 5.4.2 of the ECSS-E-ST-40C standard can be supported by a set of detailed use cases and scenarios.

6.1.2 Introduction to use cases

NOTE In the two next sections, the “system” represents the entity to be considered for the modelling: a whole system, or a software or a part of a software. In ECSS-E-ST-40 context, "system" means "the software to be developed”.

The use case technique is a good way to capture a system's behavioural requirements with the addition of detailed scenarios. A use case indeed does not go into deep details and considers the system from the user point of view, as a black-box. It is limited to the questions “who”, “what”, maybe “what for” but never “how”. It is also easy to read and to understand. It eases therefore the communication between the users and the requirements producer; the requirements producer being the customer during the software related system requirement process (see clause 5.2.2 of ECSS-E-ST-40C) and being the supplier during the software requirement process (see clause 5.4.2 of ECSS-E-ST-40C).

The intended use of the system (or the intended use of the software embedded with the system) is specified by writing the requirements in terms of scenarios or use cases.

With this approach, the analysis of the system requirements allocated to software (see clause 5.2.2 of ECSS-E-ST-40C) or the definition of software requirements (see clause 5.4.2 of ECSS-E-ST-40C) consists of:

a. the identification of use cases (including the identification of actors, i.e. identification of the external entities interacting with the system via data exchange or by activating certain functionalities);

b. the elicitation of each use case (using a pre-defined template, e.g. using pre-defined sections to list functional and non-functional properties);

c. The properties expressed in the use cases are then used to derive the requirements of the software.

6.1.3 Identification of use cases

The first step in identifying the use cases is to determine the actors that interact with the system.

An actor can be a human, e.g. an operational person that administrates the ground segment. An actor can also be a piece of equipment or software, e.g. the mission centre that sends some commands to the command & control centre. In that case, the command & control centre is the system to be specified and developed; the mission centre is an actor because it interacts with the system via data exchange (commands).

A use case has a name, which is often the name of a main function of the system and a short description that describes briefly the actions performed by the system.

For example, the requirements producer of a "Monitoring & Control" system may identify the following use cases:

a. TM acquisition and packet extraction: Reading of the telemetry data frames received from the station; packet extraction from the received frames and their distribution

b. Decommutation: Decommutation and supervision of the received packets; distribution of the results

c. Communication with the station: Dialog with the station for TM/RM acquisition and TC/RC sending

d. TC encoding: Elaboration of commands binary profiles

e. TC sending: Commands sending to the satellite

f. COP1 management: Implementation of COP1 protocols and BC directives sending.

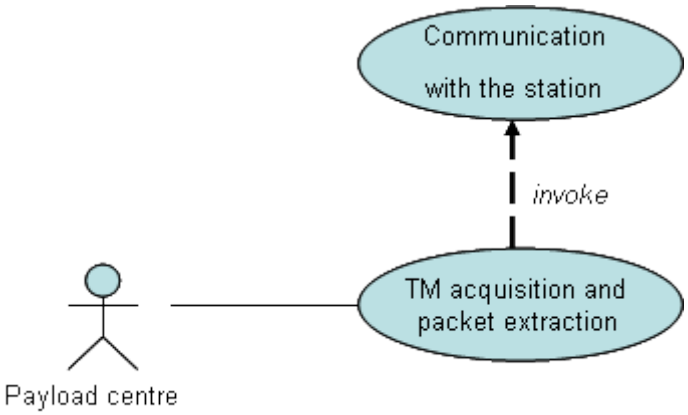
g. Station commanding: RC elaboration and sending

h. PUS services management: specific PUS services implementation.

Example 1: Use case list

6.1.4 Formalization of each use case

A use case diagram (cf. clause 6.1) may be provided to display the actors, the use cases (and the interactions between use cases. But it is not a formal way to specify the functional requirements of the system.



Example 2: Use case diagram

A textual specification is provided to express the use cases. The following template is proposed as an example, with specific sections as a guideline to the requirements capture. The part "Description" and "Non-functional constraints" can be used to derive functional and non-functional requirements.

SUMMARY

Short abstract describing the purpose of use cases: brief description of the service rendered by the system to the actors.

CONTEXT

Activation frequency, operational mode.

Actor or event that triggers the use case.

INTERACTION WITH OTHER USE CASES AND ACTORS

Identification of interactions with other use cases and actors. Illustrated by an excerpt of the principle use case diagram detailing the main aspects specific to this use case. The diagram can show actors and use cases of lesser importance not described within the main diagram.

PRE-CONDITIONS

Pre-conditions necessary for the execution of use cases.

DESCRIPTION

Description of the sequences (scenarios) and identification of properties (requirements). Errors are described in the" Exception" section. Complex scenarios can be illustrated using the sequence diagrams, activity diagrams or state transition diagrams (see Annex B for more details on the Use Cases and UML techniques).

POST-CONDITIONS

Conditions to be met by the system after executing the use case.

EXCEPTIONS

Known errors that lead to certain actions.

DATA

Input and output data of a use case.

NON FUNCTIONAL CONSTRAINTS

Constraints (requirements) other than functional related to the use case (e.g. volume, frequency, safety, reliability, integrity, availability, performance).

6.1.5 Definition and guidelines

A use case in software engineering and systems engineering is a description of a system's behaviour as it responds to a request that originates from outside of that system. In other words, a use case describes "who" can do "what" with the system in question. The use case technique is used to capture a system's behavioural requirements by detailing scenario-driven threads related with the functional requirements.

A use case is a series of related interactions between a user (and more generally, an "actor") and a system that enables the user to achieve a goal. In other words, a use case describes the system's behaviour as it responds to a series of related requests from an actor.

A use case describes a set of sequences of actions a system performs to produce a tangible result for a user. A use case includes a set of scenarios.

Use cases and scenarios support requirement analysis activities, i.e. they help capture who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internal details (black-box view of the system). They describe how the system will be operated to meet stakeholder expectations. They describe the system characteristics from an operational perspective and facilitate an understanding of the system goals.

Use cases involve the input from stakeholders including the different user roles for the system operation and maintenance. Their primary objective is to communicate with stakeholders to ensure that operational needs are clearly understood.

A complete set of use cases and scenarios specifies all the different ways to use the system, and therefore help identify all the capabilities required for the system and fix therefore the scope of the system.

The typical use case elicitation process is the following:

a. For each role an external entity plays that is relevant to the system, identify it as an actor and identify all the significant goals of the actor that the system will support. A statement of the system's value proposition is useful in identifying significant goals.

b. Describe a use case for each goal. Maintain the same level of abstraction throughout the use case. Steps in higher-level use cases may be treated as goals for lower level (i.e. more detailed) sub-use cases.

c. Structure the use cases. Avoid over-structuring, as this can make the use cases harder to follow.

d. Review and validate with users.

UML, and derived profiles as SysML, offer specific diagrams to express use cases. The definition of a use case includes a plain text describing all of the behaviour it entails, that is the main sequences of use, different variations on normal behaviour, and all of the exceptional conditions that can occur. A use case template to structure the textual description of the use case behaviour should be adopted.

Scenarios are single threads of behaviour that are used to trace the use cases to better describe the dynamics of the system, or in some cases are supersets of many single threads operating concurrently. Scenarios cover the aspects of the operational performance, including modes of operations, operation under designed conditions and behaviour required when experiencing mutual interference with other systems.

Plain text associated to the identification of the use cases, describing the behaviour of use case can be considered as an un-formal way to describe scenarios. Scenarios are described in UML or SysML using interaction diagrams or activity diagrams, or even state-charts. Other languages can be used to express scenarios such as the Message Sequence Charts (MSC) of the Specification and Description Language (SDL). Data flow or control flow diagrams can also be used.

Use cases and scenarios target operational and functional requirements, so they are independent of any possible implementation detail; in fact they should avoid designing a final solution or establishing unjustified constraints on the solution.

Uses cases and scenarios are simple ways to describe a system's behaviour, that may be defined and approved in collaboration with the stakeholders and system engineers, so that, once they have been approved, the related requirements may be formalized.

6.2 Life cycle

6.2.1 Relation to the Standard

Life cycle is addressed mainly in the clause 5.3 of the ECSS-E-ST-40C standard (in particular in sub clause 5.3.2 for its definition, sub clause 5.3.3 for the link with reviews, sub clause 5.3.6 for its synchronisation with the system), and in the Annex O for the Software Development Plan DRD.

6.2.2 Introduction

The ECSS-E-ST-40C requires the selection of a life cycle for space software projects.

The lifetime of software goes from its initial feasibility study up to its retirement. Complex space systems are initially being studied in early project system phases (0, A, and B) (see ECSS-M-ST-10C) in order to get a better assessment of what the final system should be capable for. The requirements are not yet stable enough to issue a business agreement (contract) for development. Space software experts are already being involved at this point as part of the system activities (co-engineering).

The system phase 0 (and very often phase A) serves to investigate complex technological challenges for feasibility. It is the goal to find combinations of techniques and methods to turn the initial ideas and dreams into reality and to exclude wrong tracks very early in order not to waste time, money and effort.

The system phase A (which can even be split in several system phases e.g. A1, A2) contributes to clarify customer and system requirements and to investigate plausible implementation possibilities taking state-of-the-art or best practice methodologies and techniques into account. The findings and the results are discussed between customer and the (intended) system supplier representatives at least in a mid-term and a final review milestone (depending on the business agreement there can be more coordination meetings as needed by the project). At the end of system phase A there is an agreed status that represents the feasibility of the future complex space system. Very often, alternative layouts w.r.t. implementation methods and system capabilities are provided to the customer. In most cases accompanying global cost estimations for the alternatives are prepared to enable the customer to select a variant that serves his needs best within his environment / situation.

System phase B (or system phases B1 and B2) is further clarifying the system needs with the goal to finally get a reliable set of requirements for the development system phase C. The set does not only consist of customer requirements, but also of system and initial software requirements enabling to start the developments. This may include the selection of the software life-cycle.

The software life cycle is a methodology used to guide and to organise the progress of software throughout its development and maintenance, in a structured way. It provides the framework supporting the software project planning, organization, staffing, budgeting, controlling, analysis, design and implementation. A software life cycle explains how the software engineering processes are mapped over time and project phases.

Processes within software life-cycle can run in different ways:

a. sequential, from requirements analysis to delivery,

b. in parallel or partially overlapping: to reduce time to deployment,

c. iterative, meaning run in several decoupled and partial steps: to progressively reduce project risks.

Moreover, a process can be activated and started using a draft work product as input (i.e. incomplete, not verified). There is a risk working on drafts but there is also an opportunity to run parallel work and to identify early problems with the drafts themselves. The project schedule shortens as more parallel work is performed. To some extent this is "concurrent engineering" applied on software development. However, when a process is working on a draft as input it may only produce a draft as output (draft-in/draft-out concept). Therefore, once the final version of the input is available the process is re-activated to "reconcile" the work done with the final input.

Life-cycle models are reference frameworks. A lot of software life-cycle models exist. A well-known life-cycle model is the waterfall model. Models such as V model, incremental, evolutionary or spiral are derived from the waterfall life-cycle. ". Other models take a perspective within the multiple levels of systems and software development, aiming at more globally modelling the software life cycle within the relevant customer-supplier network.

In general terms, life-cycles can be broken down into successive stages. Each of them uses the results of the previous stage to advance the life cycle, from one baseline to another one, after successful completion of the relevant activities. During these stages, usually at the end, milestones are planned (potentially through project reviews**)**.

Even if not considered as a life-cycle model, the Agile methodology is also considered in this chapter, as a succession of stages with dedicated successful end criteria.

Each review in the software life cycle is a major assessment performed by a designated team. This includes:

a. assessment of the validity of process output elements in relation with the requirements or the predictions;

b. decision to start the next stage of the project.

Whatever the selected software lifecycle model, there should be at the minimum the set of ECSS-E-ST-40-C reviews (i.e. SRR, PDR, CDR, QR and AR) synchronised with system level (see ECSS-E-ST-40-C sub clause 5.3.6). Then, they can be completed by technical reviews where appropriate to map to the software lifecycle (see ECSS-E-ST-40C 5.3.3.3). The sequence of software project reviews starts with the software requirements and definition System Requirement Review (SRR) and Preliminary Design Review (PDR) and continues with the justification and verification Critical Design Review (CDR), Qualification Review (QR) and Acceptance Review (AR). Although each stage is usually a part of a sequential logic, the start of the next stage can be decided before all the tasks of the current stage are fully completed (e.g. starting validation of some components although coding of others is not finished). Starting the work for the next phase in parallel can be considered if the induced risks are identified, accepted and monitored by the project.

Each software project has its own life cycle (taking benefits from existing models and experience) fitting the need of that project and its context (e.g. internal organization, customer needs). Frequently, several life-cycles models are combined to fit the various projects needs and constraints.

The ECSS-E-ST-40-C defines a set of processes. Each process is defined as a set of interrelated activities that transform inputs into outputs. However, ECSS-E-ST-40-C does not prescribe any specific order of execution of these processes over time nor does it assume a rigid sequential execution where only the end of one process triggers the start of the processes using its output. Nevertheless, the ECSS include some requirements relevant to the software life cycle definition:

a. ECSS-E-ST-40-C, Clause 4, in terms of system and software life cycles and related phasing, and sub clauses 5.3.2 in terms of specific requirements;

b. ECSS-Q-ST-80C, sub clause 6.1. in terms of the characteristics of the lifecycle that shall be identified.

With ECSS-E-ST-40-C, any life-cycle model can be applied, provided that the process and output requirements are satisfied. This chapter provides some recommendations to implement the requirements of ECSS-E-ST-40-C related to life-cycles, i.e. choosing and implementing a life-cycle appropriate to the software development constraints. In addition, other recommendations are also provided for specific technologies such as Database development (see clause 5.2.4.4), Autocoding (clause 6.5).

6.2.3 Existing life-cycle models

This chapter aims at briefly introducing the major models of life-cycles from which project life-cycles can be defined. Each of them has advantages and drawbacks. The choice of the appropriate model (or a combination of models) needs to well know the project context, constraints and objectives as discussed in this clause.

Additional guidance on life-cycle models can be found in the IEC ISO TR 15271 Guide for ISO/IEC 12207 (Software Life Cycle Processes).

6.2.3.1 Sequential models (Waterfall and V Models)

In this sequential model, the software evolution proceeds through an ordered sequence of stages, from one to the next up to the end of the life-cycle. As such, this model is mainly intended at supporting the proper definition of the pre-conditions and post-conditions of each stage and can help in structuring and controlling large software development projects from an organization and planning perspective. In such model, the Requirements Baseline is defined at the beginning of the life-cycle and represents the reference for the final software to be accepted. The principle is that each stage is completed before the next stage begins. For V model, in addition to Waterfall, testing stages are linked to the related definition stage.

6.2.3.1.1 Advantages

The Waterfall model enforces discipline in the life-cycle process and progress control. The verification is inherent in every stage of the life-cycle. Moreover, it is easy to contract this model.

6.2.3.1.2 Disadvantages

The Waterfall model requires a complete and mature specification before starting and the expected delivery is available only at the end of the life-cycle. Moreover, this model presents programmatic and technical risks for large software products, or when there are new technologies applied.

6.2.3.1.3 Utilization

According to the experience, the Waterfall is only considered as a reference, because changes often occur during the development, which affect the outputs of the previous stages, even up to the RB. The consequence is to re-run a sequence of delta Waterfall stages, which is not cost and schedule effective. This is the reason why other models are regarded as more suitable.

The Waterfall model is more efficient for software products with well-known customer needs. It is also appropriate when no technological risks are involved, e.g. when the target system (e.g. spacecraft, or a ground segment facility) is either already available or its design does not imply constraints on the software design to be verified early in the system life-cycle.

6.2.3.1.4 Stages and reviews

In the Waterfall life-cycle model, the ECSS software process activities are performed sequentially as well as the corresponding reviews:

a. Software requirements specification;

b. Software design and implementation (which includes the detailed design, the coding and unit testing);

c. Software validation;

d. Software delivery and acceptance (i.e. the validation against the applicable requirements baseline);

e. Software maintenance.

6.2.3.2 Iterative models

6.2.3.2.1 Overview of iterative models

Iterative life-cycles exploit the “soft” nature of software, and proceed by developing the software through a series of steps called iterations. Each iteration executes, on the functional scope of the iteration, all or a subset of the activities : requirements analysis, design, implementation, integration, and test (highlighted by the classical Waterfall model).

For each iteration, a decision is made as to whether the software produced is discarded, or kept as a starting point for the next iteration. Artefacts are in some ways “grown” or “refined,” from one iteration to another. The early iterations produce very small software, which gradually expands over a series of iterations to become the complete software. Feedback takes place from one iteration to the next. Each iteration reduces the number of problems and risks and becomes a solid base (baseline) for the next iteration. Eventually a point is reached where the requirements are complete and completely implemented and tested, or where it becomes impossible to enhance the software as required, and a fresh start is needed. The last iteration finally assures a full conformance of the delivered software product with the user needs.

Iterations may apply with different goals, using different life-cycle models, e.g.:

a. Need to explore unclear requirements e.g. with mock-ups (Agile, Spiral model),

b. Need to explore technically risky area e.g. through prototype (Agile, Spiral model),

c. Progressive design implementation (Incremental model),

d. Progressive functional evolution (Evolutionary model),

e. Performance improvement (tuning & optimization),

f. Test coverage improvement (e.g. beta version up to completely validated version),

g. Quality improvements.

Iterative life-cycles can be likened to producing software by successive approximation. Drawing an analogy with mathematical methods that use successive approximation to arrive at a final solution, the benefit of such methods depends on how rapidly they converge on a solution.

Advantages

In general, iterative life-cycles allow early delivery of partial software (e.g. implementing only a part of the Requirement Baseline or only partially validated). In addition, these life-cycles enable to start the software development and the system integration earlier, to mitigate the risk to detect too late potential issues (e.g. system - software co-engineering, software design). This is to be balanced with the disadvantage a).

Disadvantages

The following disadvantages are shared by iterative life-cycles:

a. at supplier level, there may be full or partial rework, such as additional non-regression test effort in case of too much coupling between several iterations, e.g. when there is a change on a part of a previous iteration,

b. The support and maintenance processes of already delivered releases start earlier and hence manage several versions in parallel,

c. There is additional effort for the increased number of required technical reviews (although there may be a balance between the extra number of technical reviews required and formal reviews with less effort).

Utilization

Iterative life-cycles are particularly appropriate for software projects where the customer needs early software releases (e.g. to perform early system tests or test facility set-up) or when system needs are not enough mature and require consolidation.

Consolidation of needs should be prioritized according to maturity needs (if there are no other specific constraints).

Moreover, to favour the overall consistency between the system and software life-cycles, it is required (ECSS-E-ST-40C 5.2.4.1) to detail the perimeter of each planned software release (i.e. the included functions) as early as possible and before starting the development.

The overall assembly logic as well as the overall testing logic (e.g. HW/SW integration, function validation, non-regression) should be adapted to be consistent with the selected life-cycle. See also section 5.5.4. In this aim, automation of tests is strongly recommended in order to ease non-regression tests.

Finally, to mitigate the risk related to the management of several versions in parallel, particular attention is given to the configuration and change control process and associated tools.

Stages and reviews

The Iterative model uses partially the Waterfall model for each of its iteration (e.g. without validation activities for iterations contributing to stabilise requirements, without engineering activities for iterations contributing to improve the quality level of a delivery).

As the life-cycle may involve several iterations, proper reviews are scheduled for these iterations. Not necessarily all reviews need to be done for each iteration, nor reviews necessarily need to be done with the same level of rigor for all iterations. Once a formal review is done for one iteration, the same review for the next iteration can be more efficiently conducted acting as a delta-review (focusing only

on whatever has changed). In other cases it is more efficient to hold an informal technical review for first iterations and then a formal review for the mast and complete iteration. These principles apply for software having an overall consistency and not to several software or several part of software without any coupling.

As a minimum, the last iteration completes the full adherence to what is required by ECSS-E-ST-40C, as to assure full conformance of the delivered software product.

Several iterative life-cycle models are presented in the subsequent chapters: Incremental, Evolutionary, Spiral and Agile.

6.2.3.2.2 Incremental

This iterative life-cycle model takes into account the progressive enhancement of the software and supports developing software by successive increments. Each increment corresponds to a delivered release of the software, the deliveries holding at pre-defined times. Each release provides an opportunity for the customer and the supplier to assess the technical baseline and the schedule. In this model, the Requirements Baseline, then Technical Specification and top level design are established once for all planned increments.

The software is designed in details, implemented, integrated, and tested as a series of incremental builds, where a build consists of a set of components interacting to provide a specific functional capability. At each stage a new build is implemented and then integrated into the architecture that is tested as a whole.

The detail definition of the perimeter of each increment is directly derived from the customer needs with respect to deliveries. As example, the initial increment could implement agreed basic functionalities of the Requirements Baseline and provide the basis for the additional components (implementing more advanced functions) of the architecture that are added in the following increments (e.g. for an on-board software only DHS, then DHS+AOCS, and finally DHS+AOCS+FDIR being the complete software).

In this model, the development of the next increment can start at the end of the previous one or can partially overlap leading to different variations of the model. However, even in a sequential versioning approach, some processes of the life-cycle for delivered versions are performed while the current version is still under development, including the maintenance, and sometimes, the operations processes.

Advantages

In addition to the Waterfall model advantages, the Incremental life-cycle enables to accommodate the specific delivery needs and associated schedule. It secures the system schedule, enabling a feasible software schedule. It allows conducting more efficient technical reviews focussed on limited functional topics, anticipating parts of the formal project reviews.

Disadvantages

The Waterfall model drawbacks remain, except for delivery available earlier with the Incremental life-cycle. In addition, the Incremental model has the drawbacks of an Iterative life-cycle.

Utilization

According to the experience, changes may occur during the development that affects the outputs of the previous stages, even up to the RB. The consequence is to re-run a sequence of complete stages, that is not cost and schedule effective. This is the reason why other models are regarded as more suitable.

This life-cycle is used when the software size is relevant with respect to the allocated time frame. It is also used when the target spacecraft system integration, verification, and qualification justify an incremental approach to progressively reduce manufacturing risks and minimise needs for hardware re-design. Finally, it is used when the software is to support the system needs (e.g. integration and qualification testing).

To be efficient, this approach should be based on a limited number of increments (e.g. 2 to 5), as decoupled as possible (see 5.3.5.2.2).

This model is justified for several reasons, including:

a. the identification of a subset of the software product capabilities earlier than the complete product (for instance for use at upper level by the customer in conjunction with other software or hardware items);

b. the size of the software project is demanding and leads to division of the effort;

c. the project budget needs to be spread over a longer time than the one required for a Waterfall approach.

Stages and reviews

The Incremental software development life-cycle is based on the iterative (rather than sequential) execution of the software process activities, with potential overlapping, resulting in incremental completion of the overall software product.

It is a variant of the Waterfall model, to which is closely related. The major difference is that a subset of the requirements (e.g. considered vital or critical for the final software product or needed for system test activities) is identified for early implementation and validation. The Incremental life-cycle starts from a complete RB, TS and architectural design then splits the development into an incremental approach at components level. Hence unique SRR, PDR (potentially anticipated by a SWRR), QR and AR are performed once for all the increments.

The software CDR (potentially anticipated by DDR), TRR and TRB can be conducted as a set of increment-dependent technical reviews, concluded by the formal CDR for the final version.

6.2.3.2.3 Evolutionary

The Evolutionary model recognises that the evolution of software takes place according to prescriptive steps, but accommodating all the changes that are required during the life of the projects.

The Evolutionary model plans for re-establishing the software Requirements Baseline at several defined milestones. In this view, the final software is developed through a controlled sequence of versions accounting for the evolution of requirements in major steps.

This Iterative life-cycle is an extension of the Incremental life-cycle with successive evolutions of the Requirements Baseline. Each iteration takes into account the RB changes.

Advantages

In addition to the Incremental model advantages, this life-cycle is mainly a way to start the development based on a frozen set of requirements, corresponding to the perimeter of the iteration, even when some requirement are not mature enough. It allows the anticipation of problems, feasibility check, etc. and enables a final accommodation of schedule needs.

The planned versions can benefit in fact from the user's feedback, to correct anomalies and to improve the software functionality and performance over the lifetime.

Disadvantages

The disadvantages are the same as for the Incremental model.

In the Evolutionary model, there is an additional activity to establish different Requirement Baselines and Technical Specifications for each version and to consolidate them. This flexibility needs to be balanced by the induced risk that the overall software design could not be robust enough to anticipate the further evolutions of requirements. This is in particular important for the real time design.

Utilization

This life-cycle is used when, even with mature enough customer needs, the Requirements Baseline consolidation cannot be fully achieved early in the life-cycle.

The Evolutionary life-cycle is effectively suitable for long-living software products. A limited and planned number of specification evolutions should be managed, leading to a limited number of product versions.

In addition, in case there is an overlap between the different iterations, the software processes need to be managed carefully. This implies that for example, the specification of the next version is completed before the validation of the previous version has been accomplished, or even before its validation starts. The commitment of the customer-supplier on the new version requirements can be in fact invalidated by the results (e.g. non-conformance) of the previous life cycle iteration.

The key principle of such model is to establish the overall software design and interfaces during the first iteration. In order to mitigate the risk on the initial software design robustness, it is beneficial to inherit from former knowledge on System and Software architecture, or to use a generic architecture, or to reuse an existing and proven architecture.

Stages and reviews

The Evolutionary life-cycle model is based (as the Incremental model) on multiple releases of the software, with the difference that for each release a, complementary (i.e. additional and modified) software requirements specification and architecture phase is conducted.

Each release implements the complementary requirement specification starting from the previous release. That is, each release includes the re-use of all the components of the previous release that still satisfy the complementary requirements.

Each Requirements Baseline is assessed through a dedicated SRR, meaning either several SRRs or one SRR for the first evolution and delta SRRs for the next ones. Then all the Waterfall model processes are performed sequentially up to the end of the iteration validation. However, due to the dependence between the evolutions, the review process needs to be adapted.

Depending on the overall schedule and content of the different evolutions, the PDR (potentially anticipated by a SWRR) is performed either through multiple reviews (intermediate PDRs with limited objectives for the first versions and then a “closing” PDR for the final version), or simply once for the final version.

The software CDR (potentially anticipated by DDR and TRR), QR and AR are conducted once the software is complete (hence for the last version) with potential technical reviews for the previous versions. For efficiency reasons, it is recommended to assess the correctness of each version (e.g. mainly the related evolution of the SVS-TS) during the corresponding review.

6.2.3.2.4 Spiral

This is a risk-driven model for software processes based on iterative development cycles in a spiral manner:

a. inner cycles are devoted to project risks minimisation, through approaches such as early analysis, prototyping, simulation, expert advice, or benchmarking, and associated progress in the development, and

b. outer cycles implementing the software, once risks are solved, with an appropriate life-cycle model.

Whilst the radial dimension accounts for cumulative development maturity, the angular dimension represents the progresses made in accomplishing each development cycle of the spiral. Each cycle is accompanied by a risk analysis to determine how the development spirals out and evolves, depending on the achieved confidence in the previous cycle. Each cycle includes therefore the planning, seeking for alternatives in design and development, the evaluation of these alternatives, and risk analysis as well as it includes the software development processes to accomplish the planned development cycle.

Each cycle of the spiral is completed by a review of work products of that cycle, including plan for the next cycle, to assess the level of mitigation of the initial risks and the level of maturity to continue.

Advantages

The Spiral model favours consolidation of not enough mature inputs, i.e. needs, requirements, design. The emphasis of the spiral model on alternatives and constraints supports the reuse of existing software.

Disadvantages

The Spiral model has the drawbacks of an Iterative life-cycle.

The main disadvantage of the Spiral model however is that too much risk analysis can be costly compared to the expected benefits. The development team can be weak at assessing risk.

Moreover, early development of low quality prototype can lead to a low overall quality of the final product.

The flexibility of this model needs to be balanced against the effort associated with possible refactoring of the design and code for accommodating evolving requirements.

Utilization

The Spiral model is particularly appropriate for software projects with not enough mature upper level needs or interfaces. It is used when project risks are evaluated to be relevant at both customer and supplier level. It is also used when the system life-cycle spans over duration supportive of the multi-cycle model of this model.

When lower level suppliers provide software, both customer and suppliers perform all risk analysis before the supply contract is signed.

Stages and reviews

The Spiral life-cycle model encompasses features of the staged life-cycles, as well as of use of prototyping. It is strongly complemented by a continuous risk assessment and analysis process. The Spiral model can use the Waterfall model for each of its cycles (i.e. code a little, test a little), as implied by the risk assessment and analysis and associated risk avoidance or mitigation strategies.

The adoption of this model impacts the planning and activities in the organizational, primary, and supporting processes. The definition of the review logic and associated milestones, as well as the project schedule and project status determination and reporting are modified.

All the process elements are tailored and optimised throughout the project life-cycle. Those include:

a. project phase Page 158 - section 5b 10-13;

b. have spurious '1's before the testing and planning, for the precise determination of objectives, alternatives, and constraints to be considered;

c. risk management, for analysis of alternatives and definition of risk mitigation or avoidance strategies;

d. development process, for definition of the spiral cycles activities and reviews;

e. quality assurance process, in all aspects.

Generally, the Spiral life-cycle outputs and reviews are made in line with the life-cycle model assumed for each cycle.

For example, a project with un-mature requirements starts a first cycle aiming at analysing the needs through prototyping; once requirements are enough mature, a SRR can conclude this cycle. Then, the project continues the next cycle with a classical incremental model, with associated reviews.

6.2.3.2.5 Agile

The concept of iteration is taken to an extreme with a software baseline produced in a very short period (several days or weeks rather than months). Several methods (not life-cycles strictly speaking) implement this principle, such as Extreme Programming, Crystal Clear and Scrum, all of them being based on the Agile method. These methods are developed with the intent of accepting changing requirements and delivering software to satisfy customer needs in the most efficient way possible.

Agile is a software development approach based on four unifying fundamental values:

a. Individuals and interactions over processes and tools;

b. Working software over comprehensive documentation;

c. Customer collaboration over contract negotiation;

d. Responding to change over following a plan.

In this approach, the development is split into very small increments (named “sprint” in the Scrum methodology). Each increment involves a team working through a full software development cycle (i.e. Waterfall) including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to the customer or the end-user. This intends to minimize the overall risk and allows the project to adapt to changes quickly.

A specific aspect of the Agile method iterations is that they have functional expansions and are time-boxed: in case of problems, deliveries always occur on time but with reduced functionality (instead of postponing delivery waiting for the full functionality).

Agile methods focus only on essential development efforts in the creation of working software that satisfies the customer’s needs. An on-site customer provides feedback on working portions of the software, thereby reducing the dependence on written documentation alone and increasing the possibility of delivering a satisfactory product.

Agile development has been widely seen as being more suitable for certain types of environment, including small teams of actors. Face-to-face communication and team member’s ingenuity is encouraged over documentation and prescriptive process.

Requirements are defined and prioritized in collaborative effort of both customer and supplier (value-driven approach).

Advantages

Even if particularly useful in domains where requirements are not mature at all or when frequent requirements change and speedy development are more prevalent, these methods can be also used with a stabilised Requirements Baseline.

A working product is delivered at each stage, tested and evaluated by the customer and/or end-user. The testing is a way to define or refine the next requirements that will be implemented during the following stages.

Disadvantages

These methods need a strong and continuous involvement and commitment of the Customer throughout the development, and even recommend co-location of customer/supplier teams. They are mainly dedicated to small software teams.

NOTE This flexibility needs to be balanced against the effort associated with possible refactoring of the design and code for accommodating evolving requirements.

This method can increase the pressure due to the high development rhythm.

Utilization

This approach is used when the requirement baseline cannot be fully defined at the beginning of the software development (either because the customer has no time to specify the software in a detailed manner, or the end-user does not know his wants relative to the software product). Anyway it is used only if the customer is ready to get involved during the development.

Stages and reviews

In theory, each Agile iteration can be considered as a "mini" Waterfall life-cycle and the related reviews should be conducted for each sprint or could be tailored and carried out as technical reviews. However in practice, due to the usual duration of a sprint (e.g. 4 weeks), the related formal reviews is not feasible for each sprint.

The most pragmatic approach is usually to combine the use of an Agile methodology with an Evolutionary life-cycle: different versions (or releases) of the software are planned. A new version is delivered after a predefined set of builds developed during a set of sprints. Each sprint can be considered as a contribution to a preliminary SRR, PDR, CDR or QR. After some iterations (i.e. a set of sprints), a formal SRR can be scheduled, then the same principle applies to PDR, CDR until reaching the QR. Documentation is made in an iterative manner up to the final one delivered for the related review. In the spirit of the Agile method, the decision to hold an ECSS-E-ST-40C project review is made jointly between the supplier and the customer during a sprint.

Finally, even if generally the expected ECSS documentation and activities can be applied with such method, more than for other life-cycle, the choice of the Agile method needs a strong software ECSS compliance analysis particularly for the schedule and reviews management, early negotiated and agreed by both parties.

6.2.3.3 Multi-level (nested)

Large software projects may be conveniently organized in a hierarchical decomposition of sub-systems and a specific life-cycle defined for each sub-system.

In a multilevel software life-cycle, coding of software is done only at the lowest decomposition level (i.e. the software products level). At the upper levels (that can be software systems, subsystems, assemblies), the life-cycle process activities are similar to the space system level, having as major objectives to provide the Requirements Baseline to the lower level and to integrate the lower level components.

The life-cycle starts by a top-down succession of SRR/PDR at each level of decomposition, followed by a full software development cycle and reviews for each of the bottom levels components, each one followed by a bottom-up succession of QRs and ARs at each level of decomposition.

See also chapter 4 of ECSS-E-ST-10C introducing the multi-level decomposition principles.

6.2.4 Choosing a Software life-cycle

As previously introduced, each life-cycle model is focused on specific topics, e.g. incremental deliveries, evolution of the specification, iteration on the validation. The characteristics of the different models presented here are summarized in the following table.

Table 6-1: Choosing a Software life-cycle

Model	Requirement Baseline	Iterative life-cycle	Short iteration	Risk driven	Utilization comment
Waterfall model	Frozen	No	N/A	No	When the customer needs are well-known and relatively stable. When no technological risks are involved.
Incremental model	Frozen	Yes	Depends	No	When the customer needs are well-known and relatively stable. When incremental deliveries are required. However, need to manage several versions in parallel. When no technological risks are involved.
Evolutionary model	Evolving	Yes	No	No	When, even with enough mature customer needs, the Requirements Baseline consolidation needs to be achieved in several steps. However, need a design robust enough to anticipate the RB evolutions. When early deliveries are required. However, need to manage several versions in parallel. When no technological risks are involved.
Spiral model	Evolving	Yes	Depends	Yes	When there are technological risks, e.g. not enough mature needs, new technology. However, require a larger schedule and cost to mitigate the risks.

Model	Requirement Baseline	Iterative life-cycle	Short iteration	Risk driven	Utilization comment
Agile	Depends	Yes	Yes	Yes	When there is a need to quickly and continuously adapt and satisfy customer needs changes (e.g. on requirements or delivery needs). However, there is a risk that the final software does not meet the initial requirements and budget. For small software teams, when there is a strong and continuous involvement of the Customer throughout the development as well as co-locative customer/supplier teams.
Multi-level	Frozen	No	N/A	No	For very large software with complex industrial organisation.

To fit the various project needs and constraints, it is frequently necessary:

a. to establish a specific life-cycle on the basis of principles coming from several life-cycle models. For example develop an Agile sprint based on a Waterfall model and other sprints based on an Incremental model;

b. to combine several of these life-cycles models. For example an Evolutionary life-cycle can be built from different basic models for each planned version, so that for instance the first version is incrementally developed, while the following can be based on the Waterfall model.

Moreover, the choice of the life-cycle depends on many factors:

c. customer oriented: e.g. related to the maturity of the Requirements Baseline, the specified or expected evolution of the software Requirements Baseline during the life of the project, the intermediate delivery needs;,

d. the link between the system and the software;

e. internal to the supplier: e.g. the used software technology, the reuse policy or the workload management.

The following chapters develop these factors: the system-software relationships, the Customer-Supplier link and the influence of used technologies.

6.2.4.2 System-Software relationships

Software is started by system level requirements allocated to software, and software is needed to perform system integration activities. Therefore, software life-cycle and system life cycle need to be harmonized and synchronized at some point in time. Moreover, if iterations are applied at system level, corresponding iterations may be propagated down to software level. However a single system iteration does not preclude multiple iterations applied at software level. It is also acceptable that system integration activities be started using not fully tested versions of software, as long as those tests are repeated on the final version of the software, once available. Inversely, software may be tested on an incomplete or not fully tested version of hardware, as long as those software tests are repeated on the final hardware version once available.

Different sub-systems may have different life-cycles. However when they interface to one another their evolution are harmonized and synchronized at some point in time. It is however acceptable that

one sub-system be tested using only a prototype incomplete version of another sub-system, as long as those tests are repeated on the final version of the other sub-system, once available.

The system life-cycle process provides the framework in which the software life-cycle is defined, taking into account a set of constraints including:

a. the policy adopted at system level for the procurement, development and manufacturing in one or more versions (or models, in the case of the space segment) of the system configuration, with impact in terms of integration of hardware and software and the associated testing;

b. the system review logic, as constraining the definition of the software life-cycle schedule and associated reviews and milestones;

c. the system needs for technology demonstrations or pre-development activities that can imply the need for early software versions to be engineered and integrated with the hardware configuration;

d. the planned evolution of the system requirements baseline and interface requirements-design that impacts a pre-planned evolution of the software requirements and its interface definitions across the software life-cycle;

e. the number of system configurations that are expected. For instance, a spacecraft can exist in different flight units with the same or with different hardware and software configurations. The software development takes this into account to support all configurations.

For each system, a careful analysis is performed in order to define the proper framework for the software life-cycle and to determine the set of constraints to be applied on the software development, delivery, and acceptance. The definition of a generic process model capable of expressing all the potential system constraints is not feasible, due to the specificity of each spacecraft space and ground segment conception, mission, and expected lifetime. However, general elements can be provided for a large variety of systems and applications.

Moreover, due to the need for reduction of the overall spacecraft development schedule, it becomes more and more difficult to have all the system requirements for software in time before the software development can start. The software life-cycle is constrained to accommodate for the late consolidation (or even gathering) of some requirements. Also, a requirement change process can be defined and applied to cope with changing customer constraints or mission/spacecraft requirements.

For these reasons, the software reviews are tailored to be adapted to the selected life-cycle. In the tailoring, the major criterion is to aim at developing the software at the same speed as the corresponding subsystem. Starting the software development too early leads to major redesign, when the system becomes mature. Waiting for all subsystems to be mature before starting the software can cause the software to arrive too late.

6.2.4.3 Customer-Supplier relationship

As explained above, the customer specifies the project constraints/specific needs (specific documentation inputs or outputs, sub-contractor or not) in addition to the product requirements:

a. interaction and collaboration of the software engineering with the other system disciplines;

b. milestones to control the next level supplier activities and, later-on at acquiring and integrating into the system configuration these items;

c. take into account the project requirements for multiple versions of the software, for system incremental integration and verification process.

The customer can consider additional reviews to be performed in order to improve the visibility at customer level and the ability to manage the overall project risks. However, this can decrease overall

efficiency of the supplier. In any case, reviews aiming at technically checking the correctness and completeness of the outputs and meetings aiming at monitoring the project progress are clearly distinguished.

The purpose of the software reviews is for the customer to accept the outputs folders specified in ECSS-E-ST-40C and ECSS-Q-ST-80C (e.g. TS) and to get the software in a given status (e.g. validated, qualified, and accepted). The ECSS-E-ST-40C (sub clause 5.3.6) provides a logical relationship between the software reviews and the spacecraft reviews.

At supplier level, it is recognized that rigid process sequencing is usually unsuitable for real industrial projects taking into account e.g. product-line or reuse policy, and different approaches are often more suitable at least for:

a. Reducing the time to market by executing processes in parallel as much as possible (i.e. concurrent development);,

b. Reducing risk by executing processes repeatedly and progressively (i.e. iterative and incremental development).

The supplier can also consider specific project technical reviews to help him properly scheduling the project progress.

6.2.4.4 Influence of the used technologies/methods/languages

Several software-engineering techniques support the adoption of the different life-cycles and can help to cope with some of their weaknesses. For example, model driven technologies may support a fast loop between specification, design and coding on one side (e.g. UML formalism, automatic code generation), specification and testing on the other side (e.g. with formal methods allowing to prove some properties). These technologies should not drive the selection of the software project life-cycles.

6.3 Model based Engineering

6.3.1 Relation to the Standard

Software development makes more and more use of models. The trend in software (and system) engineering is to move from a document centric development towards a model centric development. Except for the initial requirements, and for some non-functional requirements, that will remain in textual language, the artefacts that represents the software, its interface, its architecture, its real-time behaviour, its tests, will be mainly models.

This has a large impact on the Standard. Although the software engineering will always include the same concepts (specification, design, implementation, testing), the interpretation of the Standard will evolve with the appearance of models.

For example, models used for autocoding could be considered either as a form of design still producing code, or as higher level abstraction coding language replacing code. The notion of unit testing has consequently a different interpretation.

This impacts, in ECSS-E-ST-40C, the Software Development Plan. In particular:

a. Clause 5.3.2.4, addresses directly autocode,

b. Clause 5.4.2.3 addresses the software logical model (that must be verified in 5.8.3.2.a.11),

c. Clause 5.4.3.1 addresses the architectural design (that must be verified in 5.8.3.3.a.),

d. Clause 5.4.3.3 mentions the computational model (discussed in the following Real-time section of this document)

e. Clause 5.5.2.3 introduces the detailed design model with its static, dynamic and behavioural views (that must be verified in 5.8.3.4.a.9),

f. Clause 5.8.3.13 is the verification of the behaviour models,

These concepts are mapped on the appropriate modelling technology in the Software Development Plan. Any modelling technology is better suited for (i) a given level of abstraction, (ii) a particular viewpoint on the software and (iii) a particular verification objective. The development plan takes care to associate the right modelling technology to the right development step. In addition, the development plan analyses the relationship between the various models, their interaction in the life cycle, and the various roles associated to their development, verification and validation.

As a non-exhaustive example:

The software logical model has the level of abstraction of software requirements. It uses technologies such as use cases, class diagrams, and state machines. The architectural design and the static view of the detailed design model are described with components, or architectural languages. The detailed design is also based on specific targeted modelling languages, such as data flow diagrams. The dynamic view is expressed with the real-time features offered by the component model. The behaviour is expressed with state machines. Interfaces are expressed with data modelling languages.

6.3.2 Definition and guidelines

6.3.2.1 Modelling

A model is a representation of an entity at a higher level of abstraction than the implementation. The software discipline, due to its intellectual nature, intrinsically offers modelling capability. Moreover, the models can be applied at any step of the intellectual exercise to develop software. When the steps are appropriately selected, the models offer the possibility to enrich stepwise the knowledge related to software implementation. This is an important evolution compared to the transformation of natural language into code through a design.

The introduction of modelling at system level enlarges the software life cycle that can be walked through step by step, model by model. Each modelling steps carries a targeted increase of the knowledge about the software, and preserves the correctness properties demonstrated in the previous step.

Model evolution through refinements, transformations, meta-model mappings, and code generation, preferably automated by tool support, are the basis of the model driven engineering approach for software systems.

Model driven development is an emerging paradigm that solves numerous problems associated with the definition, composition and integration of large-scale and complex systems. Model-driven development elevates the software development to higher level of abstractions.

6.3.2.2 Model Based System Engineering

Model-based systems engineering (MBSE) is an INCOSE initiative for achieving “the formalized application of modelling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases. MBSE is part of a long-term trend toward model-centric approaches adopted by other engineering disciplines, including mechanical, electrical and software. In particular, MBSE is expected to replace the document-centric approach that has been practiced by systems engineers in

the past and to influence the future practice of systems engineering by being fully integrated into the definition of systems engineering processes."

Applying MBSE is expected to provide significant benefits over the document centric approach by enhancing productivity and quality, reducing risk, and providing improved communications among the system development team.

6.3.2.3 Model Driven Architecture

Model Driven Architecture (MDA) is a major initiative of OMG to achieve a cohesive set of model-driven technology specifications, which represents systems using the UML language, along with specific profiles. MDA is not itself a technology specification.

The three primary goals of MDA are *portability*, *interoperability* and *reusability* through architectural separation of concerns.

The Model-Driven Architecture starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform.

MDA provides an approach for, and enables tools to be provided for:

a. specifying a system independently of the platform that supports it,

b. specifying platforms,

c. choosing a particular platform for the system, and

d. transforming the system specification into one for a particular platform.

The basic MDA pattern involves thus defining a platform independent model (PIM) and its automated mapping to one or more platform-specific models (PSMs). Therefore, model transformation allows the same model expressing business logic to be realized in multiple platforms with the maximum *automation.*

The scope and content of each level of model needs to be strictly defined and standardized. This is where MDA offers the most significant advances compared with previous industry best practices: this standardization enables tools to support and automate the process.

The use of modelling in the different phases of the development process is considered of crucial importance in order to successfully realize complex systems.

Remark: Model exchange and tool interfaces deserve particular attention due to lack of standardization across different vendors (i.e. most MDA tools implement their own proprietary extensions not compatible to others).

6.3.2.4 Component model

There is no single widely accepted definition of a software component (or component for short).

The motivation is composing software application of reusable pieces which communicate to each other using mutually agreed protocols.

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. The following criteria are met by a component:

a. Multiple-use;

b. Non-context-specific;

c. Composable with other components;

d. Encapsulated, i.e., non-investigable through its interfaces;

e. A unit of independent deployment and versioning.

However, a software component is a software element that conforms to a component model (see definition below) and can be independently deployed and composed without modification according to a particular composition standard. In other words, the definition of a software component is a part of a component model, and this term is interpreted in the context of the corresponding component model.

The impact of the adoption of a component based development on the ECSS-E-ST-40C should be elaborated on project bases (e.g. in form of governance model).

6.3.2.5 Model Based Testing

Nowadays the practice of SW produced by manual coding is decreasing, SW is more and more automatically generated starting from models.

In the classical approach, unit, integration and system test procedures are usually handwritten, only coverage tests can be partially automated using specific commercial tools. The focus is now on the development of models, the code is automatically generated from these models. The abstraction level is higher than in the past.

In this new scenario also the testing methodology changes, as the model itself is tested, If the model is verified, and validated against the SW requirements, and the code generator is validated or proven in use, then model V&V, achieving the model coverage, can replace (manual) unit and integration testing.

V&V of models requires combining a dynamic approach, by animation or simulation, together with static analysis techniques.

V&V of models may be applied to the different view/diagrams, depending also on the applied methodology:

a. Interaction scenarios are associated to use cases to describe use or component interaction , in relation to SW requirements, to help to verify the system model behaves according to the specification (model validation).

b. State machine diagrams show the SW components functional behaviour and state evolution related to events. Formal verification as well dynamic approaches can help to verify the system model behaves correctly (model verification).

NOTE Starting from validated and verified models, model-based testing may control automatic test generation. Scenario-based and state machine-based testing approaches can be combined by defining an integrated and practical method for the strategic generation of model-based test suites.

6.4 Testing Methods and Techniques

6.4.1 Relation to the Standard

Testing is referred to in the ECSS-E-ST-40C Standard as an activity for unit test (sub clause 5.5.3.2), integration (5.5.4.1), validation (5.6.2.1) and acceptance (5.7.3.1). In addition, the Quality Assurance handbook on software dependability (ECSS-Q-HB-80-03A) emphasizes particular testing techniques that are especially appropriate for software dependability verification. These techniques are presented in the following sections.

6.4.2 Introduction

The testing methods and techniques are presented by first defining different test objectives: emphasize on specific aspects of the software product, like robustness, performance or interfaces. Secondly, some specific testing strategies are presented which can be used for test approach and definition of some of the test objectives presented above. These techniques can be either white box or black box techniques (to be specified when introducing each of them).

The process or activities versus the criticality categories is provided by two different tables: one relating the criticality categories versus the testing techniques; and the second one detailing the overall testing activities such as planning, execution, reporting.

6.4.3 Definitions

6.4.3.1 Black box test

Test of the software without considering the internal logic.

6.4.3.2 Test coverage

The degree to which a given test or set of tests addresses all specified requirements for a given system or component.

[IEEE 610.12]

6.4.3.3 Test objective

Purpose or focus of the test.

6.4.3.4 White box test

Check of the internal logic of the software.

6.4.4 Test objectives

6.4.4.1 Interface testing

The major test objective of performing interface testing is:

a. - Locate errors that can prevent the system from operating at all or

b. - locate errors in timing of interface responses

Interface Testing is similar to interface analysis, except test cases are built with data that tests all interfaces. The types of errors detected with this kind of testing are:

a. -Input or output description errors

b. -Inconsistent interface parameters

NOTE Without the use of a CASE tool for the production of the design, or code, manual searching for interface parameters in all design or code modules can be time consuming.

Several levels of detail or completeness of testing are feasible. The most important levels are tests for:

a. all interface variables at their extreme values;

b. all interface variables individually at their extreme values with other interface variables at normal values;

c. all values of the domain of each interface variable with other interface variables at normal values;

d. all values of all variables in combination (this is only feasible for small interfaces);

These tests are particularly important if the interfaces do not contain assertions that detect incorrect parameter values.

One example of interface testing is the so-called '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 satisfies the high-level requirements. Typical errors revealed by this testing method include:

a. Incorrect interrupts handling.

b. Failure to satisfy execution time requirements.

c. Incorrect software response to hardware transients or hardware failures. for example un-correctable EDAC error or a "FPU exception.

d. Inability of built-in test to detect failures.

e. Errors in hardware-software interfaces.

f. Incorrect behaviour of feedback loops.

g. Incorrect control of memory management hardware or other hardware devices under software control.

h. Stack overflow.

i. Incorrect operation of mechanism(s) used to confirm the correctness and compatibility of field-loadable software.

j. Violations of software partitioning.

Another example of interface testing is the so-called 'Requirements-Based Software Integration Testing' that concentrates to exercise the code on the components inter-relationships. 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 can 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:

a. Incorrect initialization of variables and constants.

b. Parameter passing errors.

c. Data corruption, especially global data.

d. Bad end-to-end numerical resolution.

e. Incorrect sequencing of events and operations.

6.4.4.2 Robustness Testing

Robustness testing is more focused on evaluating specific dependability aspects of the software. Robustness testing goal is to demonstrate the ability of the software to cope with abnormal environment conditions or behaviour (abnormal inputs or catastrophic 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 target computer emulator or a host allowed by the software requirements.

h. environment conditions should also be exercised to extreme or abnormal situations to ensure protection mechanisms work as expected

6.4.4.3 Performance testing

Performance testing is focused on the requirements specification that includes throughput and response requirements for specific functions, perhaps combined with constraints on the use of total system resources (e.g. CPU usage, memory usage, timing conditions). The proposed system design is compared against the stated requirements by:

a. producing a model of the system processes, and their interactions;

b. determining the use of resources by each process, for example, processor time, communications bandwidth, storage devices, etc.

c. determining the distribution of demands placed upon the system under average and worst-case conditions;

d. computing the mean and worst-case throughput and response times for the individual system functions.

For simple systems an analytic solution can be used, while for more complex systems some form of simulation should be used to obtain accurate results.

Before detailed modelling, a simpler ‘resource budget’ check can be used which sums the resources requirements of all the processes. If the requirements exceed designed system capacity, the design is infeasible. Even if the design passes this check, performance modelling can show that excessive delays and response times occur due to resource starvation. To avoid this situation, engineers often design

systems to use some fraction (for example 50%) of the total resources so that the probability of resource starvation is reduced.

Performance testing objectives are to ensure that the system meets its timing and memory requirements, and acquire measures to be taken as soon as the target operational equipment is available. Stress testing is the technique mostly used for this type of tests.

6.4.5 Testing strategies and approaches

6.4.5.1 Introduction

For the above test objectives, different testing strategies can be used. Some testing strategies focus on the method used for the selection of input data. For both white box and black box techniques, the selection of input data and objectives of the test cases drives the testing activities to be performed. Thus black box testing is testing against the specification and to better discover the part of the specification that was not fulfilled. White box testing is testing against the implementation and to better discover the part of the implementation that is faulty. In order to fully test a software product both black and white box testing should be used.

White box testing can be planned and produced based on low level software functionality (e.g. pseudo-code). White box testing is much more laborious in the determination of suitable input data (when not using a tool) and the determination if the software is or is not correct.

Test planning should start with a black box test approach as soon as the specification is available. White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Test cases can be derived depending on the criticality of the software (see ECSS-Q-ST-80C requirement 6.2.3.1) and for example to:

a. guarantee that all independent paths within a module are exercised at least once.

b. exercise all logical source code structure decisions on their true and false sides.

c. execute all loops at their boundaries.

d. exercise internal data structures to ensure their validity.

For dependability and safety testing, there are more specific techniques to be highlighted, being either white box or black box. Among these techniques, the following ones can be highlighted:

a. Fault injection

b. Stress testing

c. Equivalence class and input partitioning

d. Boundary value testing

6.4.5.2 Testing techniques

6.4.5.2.1 Fault injection

Fault injection is focused on reducing the impact of any residual fault by first determining how badly the software can behave if it is faulty and see how effective its fault-tolerance mechanisms are. Potentially specific software failures can be removed; though it is a strategy still very much hardware-fault-injection oriented handling the software product as a black box. Fault injection technique can be used to support robustness testing objectives.

Nevertheless, fault injection is recognized as a key element in the assessment and validation of critical software systems, as it is a practical approach to perform stress testing, i.e., raising conditions to trigger rarely executed software operations such as error handling and recovery. Fault injection is used to assess the FDIR mechanisms of any software system. Fault injection can be seen as a special case of testing, with faults becoming the main input in addition to its outputs. Fault injection is quite effective to spot the "interesting" faults, i.e., software faults that have a high probability of escape test case design. It is this inherent ability to trigger unexpected system behaviour that places fault injection technology as a definitive "should have" for achieving more confidence on accomplishment of dependability and safety requirements of critical software.

Fault injection should be used in a more comprehensive way where the fault models to be used address both hardware and software faults.

Hardware faults or software faults can be supported by three fault injection techniques:

1. Use of hardware injection support is a unique way to assess the containment and error propagation into low-level (software directly controlling hardware) critical software, and to achieve evident confidence that dependability and safety requirements are fulfilled.

2. Use of compile time software injection support, e.g. by use of mutation or fault seeding techniques. These two white box techniques are done with intrusion, which can be a disadvantage.

   These two techniques are performed by inserting or changing the original source code. They evaluate the software product from two different perspectives: the first one intended to locate faults and the second one intended to evaluate the effect of single changes to the original code. These techniques as modifying the source code, are to complement black box testing, and should be carefully used since in the insertion or change of the original code, other non-intended faults can be introduced. To be effective, these two techniques need good automated tools and significant amount of human analyst time and good insight of the software.

3. Use of run-time software injection support, e.g. by modifying the content of the memory, register, etc.

The use of fault injection is based on skilled personnel and good automation tools, which can be a disadvantage.

6.4.5.2.2 Stress testing

Stress or avalanche testing are black box testing, intended to burden the test object with an exceptionally high workload in order to show that the test object stands normal workloads easily. Under these test conditions the time behaviour of the test object can be evaluated. The influence of load changes can be observed. The correct dimension of internal buffers or dynamic variables, stacks, etc. can be checked. There are a variety of test conditions which can be applied for avalanche stress testing. Some of these test conditions are:

a. if working in a polling mode then the test object gets much more input changes per time unit as under normal conditions;

b. if working on demands then the number of demands per time unit to the test object is increased beyond normal conditions;

c. if the size of a database plays an important role then it is increased beyond normal conditions;

d. influential devices are tuned to their maximum speed or lowest speed respectively;

e. for the extreme cases, all influential factors, as far as is possible, are put to the boundary conditions at the same time.

Under these test conditions the time behaviour of the test object can be evaluated. The influence of load changes can be observed (these tests are sometimes called volume testing). Throughput analysis is considered part of this king of stress tests.

Requirements for the usage of resources such as CPU time, storage space and memory can be subject of these resource stress tests. The best way to verify these kinds of requirements is to allocate these resources and no more, so that a failure occurs if a resource is exhausted. If this is not suitable (e.g. it is not always possible to specify the maximum size of a particular file), alternative approaches are to:

a. use a system monitoring tool to collect statistics on resource consumption;

b. check directories for file space used. The correct dimension of internal buffers or dynamic variables, stacks, etc. can be checked.

As part of the main advantages of stress testing is that it is often the only method a) to determine that certain kind of systems are robust when maximum numbers of users are using the system, at fastest rate possible (e.g., transaction processing); and b) to identify that contingency actions planned when more than maximum allowable numbers of users attempt to use system, when volume is greater than allowable amount, etc.

One of the disadvantages is that is requires large resources.

6.4.5.2.3 Input partitioning

This test technique is intended to test the software using a minimum of test data. The test data is obtained by selecting the partitions of the input domain to exercise the software.

This testing strategy is based on the equivalence relation of the inputs, which determines a partition of the input domain (orthogonal groups- FTA can help to select inputs where to focus on robustness issues).

Test cases are selected covering all the partitions previously specified. At least one test case is taken from each equivalence class.

There are two basic possibilities for input partitioning which are:

a. equivalence classes derived from the specification – the interpretation of the specification can be either input orientated, for example the values selected are treated in the same way, or output orientated, for example the set of values lead to the same functional result.

b. equivalence classes derived from the internal structure of the program – the equivalence class results are determined from static analysis of the program, for example the set of values leading to the same path being executed.

Equivalence classes can be defined according to the following conditions:

a. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.

b. If an input condition requires a specific value, then one valid and two invalid equivalence classes are defined.

c. If an input condition specifies a member of a set, then one valid and one invalid equivalence class are defined.

d. If an input condition is Boolean, then one valid and one invalid equivalence class are defined.

Test cases from equivalence partitioning are complemented with test cases from Boundary Value Analysis and, applied in conjunction with other test practices to ensure the specified test coverage. Equivalence partitioning used to test dependability and safety of software focuses the test cases on the limit values or 'out of bound' values of each class identified.

The advantage of Test Cases from Equivalence Partitioning is to analyse that program behaves correctly for any class of input by selecting a representative value, reducing the total number of test cases that are developed.

No significant disadvantages are identified in the application of Test Cases from Equivalence Partitioning.

6.4.5.2.4 Boundary Value testing

The purpose is to provide test cases to detect and remove faults occurring at parameter limits or boundaries. The input domain of the program is divided into a number of input classes. The Test Cases should cover the boundaries and extremes of the classes. The tests check that the boundaries of the input domain of the specification coincide with those in the program.

Test cases from boundary value testing are complement with test cases from equivalence partitioning and applied in conjunction with other test practices to ensure the specified test coverage.

The use of the value zero, in a direct as well as in an indirect translation, is often error-prone and demands special attention:

a. zero divisor;

b. blank ASCII characters;

c. empty stack or element list;

d. full matrix;

e. zero table entry;

f. maximum or minimum values;

The boundaries for input have a direct correspondence to the boundaries for the output range. Test cases should be written to force the output to its limited values. Consider also if it is possible to specify a test case which causes the output to exceed the specification boundary values. Extreme values depend on the structure of the data.

If the output is a sequence of data, for example a printed table, special attention should be paid to the first and last elements and to lists containing none, 1 and 2 elements.

Examples of types of fault detected with the use of this technique:

a. Calculation faults.

b. Array size.

c. Null pointers.

d. Loop iterations

The advantages of the test cases definition from the boundary value testing, is that it analyses that the program behaves correctly for any permissible input or output

No significant disadvantages can be highlighted to the test cases definition from the boundary value testing in itself, but for programs with many types of input, all combinations of input cannot be tested

and therefore problems resulting from unexpected relationships between input types cannot be identified.

Table 6-2 summarizes the suitability of each of the above testing techniques to cover the different testing objectives described at the beginning of this annex.

Table 6-2: Relation between the testing objectives and the testing strategies

Testing techniques	Test objective		
	Interface	Robustness	Performance
Fault injection		X	
Stress testing		X	
Equivalence partitioning	X		
Boundary value testing	X		

The marked cells of the table mean that the testing technique suits best to achieve the respective testing objective. This does not deny that other techniques can contribute to achieve the other objectives.

6.4.6 Real Time Testing

Testing that real time software meets its performance criteria is always a difficult issue.

In these types of system it is essential to show during testing that the latency of the outputs from the software are always within the specification no matter what the input conditions that are applied. The primary methods that may be used to do this are as follows:

a. Logging

The real-time software should log an event that can be examined after the tests have been conducted. This log will show when the software has exceeded its specification or when it is within a margin of exceeding its specification. The relevant tests can be conducted and the afterwards the logs will show how the software has performed.

The logs can be obtained from measurements taken from the own computer internal hardware timers.

It can also be based on information provided by an external probe (JTAG, LICE/SIEL), which is a much less intrusive technique.

b. Monitoring Outputs

External test equipment can be used to determine the exact latency between input and output of the software, or in the case of hard real time, the position of outputs in the processing cycle. Output monitors could be as simple as a PC running with its NIC in promiscuous mode running an Ethernet monitor such as Ethereal. Alternatively they may be dedicated test equipment such as a logic analyser or 1553 bus monitor.

c. Emulation

A more cumbersome way of determining performance would be to use a Processor Emulator to replace the main processor. Facilities within the emulator may then be used to determine latency and performance without instrumenting the software. The problem with this approach is that the emulator can usually not be used in the later phases of integration and acceptance testing. Additionally, some emulators are accurate from the functional point of view, but not representative of the timing behaviour of the software.

The recommended approach would be to use a combination of methods during all phases of testing. Logs and outputs would then show if the software exceeds specification during different functional testing conditions as well as under full load and performance testing –in a non-intrusive way.

Design of load testing for real-time systems requires additional analysis to normal load testing. Realistic typical functional worst-case scenario should be elaborated and tested in order exercise the system under realistic worst-case constraints. However it may be simply impossible or impractical to enforce the worst-case condition at hardware and software level for a measurement-based observation to take place. Not only does the amount of inputs and quantity of data need to be represented with a test, but also the timing of this data should be considered. Individual loads of each task/process are also depending on some parameters (mode, configuration). Failure conditions of input systems and devices and the timing tolerance may lead to a race condition or timing issue on the software. This needs to be explored during the performance testing campaign by designing tests that provide different input timing constraints.

Real time testing cannot be exhaustive. It needs to be complemented with schedulability analysis based on WCET. Real time software testing is necessary to increase the confidence and provide real individual loads figures (to feed the real time analysis). It should provide the evidence of comfortable margins w.r.t. schedulability analysis which is too much theoretical/pessimistic.

6.5 Autocode

6.5.1 Relation to the Standard

Automatic code generation is addressed in the ECSS-E-ST-40C Standard clause 5.3.2.4 about the software development plan (and therefore also in Annex O the software development plan DRD), as well as in 5.4.2.2 for the Technical Specification requirements related to autocode.

However, the use of autocode techniques impacts all the development life cycle and is expected to be reflected in many places during the project development.

6.5.2 Introduction

Automatic code generation is primarily assumed (due to the current experience in projects) in the scope of a functional system design that has defined a set of subsystems (like AOCS, thermal control or power management) that can be subject to modelling and autocode. The subsystem team is in charge of the functional aspect of the subsystem, using a functional model. This team produces a high level model representing the expected behaviour of the subsystem level component. Then it is refined together with software team to become autocodable.

The functional model contributes to the subsystem RB, complemented with a textual RB covering the parts which are not addressed by the functional model (e.g. data handling, FDIR, non-functional aspects). The functional model is actually part of the software, as it carries information on technical specification, architecture, and design. As such, it follows some software related rules such as modelling standards, structural coverage measure, code generation process, etc.

The functional test suite used to validate the model contributes to software validation.

The software team is in charge of the actual autocode generation process, of the autocode integration with the rest of the software and the software validation

Another case of use of autocode is when the software team decides to use a modelling autocodable language instead of (or in addition to) a classical coding language like C or Ada. This is fully internal to the software development process.

6.5.3 Subsystem and software relationship around autocode

6.5.3.1 Introduction

For the software implementation of the subsystem that develops a functional model, the software development plan is expected to address the relationship between subsystem and software in case of automatic code generation, and in particular to assess:

a. are changes to the subsystem requirements specification function needed,

b. who is responsible for model specification,

c. who checks that the model matches the specification,

d. who verifies and validates the models,

e. who checks that automatically generated code matches (i) the model and (ii) the specification,

f. any associated questions.

The following section introduces how a given organisation may adapt its existing role model in order to provide effective execution of projects involving automatic code generation. The experience given here comes primarily from the development of space application software, in the highly specialized area of AOCS/GNC; therefore this particular subsystem is mentioned.

6.5.3.2 Roles

6.5.3.2.1 Subsystem modelling team responsibility:

This team is responsible for defining/designing the modes and the control laws for realisation of a given mission. This team verifies that the proposed design (for AOCS/GNC) complies with the required performance levels. To this end it develops models and performs simulations proving that the model is robust and meets the requirements. To ensure the completeness of the simulation tests with respect to the designed model, the coverage of the model during tests is checked.

6.5.3.2.2 Software team responsibility:

The software team is in charge of checking that the model can be auto-coded. It proposes the global architecture of models and any standards, rules or modifications allowing the code generation. It is in charge of model configuration (via the code generator tool) and performs code generation. It checks that modelling rules are respected.

The software team verifies that the code generation has not introduced errors at the functional level, by running reference (regression) test cases (provided by the subsystem team). The software team is in charge of integrating the generated software in the rest of software, and generating a single executable file. It verifies the behaviour of the software in the real target environment.

6.5.3.3 Autocoding process

The process is iterative. This means that it is applicable several times for each subsystem iteration. The subsystem iterations are defined e.g. for a subset of functions that can be easily validated independently. For example, an AOCS iteration is associated with an AOCS mode. In the following, the subsystem of choice is the AOCS, but could be any functional chain subsystem expressed with models having autocode capability (e.g. thermal, power)

Step 1: AOCS preliminary design is defined during phase B, which is ended by the **AOCS Preliminary Design Review** that baseline the AOCS design, and therefore the subsystem functional models.

Elements of the software AOCS Requirements Baseline are produced, including non-functional requirements, requirements which are not part of the model, and potentially the model.

Step 2: At the beginning of the C/D phase the AOCS and software teams can work separately. The software team works first on the global needs analysis and the definition of the software architecture. The AOCS team refines the preliminary design and enriches the AOCS models according to the modelling standards defined for auto-coding. It is supported by the software team to help the definition of the software architecture and interface of the AOCS models with the rest of the software. The AOCS team performs the functional validation of the AOCS model. The AOCS "Critical Design Review" is held once all increments are designed and after performance tests.

In this step, there is a co-engineering phase between the AOCS and software teams to evaluate the auto-code-ability of the AOCS model and to complete it with respect to non AOCS requirements such as operability, FDIR, and software constraints.

Step 3: The validation of the AOCS model can lead to modifications that may induce to restart the previous step

Step 4: The software team develops the software that cannot be generated from the models, in a standalone way. The classical unit test, integration tests, validation tests and code coverage are performed.

Step 5a: This is the genuine autocode approach that maximizes the gain of autocode, but does not insure full code coverage. It is acceptable for software that is not of criticality category A. It is also acceptable for software of criticality category A if the code generator is qualified. Once the AOCS needs are validated using simulations at model level (e.g. Simulink), the unit test concept is applied to the model. Model units (elements of the model) are tested. The model structural coverage is measured (tools exist for the most common modelling environments, e.g. Simulink). Then the code is generated automatically and can now proceed to integration.

Step 5b: This (alternative) approach insures full code coverage. It is mandatory for software of criticality category A and B, except if the code generator is qualified at the same level as the generated code (see ECSS-Q-ST-80C 6.2.8.3). In this approach, the code is automatically generated from the functionally validated model. It is then unit tested in a classical way, and its structural coverage starts to be measured.

Step 6: The complete software, whether manually or automatically generated, is integrated and validated. If automatable, the model tests (i.e. unit tests, validation) may be rerun on the modelling environment with the generated code in order to check the code generator. This is not necessary if the code generator is qualified.

After code generation of the software, the subsystem team continues to use the model to execute the performance tests of the sub-system.

The software maintenance is performed at model level in order to ensure the consistency of the modification.

This is illustrated in Figure 6-1.

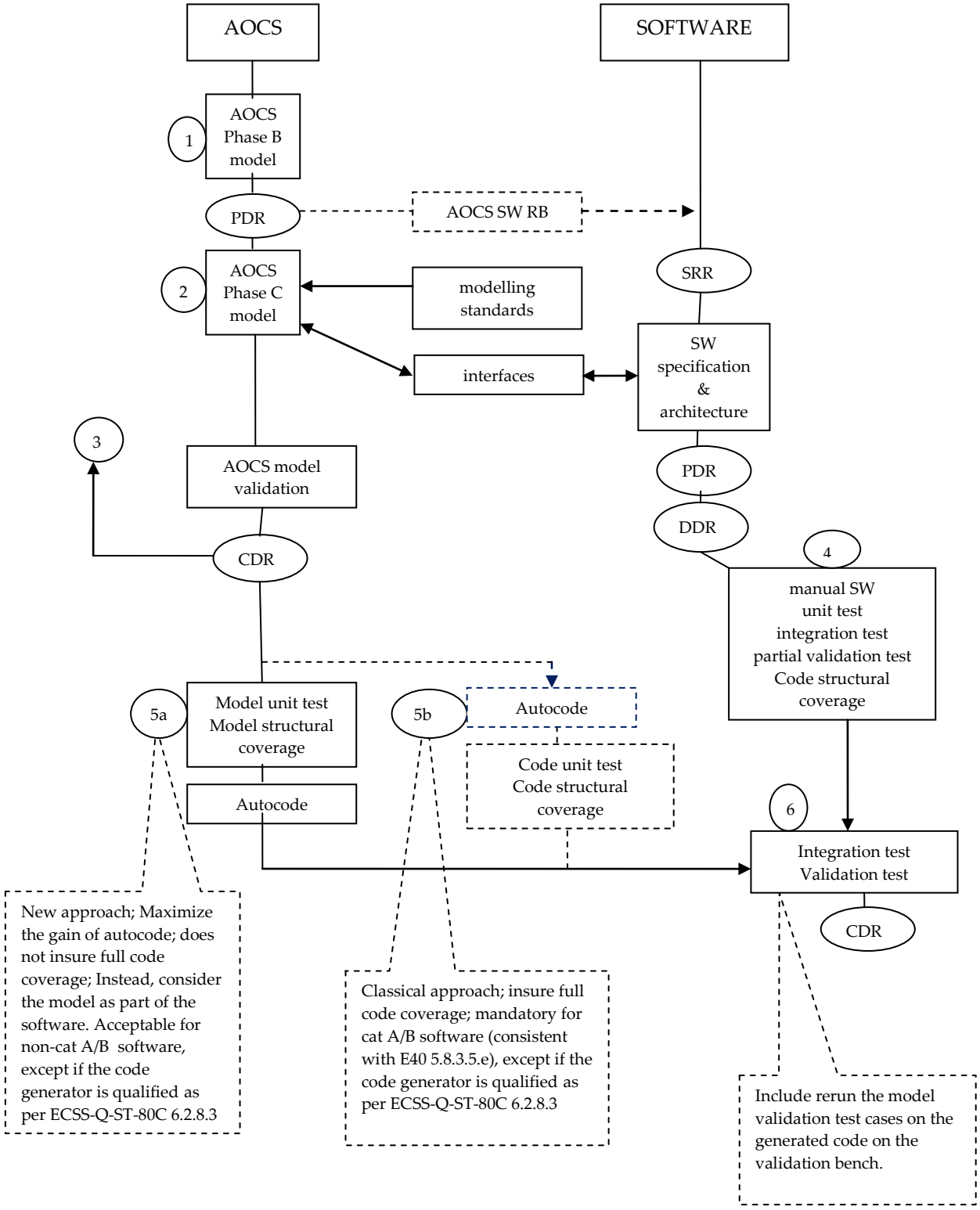


Figure 6-1: The autocoding process

6.5.3.4 Impact on the software reviews

6.5.3.4.1 Software SRR (Software System Requirement Review):

This is the classical subsystem Software Requirements Review. At software SRR, the RB part of the autocoded subsystem is available. This includes the information necessary for the software architect to design, in its architecture, the container that will receive the autocode (frequency, execution time, more generally all non-functional requirements affecting the software architecture).

6.5.3.4.2 Software PDR (Preliminary Design Review):

The goal of the review is to come to common agreement on the requirements and architecture. The objects under review are models, interfaces, and other textual requirements (for parts of the SW subsystem that cannot be modelled).

The models represent all the expected functional requirements (for those requirements that can be modelled) and all external interfaces of the subsystem that can be modelled. The model defines the subsystem SW architecture, i.e. what the main components are and how they communicate. At this stage the model is not necessarily ready to generate product code.

At the PDR the part of code to be manually implemented is identified and the handling of the interfaces between manual and generated code is defined. The selection of the tools for automatic code generation and the coding standards are baselined. The modelling standards have been agreed with Subsystem modelling team before their PDR. Simulation on the model may have been performed but is not subject to review.

6.5.3.4.3 Software CDR (Critical Design Review):

The goal is to demonstrate that software is ready for qualification.

The model used to generate code must be fully completed. The model has been validated in the modelling environment. Generated units have been successfully integrated and SW validation performed,

6.5.4 From subsystem model to autocoded model

6.5.4.1 Traceability

Software requirements dedicated to the autocoded model part, are defined by a combination of textual requirements and models (e.g. Simulink), instead of textual requirements with pseudo code defining the algorithms. Some modelling tools offer traceability capabilities allowing establishing the links between upper level requirements and the subsystem model, or from subsystem model elements to tests.

6.5.4.2 Configuration Management

As a part of the overall software, a model used as input of the autocode needs to be managed under configuration, meaning the full model managed by the modelling tool, initialisation files, attributes and libraries.

6.5.4.3 Modelling standard

Autocoded models need to be developed according to a modelling standard, to ensure that the model answers the quality characteristics required for the software (e.g. maintainability, robustness, reliability). Modelling standard should thus include naming conventions, rules to ensure that code generation is possible, some appropriate defensive programming principles, and rules to improve the readability of the models.

The model needs also to be built in such a way to satisfy the overall model/software interfaces and also to ensure that the software code is properly generated (i.e. partitioned in accordance with the software architecture). The action performed by the software when a numerical error or robustness check is executed is agreed between the Subsystem modelling and SW teams.

6.5.4.4 Verification and Validation

The overall assumption of the autocoding approach is that the code is automatically generated in a consistent way with the associated model, any manual verification or tests at source code level (e.g. unit or integration tests) can be skipped provided that the conditions set by the ECSS-Q-ST-80C 6.2.8 requirements are fulfilled. This should be considered in a similar way than between a source code and a binary code automatically built with a compiler.

As it is the case today for compilers w.r.t. object code for non-criticality A software, it is envisaged that code generators will reach the same maturity level, such that engineering activities made today on code will be made on the model, relying on the validation activities for the verification of the actual binary generated by "model compilers". Hence, verification and tests made at source code level in a manual approach are simply transferred at model level in the autocoding approach, in order reducing the errors potentially introduced at model level. For criticality A software, such verifications and tests at autocode level are applicable, in a similar way to ECSS-E-ST-40C clause 5.8.3.5.e, except if the code generator is itself qualified.

Moreover, according to the capabilities of the used tools, the autocoding approach also favours the early verification of requirements thanks to the simulation at model level which permits the early detection of errors and deficits in the "specification".

The validation of the software product (i.e. against upper level requirements) would still need to be performed to ensure the functional correctness of the software. It generally consists in the adaptation of the tests already run at subsystem model level, these tests being preferably specified by the team specifying the model for efficiency reasons.

In some case, it may be interesting to replace the model by the generated software, in order to re-run tests, made at model level, on the final software in the subsystem simulation environment. In this way the subsystem team contributes to the validation of the final software product against the specification.

6.5.4.5 Modelling Coverage

According to ECSS-E-ST-40C, the coverage measure is necessary. With the autocoding approach, the measure is performed at modelling level instead of source code, through the various verifications and tests performed by the subsystem modelling team.

7
Real-time software

7.1 Relation to the Standard

The Standard addresses real-time software in particular:

a. in clause 5.4.3.3 where the computational model must be selected,

b. in clause 5.5.2.5 and 5.5.2.7 where the detailed design of real-time software is specified,

c. in clause 5.8.3.11 where the real-time properties are verified through schedulability analysis,

The margin definition is addressed in 5.3.8.1, the margin computation in 5.3.8.2, the budget management in 5.8.3.12.

The following section introduces the notions of margins, computational model and schedulability analysis that are called for in the Standard in order to make clear what is expected when applying it.

NOTE Partitioned systems are not addressed in this handbook for the definition of margins, budgets, computational model, real-time design, and schedulability analysis. They are just described in 7.4.3.1.5.

7.2 Software technical budget and margin philosophy definition

7.2.1 Introduction

The software product margins apply in particular to:

a. load and real-time figures including:

   1. CPU load,

   2. deadline for a processing with constraints (e.g. period, rendez-vous, reaction constraint, timing accuracy constraint);

b. memory capacity;

c. numerical accuracy;

d. external (including hardware) interface/event synchronisation and timing.

Margins definitions are agreed between the customer and the supplier. Guidance on margin definition is given in this section. Margin definition and management are likely to raise more attention for embedded or real-time software, as it is more difficult to extend the hardware resources, or to access to the system once launched. Hardware on the ground can be scaled up vertically and horizontally.

However, ground software may face other concerns at system level, e.g. in terms of ergonomics (for instance response time), network load, the definition of the environment such as the version of the hardware and software, which are not addressed here.

In order to allow the Supplier to manage the margins in an appropriate way (e.g. design flexibility), they should be defined by the Customer in the context of their need, e.g. for already known growth capability, or for risk reduction.

In particular, the margin philosophy should be specified in such a way that it does not over-specify the design, unless the customer has explicit requirements regarding the software architecture, scheduling and interfacing.

Criteria for the selection of all the margins can be:

a. The processor module capability. The processing budget management and reporting presume that basic hardware choices, such as processor type, clock frequency, wait-states have been made and clearly described.

b. Equipment, communication and performance aspects, e.g. buses, protocols, acceptable errors, capacity bus usage by other sources.

c. System design which is derived in timing constraints (e.g. constraints on state transitions, especially when recovery from a faulty state is concerned)

d. Expected future changes in the requirements baseline (e.g. due to iterations, requirements on reprogramming of the system during operational use, required budget for temporary copies of software images)

e. Expected phasing/versioning of the software development and its impact on produced code and CPU load

f. Accuracy aspects, such as conversion to/from analogue signals, and accuracy of timing signals

g. Allocation of responsibilities regarding the characteristics of Service Access Point interfaces, if not handled by one contractual party.

Reuse of existing software and software maturity (level of uncertainty on the figures) are also criteria in particular during the early phases where the figures are estimated

7.2.2 Load and real-time

7.2.2.1 Schedulability analysis

For load and real-time characteristics, information about schedulability analysis is given in section 7.5.2. Schedulability analysis demonstrates different real-time constraints such that all tasks meet their deadline.

The schedulability analysis uses (among others) the following two inputs:

a. the task worst case execution time (estimated in the beginning of the life cycle, measured later on)

b. the occurrence of events. For example, all events may be released at the same time, or phasing could be introduced between events.

7.2.2.2 Theoretical worst case and operational scenario

The worst case execution time of a function, task or software is widely understood as the *theoretical worst case*, either measured, or estimated or derived from code analysis techniques.

Alternatively, the notion of worst case can be refined functionally-wise, by selecting (realistic) operational scenarios (nominal or degraded, per mission phases or modes) that maximizes the execution time of each function (*operational worst case*).

The former (*theoretical worst case)* is more pessimistic than the latter (*operational worst case),* which is based on operational scenario.

The theoretical worst case is established on the basis of a discussion between customer (or system level) and supplier (or software level). The context of the occurrence of this worst case can be more or less critical, more or less realistic, more or less likely to happen. For example, CPU utilisation may go over 100 % in case of simultaneous occurrence of failures, or maximum frequency of all telemetry, which are generally not the system level assumptions.

In this context, the use of operational scenarios to derive an operational worst case and realistic operational sequences are a mitigation of the theoretical worst case (see Figure 7-1), as the difference between the two times could be substantial. It is nevertheless interesting to know the theoretical worst case, because it allows understanding the conditions when it can happen and to relate to the operational context.

NOTE ECSS-E-ST-40C clause 5.2.3.2 requires the customer to define representative scenarios.

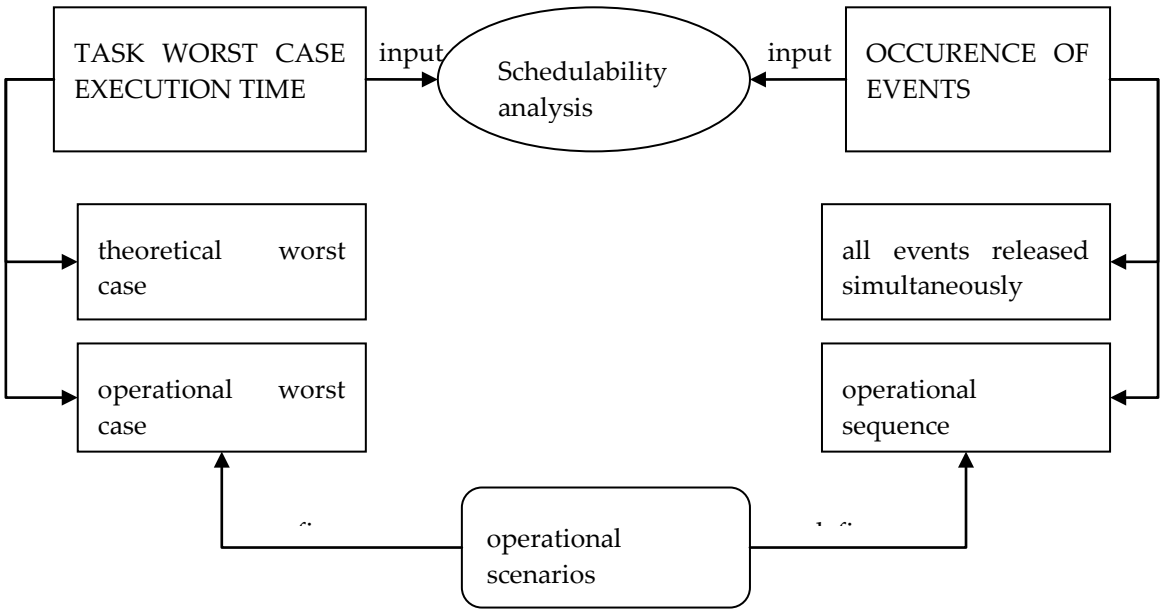


Figure 7-1: Mitigation of theoretical worst case with operational scenarios.

7.2.2.3 Margins

Margins can therefore be proposed:

a. per task in isolation, by provision of a margin of execution time (*margin_WCET*)

b. per task in context, by provision of a growth capacity before it reaches its deadline (*margin_slack*),

c. globally, by provision of a growth capacity before the CPU saturation (*margin_utilisation*).

The margins can be based on either theoretical worst case or operational worst case.

1) Margins can be defined on the deadline of a particular task taken in isolation using its Worst Case Execution Time:

margin_WCET(i) = [deadline(i) - WCET(i)] / deadline(i)

This can be used to force a margin on the estimation of the WCET if the customer needs to have confidence on the estimated execution time, even if WCETs are already pessimistic in nature. In addition, this does not account for the context of execution of the task (i.e. the interference by higher-priority tasks). The only useful use of this margin is to understand the time window during which the task of interest completes its execution before its relative deadline occurs.

Typical values of this margin should be maximum 10% at PDR where the WCET is estimated, and then they should not be required later on.

The *margin_WCET* is generally based on the (theoretical) worst case execution time, because the focus is typically put on the execution time of the particular algorithm implemented in the task that is assessed against the envisaged deadline.

2) To take into account the context of execution of the task, a margin on the slack time can be defined. The slack time is based on the response time of the task, i.e. the time after which it actually completes its execution in the worst-case scenario, taking into account all the potential pre-emptions by higher-priority tasks, the blocking time due to the access to protected objects, the execution time of interrupt handlers, and interference by mechanisms of the RTOS / real-time kernel.

response_time(i) = WCET(i) + interference(i) + blocking_time(i)

slack_time(i) = deadline(i) - response_time(i)

margin_slack(i) = slack_time(i) / deadline(i)

Typical value of this margin very much depends on the nature of the task. Customer may wish to ensure computation margin to a particular computational-intensive control loop with short deadline, or instead to minimize margin to a background task. It is likely difficult to define a global margin such as 10% for all tasks. This could be a target at PDR, but should be refined later on.

The *margin_slack* uses the response time computed by the schedulability analysis, therefore based on the theoretical worst case (execution time and events). If the result is found pessimistic, the reasons are analysed, and the operational scenarios are used to assess the extent to which the operational assumptions are valid.

3) The theoretical CPU utilisation given by the schedulability analysis (in the absence of memory cache) is:

$$U = \Sigma_i \frac{WCET(i)}{T(i)}$$

where :

WCET(i) is the worst case execution time of the task i (estimated in the early phase, measured at the end of the project)

T(i) is the period of the task i if this task is periodic, or the Minimum Inter-Arrival Time (MIAT) of the event triggering the task if it is sporadic.

The CPU utilisation of a schedulable system is less than or equal to 1 (this is a necessary condition, but not sufficient to declare the system as schedulable).

The margin is:

margin_utilisation = 1 - U expressed in percentage

As said in ECSS-E-ST-40C, typical values of this margin are 50% [PDR], 35% [DDR/TRR] and 25% [CDR and after].

The margin_utilisation uses execution time. Therefore, it is based on the theoretical worst case execution time. If the result is found pessimistic, the reasons are analysed, and the operational scenarios are used to assess the extent to which the operational assumptions are valid.

In addition, it is handy to assess the margin_utilisation by measuring the execution time of the background task (the task in which the software is idle). However, the time during which it is measured must be longer than the largest period or Mean Inter Arrival Time (MIAT) of the system.

7.2.3 Memory capacity

For memory capacity, the margin is defined as follow:

margin_memory = free size / total size

As said in ECSS-E-ST-40C, typical values of this margin are 50% [PDR], 40% [DDR/TRR], 35% [CDR] and 25% [QR, AR].

7.2.4 Numerical Accuracy

For numerical accuracy, today's processing systems use standardised integer and floating point processors (e.g. IEEE-Std-754-1985) in relation to specific data path widths, therefore the accuracy of both integer and floating-point calculations is known by these provisions.

The customer should clearly define all accuracy requirements and constraints, at least at system level (e.g. pointing requirements). The software supplier should derive the related numerical accuracy requirements, and manage them as per the ECSS-Q-ST-80C requirements 7.1.7.

When the algorithm is modelled at system level (e.g. Attitude and Orbit Control), the floating point representation and the mathematical library used on the target processor is introduced in the system modelling tools.

7.2.5 Interface timing budget

For interface timing budget, as a minimum, the following elements should be considered:

a. latency of interrogated subsystems (incl. error)

b. latency of (bus) interfaces (incl. collision, bus error)

c. interface reconfiguration

d. hardware driver characteristics

e. software driver characteristics

f. This interface timing budget is used to compute the WCET (see section 7.2.2).

7.3 Technical budget and margins computation

Budget reporting should provide breakdown for the customer to assess the risk areas and possible sources of non-compliance with the budget.

If margins are not met at a given milestone, the supplier justifies it and proposes workaround solution, or corrective actions, or deviations with respect to the RB, which permit to come back within the margins.

7.3.1 Load and real-time

The verification of load and real-time margins is typically based on a schedulability analysis (see 5.5.1 for detailed information).

The verification of the CPU load budget intends:

a. to verify that the definition of the allocated resources and margin to be reached is unambiguous.

b. at PDR to verify that the budget presented is based on the first level of design and it is the result of elementary value for each processing to be executed on the system. The elementary value of each processing is justified by analogy or by measurement performed on a prototype. To verify that the concurrent processing taken into account to define the budget and so the margin is a realistic worst case.

c. at CDR to verify that the budget presented is based on measurement of each processing. To verify that the budget is refined by introduction of interrupt and kernel impacts. To verify that the concurrent processing taken into account to define the budget and so the margin is a realistic worst case.

d. at QR to verify that the budget presented is based on a measurement collected during a dimensioning scenarios and performed on a representative environment. To verify that the budget is close to the budget computed with elementary values, else to verify that the disparity is commented/justified.

The verification of the deadlines and timing constraints:

a. at PDR (and to be confirmed w.r.t. the DDR), performs schedulability analysis with the elementary value of each processing and according to the timing requirements identified in the RB or TS. Each timing requirement which cannot verify the complete schedulability analysis is verified by a partial or adapted schedulability analysis focused on it. If a schedulability coefficient for an activity is superior to the specified margins, it would be commented – justifying the actions in risk reduction. This first status is used as software design input

b. at CDR, refines schedulability analysis with the elementary measurement of each processing and according to the timing requirements identified in the RB or TS, according to the design constraints (response time, blocking time, phasing, etc…). Each timing constraint which cannot be verified by the complete schedulability analysis is verified by a partial or adapted schedulability analysis focused on it. If a schedulability coefficient for an activity is superior to the specified margins, it would be commented – justifying the actions in risk reduction.

c. at QR, refines schedulability analysis with the consolidation of the elementary measurement of each processing based on measurement performed during validation campaign on a representative environment. If a schedulability coefficient for an activity is superior to the specified margins, it is highlighted and guaranteed as a maximum limit.

7.3.2 Memory margins

The memory is based on non-volatile memory (e.g. ROM, PROM, EEPROM, hard disk) and volatile memory (e.g. RAM, DRAM).

The non-volatile memory budget for dedicated software is based on its static analysis (code and constant size).

The volatile memory budget for dedicated software can be based on its static analysis (data and stack size) if no dynamic memory usage is implemented, or dynamic analysis in others cases. Generally, for flight software there is no dynamic memory mechanism implementation (state of the art).

If specific constraints exist on each of these areas, the associated margins are identified (e.g. paging, MMU, virtual memory).

The verification of the memory budget intends:

a. at PDR, to verify that the budget presented is based on representative criteria from comparable projects already completed (by analogy approach). In case of new and complex function, the budget is based on a prototype or by analogy / reuse.

b. at CDR and following milestones, to verify that the budget presented is based on compiled object files.

In addition, the verification of the volatile memory budget intends, at CDR and following milestones, to verify that the stack allocation has been analysed based on the call graph and pre-emption phenomena and that a margin for it is specified.

In case of dynamic memory mechanism implementation, the verification includes also:

a. that the scenario used to define the necessary memory budget is justified and correctly dimensioned.

b. that the budget is the result of an analytic model and/or the result of a measurement on a specific test.

This memory budget includes aspects related to the compilation environment, the assumed code expansion factors (and the references for those numbers, including the project's definition for Line of Code used), the effects of compiler settings (e.g. for optimisation, runtime checks, setting of the operating system kernel, runtime system and/or Board Support Package, …).

NOTE The budgets related to sizing of stacks, buffers, FIFOs are verified by the Supplier (in particular for robustness), but are potentially not subject to margin specification by the customer, except if future expansion is expected. The Customer should anyhow check if related requirements are necessary (e.g. for payload data management).

7.3.3 Numerical accuracy budget management

The supplier analyses for any (critical) function, that the selected (floating-point) format fits the required accuracy of the algorithms, including the effects of repetitive calculations and calculations with extremely small or deviating numbers.

The verification of the accuracy is usually based on the fact that the results of a reference scenario on the system modelling tool (running the algorithms that requires accuracy), and on the software target, remains in an acceptable difference. This can be assessed using absolute and/or relative differences; interval arithmetic can also help to determine what an acceptable difference is.

For analogue-to-digital and digital-to-analogue interfaces, the choice of the converter must fit the required signal resolution. If the software does not use the full resolution provided by the hardware, then the software supplier should analyse if the software resolution fits the requirements for the signal processing.

The verification intends:

a. at PDR, to verify that the (numerical) accuracy requirements are clearly specified in the RB and TS

b. at CDR, to verify that the numerical accuracy requirement is reached according to the data type used and propagation effects. This verification is based on analytic analysis or a measurement performed on a representative numerical environment.

7.3.4 Interface timing budget management

The customer is responsible for establishing interface budgets. He may involve the supplier to optimise the interface requirements.

Interfaces determine, to a large degree, the system architecture. Hence the interfaces are usually defined in the early project phases. However, in those phases it is often not possible to assess all timing restrictions. It is therefore good practice to design the interface system such that margins and blocking are not critical to the design, in order to avoid costly changes to the architecture.

Changes to the architecture should not be excluded *a priori*. However, the customer, being responsible for the specified budgets, should show awareness of the potentially dramatic impact of architectural changes, by carefully re-assessing the specified budget values in those cases.

Interface timing budget management should be performed per subsystem interface, i.e. each interfacing system should be considered separately. Subsequently, the overall budget should be determined.

The budget per interface should include at least:

a. Nominal latency (no collision, error), per subsystem

b. Maximum latency (collision, no error), per subsystem

c. Error and reconfiguration handling, including uncertainty

d. Blocking behaviour, and reference to the tasks affected by the blocking

The budget reporting is the supplier's task. It should provide sufficient insight and evidence for the customer to assess the feasibility of both the interface design and the overall architecture, in support of the computer processing budget management. This interface timing budget is used to compute the WCET (see section 7.2.2).

7.4 Selection of a computational model for real-time software

7.4.1 Introduction

Real-time software executes within strict constraints on response time, meaning that it is subject to "real-time constraints". Such real-time constraints are operational deadlines on the response to be produced by the software processing on occurrence of an event.

For real-time software the correctness of some of its operations depends not only upon their logical correctness, but also upon the time at and in which they are performed. Real-time systems, as well as

their deadlines, are often classified by the consequence of missing a deadline: : a system is considered hard real-time when missing a deadline incurs a system failure. It is considered a soft real-time system when a late response degrades the system's quality of service yet still delivering an acceptable outcome

Real-time software is therefore a software characterized by some timing properties (e. g. execution period, duration, periodic or sporadic event occurrence), and that satisfies the timing requirements (e.g., deadlines). The applicable timing properties should be captured and properly traced through the design and code. More specifically, these properties are treated in the dynamic design that presents the information required to understand the flow of information, the flow of processing and related timing issues in the software.

The time requirements are verified through static analysis and by testing. Testing the real-time behaviour of software requires in fact specialized techniques – and since exhaustive testing is impossible in real-scale systems, there is usually no guarantee that the time requirements are eventually verified by test means. Static analysis, which seeks, computes and considers worst-case bounds, is instead capable of providing absolute guarantees if the conditions for the analysis are met..

In order for static analysis to be usable, a computational (and therefore analysable) model of the real-time software is identified at design time so that the timing requirements can be verified analytically. Timing requirements can be analytically verified for deterministic or predictable execution models, i.e. proven, through schedulability analysis. The elements of the computational model should in this case be captured by the software dynamic design.

Real-time software is usually intended for use in embedded/flight software, but it can also apply to other software, such as simulators (e.g. in case the software simulator interface with real hardware, or when the simulator time needs to be real-time accurate) and generally speaking to ground software (e.g. minimum load of data to be treated per second, maximum handling time for a telecommand or telemetry, …). In the latter cases, the real-time aspects to consider are the same as for embedded real-time software, and the process is similar. However, depending on the software environment and the CPU resources, meeting the timing requirements necessitates potentially less engineering efforts than for flight software.

7.4.2 Recommended Terminology

This section provides recommendations for term definitions.

Term	Recommended definition
Absolute deadline	Absolute time at which the deadline of task's job occurs, which is computed by adding the task's relative deadline to the absolute release time.
Activation	Task activation, which is the release of a task job.
Aperiodic task	A task whose execution is triggered by an activation event which can repeat at irregular intervals. Aperiodic tasks are either idle waiting for the next activation event, or they are executable after being triggered by an event occurrence.
Asynchronous	Independent of execution. For example, asynchronous events are events that occur independently of the execution of the application.

Term	Recommended definition
Asynchronous I/O (communication, data exchange)	An I/O (communication, data exchange) operation that does not cause the task requesting the I/O (communication, data exchange) to be blocked waiting for the end of the call. This implies that the task and the I/O (communication, data exchange) operation may be running concurrently.
Blocking	State of a task caused by the mechanisms that enforce mutual exclusion of resource use by multiple tasks.
Deadlock	Situation whereby two or more tasks are prevented from proceeding, each waiting for a needed shared resource to be released
Determinism	Property of a computational model such that the response time of all tasks is statically known and always the same.
Elapsed time	The time measured on a clock between two events (thus including the execution time spent by other tasks in case pre-emption occurs)
Execution time	Task's execution time is the time its execution takes, i.e. the time spent by the CPU executing the task.
Execution time monitoring	ET monitoring, operating system feature allowing monitoring the task execution-time
Hard deadline	Deadline of a hard real-time task.
Hard real-time	Denoting an entity (e.g. a task, a system) where missing a deadline is a system failure.
Interference	Interference of a task T_i with higher priority task T_j occurs when T_i is pre-empted by T_j.
Jitter	Variation in time of an event, e.g. task release jitter, task response time jitter, sampling jitter (variation in the input instant) and input-output jitter (variation in the delay from input to output).
Job	A single complete execution of one task activation, triggered by an occurrence of activation event. I.e. a task is a potentially infinite sequence of jobs arriving at either irregular or regular intervals.
Latency	Delay between the activation of a task and its start of execution.

Term	Recommended definition
Minimum inter-arrival time	The worst-case minimum time between two activation events of a sporadic task.
Missed deadline	Situation when activity is not completed at the time when it should have been finished. A missed deadline does not necessarily provoke a task overrun.
Period	If not used in a different context, period refers to the task's period, i.e. the time between two activations of a periodic task. Sporadic tasks are often analysed the same way as periodic ones and the term "period" is used in the sense of the minimum inter-arrival time.
Periodic task	A task whose execution is repeated based on a fixed period of time. Periodic tasks are either idle waiting for the next period or they are executable after being triggered at pre-defined regular intervals by a timer.
Predictability	Property of a computational model such that the response time of all tasks is guaranteed to always be between a statically known best case and a statically known worst case.
Pre-emption	act of temporarily interrupting (suspending) a task being executed, without requiring its cooperation, and with the intention of resuming the task at a later time.
Priority	Precedence; for tasks, attribute allowing the selection of the task eligible for execution by the scheduler
Process	Run-time entity recognised by an operating system; an address space with one or more threads executing within that address space, and the required system resources for those threads.
Race condition	Condition where update to shared resources depends on the interleaving of task accessing them. This can be avoided by non-pre-emptible mutually-exclusive access to shared resources
Relative deadline	The deadline for the completion of a task's job relative to the release instant of the task. If not explicitly stated, all deadlines are to be intended as relative deadlines.
Release	Task release, i.e. the moment when task's activation/job starts after an arrival of its activation event. After its release the task is *released*.

Term	Recommended definition
Response time	The worst-case elapsed time between the release of a task (in fact, of any of its jobs) and its subsequent completion. Response time is used in response time analysis in the way that if all tasks in a system have their response times lower than or equal to their deadlines, then the system is said to be schedulable. In case system overheads are ignored, response time of a task is equal of the sum of its worst-case execution time, interference and blocking times.
Soft real-time	Denoting an entity (e.g. a task, a system) where missing a deadline degrades the result, thereby degrading the system's quality of service.
Sporadic task	An aperiodic task for which a minimum inter-arrival time between two activation events is defined and guaranteed.
Task	Run-time entity sequentially executing a program written in a programming language, having a single thread of control. An execution entity to which a processor is allocated.
Task overrun	Situation when task's execution time exceeds the task period, i.e. when the task is activated while its previous activation has not finished yet
Thread	Run-time entity recognised by an operating system; a single flow of control within a process. Operating system thread usually maps to a single task.
Utilisation	Referring to processor utilisation, unless used in a different context. The proportion of processor time used by tasks, measured over representative time. Task's utilisation refers to the proportion of the processor time used by that task during its activation period, i.e., the time interval from its release to its completion. The time over which utilisation is measured should be the major cycle of the system, i.e. the least common multiple of all periods, unless specified otherwise.
Worst-case execution time	The longest execution time of a sequential code under worst possible circumstances. Usually referred to as WCET.

7.4.3 Computational model

7.4.3.1 Overview

A Computational model defines abstract system entities representing computations, communication and synchronisation between computations, and their temporal and concurrency properties, in a manner ensuring that a system built using them is by definition amenable to static analysis. Typically, the entities representing computations are tasks and the computational model describes what the tasks can and cannot do, it defines types of tasks, how they communicate and synchronize. A computational model provides abstractions and constraints which make it possible to analyse the real-time software and verify its time properties, i.e. to perform schedulability analysis.

ECSS-E-ST-40C requires that the computational model be selected and defined as a part of the software architectural design. The detailed design as well as the code must be compliant to the selected computational model.

The computational models most commonly used for real-time software executing on single-core processor are:

7.4.3.1.1 Cyclic Executive

Cyclic executive, based on the time-triggered activation scheme, allows the creation of systems based on complete temporal determinism. Cyclic executive is characterised by a fixed major cycle composed of an integral number of minor cycles with fixed duration. While major cycle represents the repetition period of the whole software system, also known as the hyper-period, minor cycles represent time slots in which periodic functions are executed. In order to accommodate functions with different periods, these periods are a multiple of the minor cycle, so that a function has several minor cycles dedicated for its execution. Long-duration jobs are split over several minor cycles – during which time resources in use may remain locked.

Important features of this approach are as follows:

a. No actual tasks physically exist at runtime; minor cycles are just sequences of procedure calls, which all complete within the boundary of the minor cycle.

b. Procedures typically share a common address space, as they are executed in the scope of a single RTOS task.

NOTE In a single-address-space RTOS, such as RTEMS, this is not an intrinsic property of this computational model. For instance in RTEMS all tasks always share the same memory address space no matter which computational model is used. In an RTOS which supports processes that have their own (virtual) address space, all tasks in the cyclic executive would execute in the scope of the same process, within the same address space, executed by the same execution thread. This is the usual and easiest implementation, but other implementations are possible too, e.g. cyclic executive triggering tasks executed by a pool of execution threads

c. Shared resources do not need to be protected, as there is no concurrency in this system.

d. The duration of the minor cycle is a comparatively complex function of the procedure duration, of the task deadlines and of the task periods. Finding a minor cycle that satisfies all of the applicable constraints may require splitting procedures and placing them in minor cycles where there is slack time available.

A periodic clock (that signals the boundary of minor cycles) is usually used to check that the periodic functions are accomplished within their minor cycles (this mechanism is also known as watchdog). Sporadic tasks and interrupt handlers could be accommodated by dedicating particular minor cycles to handling asynchronous events.

NOTE Assuming that the interrupt/asynchronous event frequency is lower than the frequency of the minor cycle dedicated to their handling

Interrupt handlers only raise a flag which is detected in the dedicated minor cycle to perform the bulk of the asynchronous event handler.

A special case of using a cyclic executive combined with the use of multiple RTOS processes and pre-emption by asynchronous tasks is described in Section [Pre-emptive System without Cyclic Pre-emptions].

The cyclic executive approach could be implemented in virtually any programming language.

7.4.3.1.2 Pre-emptive System

A Pre-emptive system requires an underlying RTOS, and uses RTOS threads to represent tasks at run time. The system is designed to be able to decide which task is eligible to run. There are multiple scheduling techniques possible, with either fixed priority scheduling, dynamic priority scheduling or scheduling without priority.

In the fixed priority scheduling approach, tasks have fixed priorities assigned. The priorities determine the highest execution eligibility: the higher the priority the higher the eligibility.

NOTE In different RTOS implementations different priority ranges are used and also both “the higher the number the lower the priority” and the higher the number the higher the priority” exist.

The most common fixed priority scheduling techniques are:

The Rate Monotonic Scheduling (RMS): RMS applies to periodic tasks with implicit deadlines (i.e., equal to period and therefore resulting in a fairly restrictive system model) and allocates the highest priority to the shortest period.

The Deadline Monotonic Scheduling (DMS): DMS applies to periodic and sporadic tasks with constrained deadlines (i.e., possibly smaller than period and therefore resulting is a slightly less restrictive system model) and allocates the highest priority to the shortest deadline.

In the dynamic priority scheduling approach, the priority of a tasks is not fixed, but re-evaluated whenever a scheduling event occurs (i.e., on every task release event). The most common dynamic priority scheduling technique is:

Earliest Deadline First Scheduling (EDF): EDF applies to periodic and sporadic tasks and assigns the highest priority to the ready task with the earliest deadline.

Finally, Round-robin (RR) is a scheduling technique which assigns time slices to each task in equal portions and in circular order, handling all tasks without priority (it falls back to cyclic executive [Cyclic executive chapter] in case it is used with a single task).

The pre-emptive computational model would typically define two types of tasks: periodic and sporadic, i.e. tasks released upon an occurrence of an event for which the minimum inter-arrival time is known. In principle, tasks are in one of the following states:

a. Runnable;

b. Suspended waiting for a clock-produced release event (for periodic tasks);

c. Suspended waiting for a software- or hardware-produced release event (for sporadic tasks);

d. Blocked on a shared resource protected by a mutual exclusion mechanism.

The scheduler based on fixed priorities guarantees that, any point in time, amongst of all runnable tasks, the one with the highest priority is executing. Pre-emptive systems guarantee that the moment a new task with a higher priority becomes runnable, the task currently executing is pre-empted and the higher priority task is run. Priorities are fixed, i.e. known at design time after an initial assessment, statically assigned to tasks and never changed at runtime. The only time they are temporarily changed for short periods of time is during RTOS application of one of the synchronisation protocols to avoid priority inversion.

NOTE The priority assignment could change at later stages of software development, during coding and testing, when actual measurements are available and replace execution time estimates.

A pre-emptive system based on fixed priority scheduling could be implemented in virtually any programming language, but some concurrent programming languages have FPS already built-in in them.

7.4.3.1.3 Pre-emptive System without Cyclic Pre-emption

A special flavour of a computational model based on a pre-emptive system has been successfully used in real-time software. Periodic tasks all have the same period (equivalent to the main system operation cycle) and are activated by phasing the activation time in a way which guarantees that they never pre-empt one another. The activations are designed based on availability of data coming from TC and TM packets.

In a way, this model has some of the advantages of a cyclic executive, for instance there is no resource contention between periodic tasks. However, periodic tasks could be pre-empted by occurrence of asynchronous events, handled either by asynchronous tasks or interrupt handler routines. Pre-emption, resource protection and blocking of these asynchronous events are taken into account.

The presented flavour of a pre-emptive system could be implemented in virtually any programming language.

7.4.3.1.4 Ravenscar Computational Model

The Ravenscar computational model [Ravenscar] is a special case of a pre-emptive system with tighter restrictions on tasking and synchronisation, and with a close link to the corresponding programming model.

The Ravenscar Computational Model (RCM) is directly linked with the definition of the Ada Ravenscar Profile that was originally defined for use in high-integrity systems. The Ravenscar Profile is a subset of the Ada programming language, eliminating all possibly non-predictable language constructs. The RCM is an example of a computational model with an event-triggered workload model.

NOTE Thus the Ravenscar computational model is directly linked and comes from the corresponding Ada Ravenscar Profile programming model.

Tasks have a single source of activation events, in the form of either a periodic clock or a software-generated event (such as a software invocation or a hardware interrupt from a source other than the system clock). In the latter case, the event is sporadic, and there is a minimum inter-arrival time between two subsequent releases. Aperiodic events are not allowed by the RCM, but aperiodic

activities can be opportunistically served by RCM servers, for example following the sporadic server model. Tasks have a single suspension point per activation – either waiting for a time clock event or a sporadic event. Synchronous communication between tasks (i.e. task rendezvous) is disallowed; instead tasks use data-oriented asynchronous communication mediated by shared resources equipped with the Priority Ceiling Protocol (see Section [Shared Resources]). The RCM allows task declaration and creation only at the system start-up time – dynamic declaration of tasks is not permitted.

In general, a RCM-based system can be implemented in virtually any programming language. However, only the Ada Language compilers are equipped with static analysis tools for the pragma Ravenscar construct, which makes it possible to check that the code is compliant to the RCM rules.

7.4.3.1.5 Partitioned System

Investigations are on-going of the execution environment that provides partitions to support logical separation of software building blocks, with different integrity levels, as opposed to the physical separation by running such building blocks on separate hardware. In avionics, the concept of partitioning is used by Integrated Modular Avionics (IMA) which introduces logical partitioning techniques to avionics computing systems [IMA]. This concept has been standardised in the ARINC 653 civil avionics standard, which defines an interface between the application software and operating system services which enforce temporal and spatial separation between applications, in different partitions in order to ensure their safety [ARINC-P1].

There are three different aspects to logical partitioning: Temporal partitioning, spatial partitioning and I/O partitioning. Temporal partitioning allows controlling how much processor time the operating system allocates to a partition, i.e. a set of tasks execution within the partition. Spatial partitioning ensures that memory spaces of execution tasks belonging to different partitions are strictly separated and no intentional or unintentional illegal memory accesses to a different partition are possible. I/O partitioning ensures that partitions can share access to the I/O stack.

The scheduling of a partitioned system is the hierarchical composition of a high level scheduling scheme, e.g. Round-Robin (which only schedules black-box partitions) and different scheduling schemes, e.g. fixed-priority pre-emptive scheduling (which only schedules tasks within partitions). In the time partitioning model according to ARINC-653, partitions are statically scheduled so that at any point in time it is known which partition is executing.

Other partition scheduling alternatives are possible, such as more dynamic scheduling based on a CPU-time budget per sliding time window allocated to each partition [Davis05].

Partitioned systems are not addressed in this handbook

7.4.3.2 Access to Hardware Resources

Real-time software commonly interacts with hardware resources. Different schemes to access hardware resources exist and they must be consistent with the computational model. Therefore the selection of the computational model or the description of the associated programming model, will specify and capture the way in which hardware resources are accessed. Here is a non-exhaustive list of possible approaches:

a. Asynchronous I/O with polling: A task initiates a hardware operation and later polls the data result at predefined time (e.g. with bounded latency).

b. Asynchronous I/O with a handler task: A task initiates a hardware operation and the data result is handled by a dedicated task (handler).

c. Synchronous I/O: A task initiates a hardware operation and stays blocked waiting for data result. It stays blocked until the data result is available or a predefined timeout expires. This

mode of hardware interaction is not directly amenable to scheduling analysis, which causes undesirable inaccuracy in the predicted results.

7.4.3.3 Criteria for Computational Model selection

For the purpose of the verification of timing requirements, the selected computational model needs to permit the design and implementation of software that is statically analysable.

A basic choice is the selection of a sequential model versus concurrent model, taking into account the system characteristics (e.g. access to hardware resources, time-triggered or event-driven architecture).

Sequential systems (i.e., those scheduled by a Cyclic Executive) have the major advantage to be deterministic, since each action takes place at a fixed time within a given schedule of frames. The scheduling analysis is minimal: it is simply necessary to ensure that each action is completed within its allocated time (known as frame or minor cycle, the duration of which is determined as a function of the duration, period and deadline of all the software activities). However, sequential design models of inherently concurrent problems require potentially large design effort, since the temporal behaviour needs to be considered in detail during the design of each activity: an activity is artificially split into small steps that are mapped and executed in different frames. These models are fragile in the face of even small changes in the code as well as in the timing attributes, since the execution schedule may need to be completely rebuilt. This fragility increases with the size of the application (in number of concurrent activities). Sequential design models are also less optimised and therefore consume more CPU.

Concurrent systems (e.g. pre-emptive Executive, Ravenscar) offer simpler and more natural designs, more stable against changes in detailed design and code, since each concurrent activity can be designed in isolation. However, there is a penalty in that the system is no longer deterministic, and more sophisticated analysis techniques are in general required to ensure predictability of behaviour and the fulfilment of timing requirements. Concurrent models are indeed more complex to analyse, because the run-time kernel will switch threads according to the scheduling algorithm. To solve the problem, *analytical models* have been developed to establish design rules and mathematical equations by which the real-time behaviour of a collection of concurrent threads can be analysed and predicted. The two best known theories, Rate Monotonic Scheduling (RMS) and Deadline Monotonic Scheduling (DMS) apply to *pre-emptive fixed priority based* scheduling models, and provide rules for the assignment of priorities to threads, rather than having the designer chooses priorities according to some informal heuristic. RMS applies to cyclic threads and allocates higher priority to tasks with shorter periods. DMS applies to sporadic threads as well, and allocates higher priority to tasks with shorter deadlines. Other scheduling algorithms, such as Earliest Deadline First (EDF), require *pre-emptive dynamic priority based* models. In this case, the priority of the tasks is not fixed, but re-evaluated whenever a scheduling event occurs (a task finishes or a new task is ready to execute) and therefore the execution is neither deterministic nor predictable.

The penalty in runtime system size and time overheads for concurrent systems is depending on the selected computational model. This penalty is of major importance for systems with limited resources (e.g. flight software). It is minimised for some computational models such as the high-integrity Ravenscar profile.

In addition, for pre-emptive dynamic priority based models, any run-time kernel implementing the model cannot be exhaustively tested. Therefore, the schedulability analysis should to be complemented with dedicated tests with the run-time kernel.

Concurrent systems using suitable computational models and supported by conforming run-time systems are now appropriate for all forms of spacecraft reactive software. The need for analysability however constrains the choice of the concurrent execution model. The more complex the application, the greater cost advantage with the use of a concurrent model over a sequential model.

7.5 Schedulability analysis for real-time software

7.5.1 Overview

In order to perform analytical verification, it is ensured that the software design and later on implementation comply with the chosen computational model.

Schedulability analysis verifies whether, given the specified timing attributes of tasks, they all meet their respective deadlines. It can also permit to quantify other time requirements on the software (e.g. jitters). Refer to section 6.5 (schedulability analysis) for further information.

Other properties of the system (possibly postulated by schedulability analysis) such as shared resource protections, re-entrancy mechanism, absence of deadlocks… are also verified by analysis (e.g. abstract interpretation technique). The same techniques can also be used to verify non-real-time properties such as sizing of task stack, sizing of message queues.

The computational model of software does not necessarily capture all the aspects of the dynamic design – but the dynamic design should cover all the elements of the computational model. The more stringent the time properties to be verified, the more detailed the computational model.

As for other verification activities, analytical verification covers the entire software lifecycle. Specific analysis techniques can be performed at the technical specification level, other at design or code level; they can thus only be applied when the needed input becomes available. For what concerns the schedulability analysis, a preliminary analysis is provided at PDR, based on the identified software tasks in the design and on estimated WCET. The schedulability analysis is then refined throughout the development until QR, with measured WCET and taking into account the actual implementation of the dynamic aspects of the software.

7.5.2 Schedulability Analysis

Schedulability Analysis is a formal static analysis method associated to a computational model, based on estimations/measurement of the WCET of individual task activities that enables to ensure the schedulability of a set of tasks, in relation with their behaviour, interactions and timing properties, and to verify the fulfilment of their timing requirements.

7.5.2.1 Needed Input

7.5.2.1.1 Overview

This section summarises all necessary inputs to perform schedulability analysis. Minor deviations from this list are indeed possible; however in general the constituents identified below form a necessary pre-cursor to any reasonable schedulability analysis.

Some of the information listed in this section should already be a part of the Technical Budget of the SVR, often in a separate document called Software Timing and Sizing Budget document (STSB) or Software Budget Report (SBR).

The following information should be provided for the purpose of performing the schedulability analysis:

a. Selection of the computational model: usually a reference to the Software SDD where the computational model is described as a part of the architectural design.

b. Selection of the associated programming model: A reference to the Software SDD or coding standard definition where the use of programming language constructs and library calls compliant to the selected computational model is specified (e.g. restricted system calls or options of the OS that can be used for tasks, semaphores…).

c. Major cycle, or hyper-period of the system, representing the repetition period of the whole software system. It is the least common multiple of all task periods.

NOTE In flight software, this is usually linked to the driving frequency of the AOCS control loop. It could also be linked with the main communication bus.

d. Tasks: the list of task in the software, as per design/implementation, together with their time properties (cyclic/aperiodic/asynchronous, frequency).

e. Priorities: Based on the used RTOS, a priority range is given and a method of assigning these priorities to tasks should be described (e.g. deadline monotonic assignment implies giving the higher priority the shorter the deadline is). It is very important to clearly state what the priority numbering is, i.e. whether the higher the number the higher the priority (e.g. in 1 – 255 priority range the priority of 255 is the highest), or vice versa (e.g. priority 1 is the highest). Priorities can be assigned by scheduling analysis – the preferred option when possible.

f. Deadlines: Specification of which kind of deadlines the tasks in the system have. It is assumed that most deadlines are hard real-time but it should be specified which deadlines are hard and which deadlines are not. In addition, it should be specified what the relation of task's deadlines is with respect to their periods. Typically the task's deadline is less or equal to its period, but some cases it could be always equal to the period or even greater than the period.

g. Task worst case execution time measurement.

h. Measurement methods: All measurement methods used. More specifically:

i. The method of measuring worst-case execution times (see Section 7.5.2.3.2 for more details);

j. Method of measuring system overheads (see Section 7.5.2.1.2 for more details)

k. Clear indication of whether CPU execution time or elapsed time is measured;

l. Detailed description of measurement setup: software and hardware tools used, whether the measurements are performed on simulator or real target hardware;

m. Precision of the measurements.

n. Status of measurements: estimates vs. measured numbers. A clear way of distinguishing estimates from real measurements and evolution by milestones/time.

7.5.2.1.2 System Overheads

The equations of the adopted schedulability analysis theory should account for the overheads of the synchronisation, mutual exclusion and communication system calls, and indicate how they are taken into account by the schedulability analysis (e.g. in the calculation of the blocking times):

1. Pre-emption (task context switch)
2. Access to semaphores, mutexes, task locks, interrupt locks
3. Handling of message queues (fixed execution time of operation + execution time dependent on the length of the message)
4. Interrupt latency (interrupt context switch)

7.5.2.1.3 Shared resources

In order to calculate all blocking times in the system, the schedulability analysis should indicate how access is granted to a shared resource (I/O, hardware resource, shared memory location…):

At design level:

a. Avoid shared resource by granting access through a single task (e.g. using incoming message queue of requests)

b. Ensure by task scheduling/task priority scheme that two tasks never access the resource at the same time

At code level:

a. mutual exclusion using protected objects (lock), including locking protocol (such as FIFO, priority inheritance or immediate priority ceiling protocols)

b. interrupt lock (disable task pre-emption and interrupts)

c. task lock (disable task pre-emption)

The schedulability analysis should also indicate whether it is allowed for a task to access two shared resources simultaneously (e.g. nested critical sections) – which is one of the preconditions potentially leading to deadlock.

The interrupt handler execution times should also be provided.

Then, a table is provided with each resources represented by a column and tasks in rows ordered by their priority, so that the cell(i,j) indicates how much time task i uses resource j. Also, the resource access types are indicated as well as possible locking protocol, so that it is possible to deduce worst-case blocking time for all tasks.

7.5.2.2 Description of the analysis method

The schedulability analysis should provide information on the algorithm or analysis equation used.

The schedulability analysis is not limited to considering the (worst-case) *execution time* of a task, but also considers the time during which the task is pre-empted by a higher priority task (*interference time*) and the time during which the task is waiting for access to a shared resource (*blocking time*). The blocking time can be reduced by applying *priority-ceiling* protocols. The priority of the resource is raised to the highest priority of any of the potential caller of the resource. The alternative solution, consisting in masking all the interrupts (run to completion), is not recommended, as it affects all the tasks in the system, not only the potential callers of the resource. Consequently, it alters also the accuracy of the schedulability analysis.

The addition of the execution, interference and blocking times gives the task *response time*. The schedulability analysis involves solving the fixed-point response time equations by iteration. To avoid that there is never any choice of which task is eligible to run at any given instant; each task has a *unique priority*. Then, the verification of the schedulability consists in checking that each task response time is less than its deadline. The schedulability analysis may propose as an output the priority allocation to the tasks. Software tools are available to support schedulability analysis.

Moreover, the schedulability analysis could be performed in different increments (typically refinements along the lifecycle, but possibly only in case the margins become too low), starting from a simple, idealised model that is then enriched with additional elements, e.g.:

a. In a first iteration, the model could take into account all cyclic tasks, with their frequency, priority and worst case execution time.

b. A subsequent iteration could enrich the model with the interrupts and asynchronous tasks.

c. The model issued from a third iteration could consider the shared resources and critical sections, with their worst-case blocking time and their relationships with the tasks.

7.5.2.3 Quality of Schedulability Analysis Input

7.5.2.3.1 Introduction

This section provides important guidelines to ensure that the input data (in particular all time measurements) are adequate and precise for the purpose of schedulability analysis.

7.5.2.3.2 How to measure the Worst Case Execution Time

When measuring the individual function/task execution times (i.e. without pre-emption), the following aspects should be considered:

1. Heavy stress conditions operational scenarios: Scenarios are defined which stress the system by the heaviest load possible. These scenarios could be different for different modes and should take into account all parts of the system.

2. Execution paths: Even within a heavy stress scenario, that not only the worst case data loads should be considered, but also all sorts of conditions which may influence the worst case execution path (e.g. time for error processing such as retry of sending message).

While aspect (1) requires good knowledge of the system and its operational and performance requirements, aspect (2) looks at possible worst case execution paths and maps them onto realistic execution scenarios. For this a number of tools are available. Some of them are based on WCET estimates: they thus tend to provide very pessimistic numbers, which are however safe bounds as opposed to measurements by observation which may not be safe (i.e., can be inferior to the actual worst case). Other tools combine execution path analysis with on-target analysis and perform profiling of the system at the granularity of function calls and execution blocks: these, combined with good system knowledge, could provide results with high degree of confidence.

7.5.2.3.3 Mitigation of Pessimism

A recurrent problem of schedulability analysis is that it may incur excessive pessimism by incurring overly pessimistic execution scenarios. A rigorous application of the theory in the static worst-case scenario of execution may lead to declaring a system infeasible, when, instead, extensive observation of real operation does not show violation of timing requirements. Another indirect consequence of this situation consists in the creation of systems which have low CPU utilisation.

Several sources of pessimism for the analysis results are be distinguished:

a. Requirements: High-level requirements may introduce strong timing constraints for rare events (for instance, upload of telecommand at very high rate for patch upload). This brings an inherent pessimism to the system, and the analysis method can hardly do anything about it.

b. Architectural and detailed design: often, lack of detail in the architecture of detailed design of the software leads to over-estimate the worst case execution time of functions and tasks.

c. Analysis method: mainstream response time analysis for single-node systems is pessimistic by definition as it considers the worst case behaviour for all real-time aspects of the software, even if the probability is very low (or even zero):

    1. Sporadic tasks period is set as the minimum inter-arrival time, although this minimum inter-arrival time can occur very infrequently;

2. Worst-case condition for mutual exclusion locks (based on chosen locking protocol, e.g. priority ceiling priority inheritance).

NOTE Priority ceiling is superior to basic priority inheritance in terms of reducing worst-case blocking time as well as in avoiding deadlock

d. Selection of worst-case scenarios. The specification of this information should be provided at least to discriminate the different execution scenarios in the various operational modes of the software. The description of the scenario should be both convenient (easy to use) and accurate (analysable).

There are several ways to mitigate this pessimism:

a. Increase the information provided by the designer.

b. In general there is a strict relation between the pessimism of the analysis and the quantity and quality of the information provided at design time (so that it can be used directly or to calculate the input for an analysis tool). The less the information, the higher the pessimism.

c. The designer should have the means to specify beneficial information at least for the main cases which (s)he deems meaningful and important. The specification of this information should be provided at least to discriminate the different execution scenarios in the various operational modes of the software.

d. As a means to specify this kind of information and the different analysis scenario, it is possible to use a set of sequence diagrams, by which the designer may complement a component description with a series of scenarios of execution to analyse. Additionally, in some execution scenarios, the designer may know that a certain sporadic operation will not be triggered. The specification of the scenario on which the overall schedulability analysis is performed would allow specifying this information (otherwise the sporadic operation could be considered as triggered at each time instant multiple of its MIAT, which is the global worst-case scenario in which operational modes are ignored).

e. Split the global worst-case scenario in several statically defined worst-case scenarios.

It consists in investigating how it is possible to split the single monolithic worst-case scenario of execution of the software, in which every single task is assumed its worst-case behaviour irrespective of the execution mode of the others, in multiple yet safe worst-case scenarios. This is especially the case for systems with distinct modes of operation, each of which calls for a distinct analysis scenario.

If static analysis is performed on a single global scenario, each task will contribute to the worst case with its worst case behaviour among all the possible operational modes. Therefore the “global” scenario is a software condition that can possibly never manifest itself during execution, as it is simply the join of separate (and logically unrelated) worst-case behaviours of the tasks.

Instead, if distinct worst-case scenarios are provided, for example one for each operational mode, each scenario would be generated with the worst-case contribution of each task specific to the operational mode under analysis (and not with the worst-case for every operational scenario).

Each scenario would permit to perform a dedicated schedulability analysis, which would provide the worst-case results for each task in the operational mode under analysis; these results are calculated statically and are safe. The set of results of all the generated scenarios is the proof of feasibility of the system w.r.t. timing requirements (the system if schedulable if it is schedulable in each worst-case scenario).

Finally, the provisions for scenario-based analysis should also include means to specify the telecommands received from ground (in particular, which telecommands and how many), as software execution of various components is certainly triggered or conditioned by them.

7.5.2.3.4 Computer cache

NOTE This section addresses mainly instruction cache, as few studies have addressed data cache effect for the moment.

Caches are very effective at speeding up memory accesses in the average case, but at the cost of more variable execution time, which complicates the task of obtaining trustworthy predictions on the timing behaviour of the program. The considerable variability in the execution time of a program caused by the use of caches may hamper, perhaps to a different extent, both WCET analysis and measurement, which are essential inputs to schedulability analysis. The execution time of the next instruction depends on the current contents of the cache and on the address of the current instruction itself: in other words, it depends on the addresses the program recently referenced in its execution path.

Trying to account for the cache impact in a scenario-based measurement approach is very challenging and cumbersome. Moreover, the impact may change from test to operation, incurring insidiously different execution times. Hence, high-integrity systems cannot rely on an uninformed use of caches, for their uncontrolled effects on the timing behaviour may invalidate the results of timing analysis and, in turn, annihilate the trustworthiness of schedulability analysis. It is worth noting that caches are not the only accelerator feature that may threaten WCET analysis: the whole system architecture may have a large impact on both the feasibility and the results of WCET analysis. In particular, complex pipelines and virtual addressing can drastically increase the complexity of static analysis.

Hybrid measurement-based analysis aims to avoid optimism in the WCET estimation by measuring the WCET of small program fragments, (typically basic blocks or groups thereof) and then combining them using static analysis techniques to produce a WCET estimate. Therefore, loop bounds can be added a posteriori to the measurement stage and the WCETs of program fragments can be triggered in different test runs. However, hybrid measurement-based methods suffer from some pitfalls. Path pessimism, much like for static analysis, infeasible paths and overestimation of loop bounds, all lead to excessive (i.e., not tight) WCET estimate.

Cache-aware schedulability analysis accounts for cache-related pre-emption delays, which are an estimate of the time required by a pre-empted task to restore its cache state at resumption, in the response time of individual tasks.

A distinct approach to increasing predictability proactively seeks more predictable cache behaviour by restricting the way cache normally operates (by way of cache locking and partitioning). Cache locking techniques consist in exploiting hardware support to explicitly control the cache contents by loading information in the cache and then temporarily or permanently disabling the cache replacement policy. Since all or some of the cache contents are locked in the cache, the cache behaviour and accesses would become fairly predictable. Cache partitioning techniques aim at increasing cache predictability by partitioning the cache in a way that a cache segment can be reserved for a specific task or group thereof. The cache interference due to interactions between distinct tasks is thus removed; on the other hand, the usable cache size for each task is reduced, which may increase cache conflicts within each task.

It is also worth noting that predictable code patterns and coding styles may help improve WCET analysis, by bounding the sources of overestimation.

Cache-aware techniques should be integrated at different stages of the development process, to explicitly counter the cache (and timing) predictability problem. It should be a combined approach that leverages on the explicit control of the cache-related variability, and the collection of timing information as early as possible in the development process. Providing guarantees on the avoidance of

pathological cache behaviour, would allow the final timing analysis to confirm the assumptions made in the early stages of development.

The measurements of execution time taking into account cache should preferably be performed on the target hardware, as emulators may not be fully representative of real caches.

7.5.2.4 Schedulability Analysis Results

7.5.2.4.1 Major results

The conclusion of the schedulability analysis is whether the software is schedulable or not. The results of the schedulability analysis should be summarized in a single table providing all significant timing information for all tasks, possibly across all of its modes of execution. Such a table is usually filled by estimated valued at PDR. Over time, these estimates are replaced by real measurements so that at CDR/QR all the timing numbers in the table are actual measurements (except those explicitly produced by the schedulability analysis). These figures can then still evolve until QR (and sometimes AR) depending on the definition of the worst case scenarios (see section 7.5.2.3.3). Below is an example of such a table (see Figure 7-2).

Task Id (Name)	Subsystem	Priority	Type	Frequency	Period	Deadline	WCET	Max Blocking	Max Interference	Utilisation	Response Time	Deadline Margin	Worst case scenario description	Comments
				Hz	ms	ms	ms	ms	ms	%	ms	%		
Time Management	System Ctrl	10	Periodic	100	10	10	0.44	0.090	0.000	4.40%	0.53	94.70%		
1553 Bus Handler	BSW	20	Periodic	50	20	20	4.70	1,120	0.440	2.66%	6.26	68.70%	4.7 ms only once per second, 2.55 ms otherwise	Hence utilisation is 4.7 + 19*2.55/20
AOCS Main Loop	AOCS	25	Periodic	10	100	40	9.70	0.260	27,900	9.70%	37.86	5.35%		
TC Handler	Data Handling	61	Sporadic	5	200	100	6.22	1,120	47,300	3.11%	54.64	45.36%	5 TC per second	
Thermal Control	System Ctrl	63	Periodic	5	200	200	11.25	0.850	53,530	5.63%	65.62	67.19%		
OBCP	Data Handling	65	Periodic	2	500	200	68.40	0.850	78,020	13.68%	147.27	26.37%		
Housekeeping	Data Handling	66	Periodic	1	1000	400	40.40	0.850	31,820	4.04%	73.07	81.73%		
MTL	Data Handling	91	Sporadic	1	1000	400	24.30	0.160	271120	2.43%	295.58	26.11%		
System Log	Data Handling	97	Sporadic	1	1000	1000	126.90	0.160	569020	12.69%	696.08	30.39%		
Scrubbing	System Ctrl	99	Periodic	1	1000	1000	114.00	0.000	744,000	11.40%	858.00	14.20%		
Total										69.73%				

Figure 7-2: An example of a complete task table with all timing figures

The task (static) priority, type, frequency, period and deadline are static information.

NOTE As assigned by the scheduling algorithm assumed by the computational model of choice and not arbitrarily chosen by the user

The task worst-case execution time is either estimated or measured.

The maximum blocking time, maximum interference time, CPU utilisation, response time and deadline margin are values calculated by the schedulability analysis.

The task table should respect the following guidelines:

a. Tasks, represented by rows, are ordered in priority order (from the highest-priority to the lowest priority);

b. Measurement units should be indicated.

c. Highlight (e.g. by choosing different background colour) which of the numbers are measurements and which of them are estimates.

d. Worst-case scenario description is used for the description of the conditions under which the estimated/measured numbers where obtained.

e. It is possible to add a column to account for interrupt handler routines (e.g. by having a margin per activation or per time period). Alternatively, the interrupt handler routines can be modelled as tasks with top priority.

NOTE In this case, it is ensured that interrupt context switch overhead is used rather than task context switch overhead when performing the schedulability analysis

Highlight response times with a small margin comparing to their deadline. The software is schedulable when all the tasks have a response time not superior to their respective deadline.

7.5.2.4.2 Additional results

Interesting information can be deduced by comparison of the analysis of different worst-case scenarios, or even by parameterizing task WCET. This allows calculating a posteriori the worst-case work load. Useful time information can be derived for the system if required, such as the duration of a complete scrubbing of the memory, the processor power down time, the duration of a particular activity, the delay (response time) to handle one telecommand, the number of telecommands that can be handled per second.

In order to verify timing requirements for specific functions or events in the software, another possible approach is to refine the analysis by decomposing a task into smaller "sub-tasks". This technique makes also particularly sense when a periodic task has a WCET different for each of its activation (based on a lower frequency pattern, e.g. a 5 Hz task executing a particular function only once every 5 activations), as it would provide a more realistic (less pessimistic) schedulability analysis.

The schedulability analysis results can also be exploited to verify other timing requirements, such as maximum jitter for a particular task or rendez-vous constraints.

7.5.2.5 Schedulability Analysis Checklists

The following non-exhaustive checklists are provided to highlight schedulability analysis aspects to be covered by the reviews.

Table 7-1: Schedulability Analysis Checklists

Requirements checklist	SSS resource requirements	Verified
	1. Does the SSS include requirements on the computer hardware to be used?	
	2. Does the SSS include requirements on the total computer resource utilization per milestone, in particular processor capacity and memory capacity available for all the software items? This usually means CPU, EEPROM, PROM and RAM utilization.	
	3 Does the SSS include requirements on a specific real time operating system?	
	4. Does the SSS include requirements on other software items to be used or incorporated into the system, having impact on timing or concurrency?	
	SRS resource requirements	Verified
	1. Are there any software design requirements for the computational model?	
	2. Are there any software design requirements for the programming model?	

	ICD resource requirements	Verified
	1. Are the time constraints defined in the ICD?	
	2. Are the shared resource accesses defined in the ICD?	
Architectural design checklist (See note)	Computational model	Verified
	1. Has the computational model been selected and as a part of the architectural design?	
	2. Has the computational model been described in sufficient detail in the architectural design?	
Detailed design checklist	Computational model	Verified
	1. Is the dynamic design model compatible with the computational model selected during the software architectural design model?	
	2. Has it been ensured that all the methods utilized for different items of the same software are, from a dynamic standpoint, consistent among themselves and consistent with the selected computational model?	
	Programming model	Verified
	1. Has the programming model compliant to the selected computational model been defined?	
	2. Does the programming model enforce the compliance with the computational model?	
	3. Has a method to check compliance with the programming model been specified?	
	Detailed design of real-time software	Verified
	1. Have all timing and synchronization mechanisms been documented and justified?	
	2. Have all the design mutual exclusion mechanisms to manage access to the shared resources been documented and justified?	
	3. Has the use of dynamic allocation of resources been documented and justified?	
	4. Has protection been ensured against problems that can be induced by the use of dynamic allocation of resources, e.g. memory leaks?	
	5. Has it been verified that testing is feasible, by assessing that computational invariant properties and temporal properties are added within the design?	

NOTE: The architectural design is mapped to contemplate highlighting API for activities such as task activation, deactivation and deadline check.

Table 7-1: Schedulability Analysis Checklists (cont.)

Software timing and sizing budget document checklist	Basic Information	Verified
	1 Is a reference to the selected computational model provided?	
	2. Is a reference to the corresponding programming model provided?	
	3. Is the main period of the system specified?	
	4. Is tasks priority range defined?	
	5. Is priority ordering defined?	
	6. Are hard real-time and non-hard real-time tasks identified?	
	7. Are characteristics of deadlines w.r.t. to tasks periods specified?	
	8. Are measurement methods and precision of all figures presented in the STSB detailed?	
	9. Is every measurement clearly marked as either an estimate or a measured number?	
	10. Does the status of the measurements vs. estimates match the planned evolution by milestone?	
Code checklist	Basic Information	Verified
	1. Is the code compliant with the computational model?	
Testing checklist	Validation Test Specification	Verified
	1. Are the WCET scenarios defined?	

Annex A
Documentation Requirement List

Although the Annex A of the ECSS-E-ST-40C software standard is only informative, project feedback considers useful to complete the DRL table with the missing reviews such as SWRR, SQSR, DRR and TRR, TRB.

NOTE 1A document, which has been reviewed in the anticipated part of the review, will be reviewed again in this review only if it has been updated.

NOTE 2 Due to the nature of the SQSR, where only the reuse file is reviewed, it is not mentioned in a specific column in the table, but only in the row of the reuse file, as alternative to SWRR.

NOTE 3 For the TRR and the TRB, the documents to be provided are either the one against TS in the TRR/TRB before CDR, or against RB in the TRR/TRB before QR or AR.

Related file	**DRL item** **(e.g. Plan, document, file, report, form, matrix)**	**DRL item having a DRD**	**SWRR**	**DDR**	**TRR**	**TRB**
RB	Software system specification (SSS)	ECSS-E-ST-40 Annex B				
	Interface requirements document (IRD)	ECSS-E-ST-40 Annex C				
	Safety and dependability analysis results for lower level suppliers	-				
TS	Software requirements specification (SRS)	ECSS-E-ST-40 Annex D	✓			
	Software interface control document (ICD)	ECSS-E-ST-40 Annex E	✓	✓		
DDF	Software design document (SDD)	ECSS-E-ST-40 Annex F		✓	✓	
	Software configuration file (SCF)	ECSS-M-ST-40 Annex E		✓	✓	
	Software release document (SRelD)	ECSS-E-ST-40 Annex G			✓	✓
	Software user manual (SUM)	ECSS-E-ST-40 Annex H		✓		

Related file	DRL item (e.g. Plan, document, file, report, form, matrix)	DRL item having a DRD	SWRR	DDR	TRR	TRB
	Software source code and media labels	-			✓	
	Software product and media labels	-				
	Training material	-				
DJF	Software verification plan (SVerP)	ECSS-E-ST-40 Annex I			✓	
	Software validation plan (SValP)	ECSS-E-ST-40 Annex J			✓	
	Independent software verification & validation plan	-				
	Software integration test plan (SUITP)	ECSS-E-ST-40 Annex K		✓		
	Software unit test plan (SUITP)	ECSS-E-ST-40 Annex K		✓		
	Software validation specification (SVS) with respect to TS	ECSS-E-ST-40 Annex L			✓	
	Software validation specification (SVS) with respect to RB	ECSS-E-ST-40 Annex L			✓	
	Confirmation of readiness of test activities	-			✓	
	Approved test results					✓
	Acceptance test plan	-				
	Software unit test report	-			✓	
	Software integration test report	-			✓	
	Software validation report with respect to TS	-				✓
	Software validation report with respect to RB	-				✓
	Acceptance test report	-				✓
	Installation report	-				
	Software verification report (SVR)	ECSS-E-ST-40 Annex M	✓	✓		
	Independent software verification & validation report	-	✓	✓	✓	✓
	Software reuse file (SRF)	ECSS-E-ST-40 Annex N	✓ Or SQSR	✓		

Related file	DRL item (e.g. Plan, document, file, report, form, matrix)	DRL item having a DRD	SWRR	DDR	TRR	TRB
	Software problems reports and non-conformance reports	-	✓	✓	✓	✓
	Joint review reports	-	✓	✓	✓	✓
	Justification of selection of operational ground equipment and support services	-	✓			
MGT	Software development plan (SDP)	ECSS-E-ST-40 Annex O				
	Software review plan (SRevP)	ECSS-E-ST-40 Annex P				
	Software configuration management plan	ECSS-M-ST-40 Annex A				
	Training plan	-				
	Interface management procedures	-				
	Identification of NRB SW and members	-				
	Procurement data	-	✓			
MF	Maintenance plan	-				
	Maintenance records	-				
	SPR and NCR - Modification analysis report - Problem analysis report - Modification documentation- Baseline for change - Joint review reports	-				
	Migration plan and notification	-				
	Retirement plan and notification	-				
OP	Software operation support plan	-				
	Operational testing results	-				
	SPR and NCR - User's request record - Post operation review report	-				
PAF	Software product assurance plan (SPAP)	ECSS-Q-ST-80 Annex B		✓		
	Software product assurance requirements for suppliers	-				
	Audit plan and schedule	-				
	Review and inspection plans or procedures	-				
	Procedures and standards	-				

Related file	DRL item (e.g. Plan, document, file, report, form, matrix)	DRL item having a DRD	SWRR	DDR	TRR	TRB
	Modelling and design standards	-				
	Coding standards and description of tools	-				
	Software problem reporting procedure	-				
	Software dependability and safety analysis report - Criticality classification of software components	-	✓	✓		
	Software product assurance report	-				
	Software product assurance milestone report (SPAMR)	ECSS-Q-ST-80 Annex C	✓	✓	✓	✓
	Statement of compliance with test plans and procedures	-			✓	✓
	Records of training and experience	-				
	(Preliminary) alert information	-				
	Results of pre-award audits and assessments, and of procurement sources	-				
	Software process assessment plan	-				
	Software process assessment records	-				
	Review and inspection reports	-				
	Receiving inspection report	-	✓	✓		
	Input to product assurance plan for systems operation	-				

Annex B
Generic Techniques

This annex browses a number of generic techniques that are useful when applying the ECSS-E-ST-40C Standard. Within these techniques, some specific methods or languages, which are being used in space software, are briefly described with a reference. However, SysML and UML are not described here, as they can be found easily elsewhere.

B.1 Formal Methods

A major goal of software engineering is to enable developers to construct systems that operate reliably despite their complexity. One way of achieving this goal is by using formal methods.

Formal methods are mathematically based languages, techniques, and tools for specification and verification of such systems.

Use of formal methods can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected, and, combined with automated or semi-automated model driven development, could support *a priori* guarantees of correctness. However, the high cost of using formal methods means that they are usually only used in the development of high-integrity systems, where dependability, safety or security is of utmost importance.

Formal methods can be applied in general for *specification* and *verification*.

Software and system requirements are usually written in "human-readable" language. This can lead to ambiguity, when a statement that is clear to one person is interpreted differently by another. To avoid this ambiguity, requirements can be written in a formal language with mathematically defined fixed semantics. This formal specification is the first step in applying formal methods.

Formal verification provides the proof that the result (software) meets the formal specification. Verification is a progressive activity. At each stage, the new product is formally verified to be consistent with the previous product. For example, the detailed design is verified against the preliminary design, which was verified against the desired properties such as safety or security.

The use of formal languages may be criticized for making specifications unintelligible to users. Explanatory notes are required to make the formal specification notation more understandable.

For verification, many methods and software tools have been constructed for this purpose, which can be roughly classified into two categories, the so called *theorem proving* approaches and *model-checking* ones.

a. Theorem proving is a technique where both the system and its desired properties are expressed as formulas in a mathematical logic, and these properties are shown to hold by construction of specific proofs, through use of a mechanical theorem prover (often only supporting semi-automated procedures).

b. Model-checking techniques allow for the automatic verification that a specific model of the overall behaviour of a system, or a critical component, satisfies a set of requirements, or properties. Model checkers use a variety of underlying technologies, but essentially work by checking if a property holds for every reachable state.

The model is specified by means of a formal language, which can be textual or graphical. Typically, behaviours are modelled as state-transition systems. Each requirement is usually specified as a temporal logic formula.

A primary advantage of the model-checking approach, when compared with other techniques (e.g., semi-automatic theorem proving), is the user-friendliness of the tools, which support the verification activity.

They fall in the "push-button" category: once a model has been developed and a temporal logic formula characterizing a desired property of the model has been formulated, the verification of whether the model satisfies the formula is performed by the tool in a completely automatic way, i.e. without any further action required from the user. If the formula is not satisfied (i.e. the desired property is not true), the tool usually provides a counter-example in the form of a sequence of computation steps, which bring to the violation of the property.

The main disadvantage of this technique is that large models can exceed the capacity of the model checker.

However recent breakthroughs have made it possible for symbolic model checkers to explore very large spaces of reachable states. Bounded model checkers use a combination of state space exploration and induction to deal with even larger state spaces. Model checkers include today symbolic and probabilistic variants.

While theorem provers do not have the state space limitation of model checkers, they generally require more mathematical skill and labour to prove the desired properties.

Formal methods provide the techniques, methodologies and tools for producing proofs and consequently for designing proved correct systems. Of course, the use of formal methods introduces costs, in terms of additional training, specific tool support, formal specification development time, and related verification effort. Such costs can be justified when assessed in relation to the criticality of the components to which formal methods are applied.

Recent studies have demonstrated the applicability of state-of-the-art model checking techniques to support a variety of V&V activities such as consistency analysis, simulation, correctness verification, performability evaluation, dynamic fault tree generation, FMEA table generation, FDIR, and diagnosability analysis.

B.2 Functional Decomposition and Structured Analysis

B.2.1 Introduction

Functional decomposition is a traditional method of analysis. The functional breakdown is constructed top-down, producing a set of functions, sub-functions and functional interfaces.

The functional decomposition method was incorporated into the Structured Analysis method in the late 1970's. An inconvenience of the method is that factorisations are not easily possible; therefore the same sub-function may appear multiply in the tree.

Structured analysis is a name for a class of methods that analyse a problem by constructing data and control flow models. Relevant members of this class are:

a. Yourdon methods (Tom DeMarco and Ward/Mellor);

b. Structured Analysis and Design Technique (SADT).

Structured analysis includes all the concepts of functional decomposition, but produces a better functional specification by rigorously defining the functional interfaces, i.e. the data and control flows between the processes that perform the required functions. The ubiquitous 'Data Flow Diagrams' (DFD) and 'Control Flow Diagrams' are characteristic of structured analysis methods. In any form, they apply finely for data-intensive or data-driven systems, where large part of the processing is ultimately represented by a sequence of functions in pipeline.

Yourdon methods were widely used in the USA and Europe. SADT has been used within space projects for some time.

According to its early operational definition by DeMarco, structured analysis is the use of the following techniques to produce a specification of the system:

a. Data Flow diagrams (DFD);

b. Control Flow Diagrams (CFD);

c. Data Dictionary;

d. Structured English;

e. Decision Tables;

f. Decision Trees.

These techniques are more adequate for the analysis of data-centred information systems. Today is clear that SA methods are not suitable for software that is much more complex.

B.2.2 Data flow

A data flow diagram (DFD) is a graphical representation of the "flow" of data through an information system. DFDs can also be used for the visualization of data processing (structured design).

On a DFD, data items flow from an external data source or an internal data store to an internal data store or an external data sink, via an internal process.

Data flow diagrams are made up of two components:

a. annotated arrows – represent data flow in and out of the transformation centres, with the annotations documenting what the data is;

b. annotated bubbles – represent transformation centres, with the annotation documenting the transformation;

Each bubble in a data flow diagram can be considered as a stand-alone black box which, as soon as its inputs are available, transforms them to its outputs. One of the main advantages is that they show transformations without making any assumptions about how these transformations are implemented.

A DFD provides no information about the timing of processes, or about whether processes will operate in sequence or in parallel. It is therefore quite different from a control flow chart, which shows the flow of control through an algorithm, allowing a reader to determine what operations will be performed, in what order, and under what circumstances, but not what kinds of data will be input to

and output from the system, nor where the data will come from and go to, nor where the data will be stored (all of which are shown on a DFD).

DFD are used in the requirement analysis, or in the detail design and implementation phases.

Data flow analysis may support checking the behaviour of program variables as they are initialized, modified or referenced when the program executes. Data flow diagrams are used to facilitate this analysis.

Advantages of the use of the DFD include, but are not limited to the following:

a. Readily automated: there are many tools in the market supporting the representation of data flow diagrams, supporting the performance of these analyses.

b. Easy to apply: mainly due to its graphical representation and especially at later development stages when all information is available.

Disadvantages of DFD include, but are not limited to, the following:

a. Not always proper software modularity, therefore making difficult to define the diagram

b. Requires some interpretation

c. It always requires support from the tool used for the data flow definition when designing the system.

d. It can require a lot of manpower

DFD can be complemented by control flow diagrams. DFD can be contained in some control flow diagrams, from which the analyses can be performed. In addition, complementary software faults can be detected through their use.

B.2.3 Control Flow

Control flow (or alternatively, flow of control) refers to the order in which the individual statements, instructions, or function calls of a programming language are executed or evaluated. Control flow graphs are representations, using graph notation, of all control flow paths that might be traversed through a program during its execution.

In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

Examples of control flow diagrams (CFDs) include, among others, PERT, state transition, activity and transaction diagrams.

CFDs can be used as a software engineering technique, mostly for real time, event and data driven systems, as well as a complement of the engineering activities to support the dependability and safety assessment. Indeed control flow analyses are intended to identify faults, failures and their propagation, unreachable code, dead code, inconsistent or incomplete interface mechanisms between modules, and logic errors inside a module by evaluating the control flow diagrams.

CFD can be complemented by data flow diagrams. Data flow diagrams can be contained in some control flow diagrams, from which the analyses can be performed. In addition, complementary software faults can be detected through their use.

B.3 Object-Oriented Analysis (OOA)

Object-oriented analysis (OOA) can be viewed as a synthesis of the object concepts pioneered in the Simula67 and Smalltalk programming languages, and the techniques of structured analysis, particularly data modelling.

Object-oriented analysis differs from structured analysis by:

a. Building an object model first, instead of the functional model (i.e. hierarchy of data flow diagrams) – objects have a closer relation to the problem domain (the real world), while functional boxes do not;

b. Integrating objects, attributes and operations, instead of separating them between the data model and the functional model.

OOA has been quite successful in tackling problems that are resistant to structured analysis, such as user interfaces (as in interactive software). Also reactive software, being typically event-driven, can be better modelled with OOA. OOA provides a seamless transition to OOD and OO programming languages such as Smalltalk, Ada and C++. It is the preferred analysis method when object-oriented methods are going to be used in the design.

Further, the proponents of OOA argue that the objects in a system are more fundamental to its nature than the functions it provides. Specifications based on objects will be more adaptable than specifications based on functions.

Leading methods of OOA were:

a. Coad-Yourdon;

b. Rumbaugh's Object Modelling Technique (OMT) – this is an integrated collection of three models:

   1. object model;
   2. dynamic model;
   3. functional model.

   NOTE These three models collectively make the logical model required by ECSS-E-ST-40C.

c. Booch.

OOA methods evolved and today the different techniques are combined. OMT provided a first answer to this demand, and UML is today the de-facto standard.

B.4 Architecture and Design

Several design methods have been produced in order to represent the design of space software.

a. The HOOD method has resulted from merging the Abstract Machines Object Oriented Design (OOD) experiences. It was first defined to meet ESA needs and then enhanced by the HOOD Technical Group under control of the HOOD User's group, comprising major aerospace and industries companies in Europe.

   Reference: http://en.wikipedia.org/wiki/HOOD_method

b. HRT-HOOD intends to select a Scheduling Model and investigate the incorporation of the existing HOOD method in the definition and development of a design method suitable for Hard Real-Time software and based on that theory. HRT-HOOD aims at providing practitioners with a structured design method allowing the timing analysis of real-time systems. Thus, HRT-HOOD extends the HOOD method taking explicitly into account both functional and non-functional (timing) requirements which constrain a real-time system.

Reference: HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems (A. Burns, A. Wellings) Elsevier, 7 avr. 1995 - 313 pages

http://books.google.nl/books/about/Hrt_Hood.html?id=Aoch3hJhFC4C&redir_esc=y

c. The Hard Real-Time-Unified Modelling Language (HRT-UML) method provide a comprehensive solution to the modelling and analysis of hard real-time software systems, by upgrading the HRT-HOOD design concepts to a more powerful and expressive method based on UML.

HRT-UML provides a methodological process that is derived from the HRT-HOOD principles, and an UML extension profile endowed with a formal, specific semantic framework suitable to model and analyse real-time applications. HRT-UML entities are specifically customized types of components to represent real-time behaviours and to take into account non-functional properties whose fulfilment is of primary importance in a real-time design.

Reference: HRT-UML: Taking HRT-HOOD onto UML (Reliable Software Technologies — Ada-Europe 2003 ; Lecture Notes in Computer Science Volume 2655, 2003, pp 405-416) Springer, Silvia Mazzini, Massimo D'Alessandro, Marco Di Natale, Andrea Domenici, Giuseppe Lipari, Tullio Vardanega.

http://www.springerlink.com/content/1gwrgt651ap3rg3k/

d. AADL (Architecture Analysis and Design Language) is a standardized notation used to represent a computer-based physical architecture. Both a textual and graphical language, AADL permits to design and analyse software and hardware architecture of real-time systems. The language contains precise semantics to describe software tasks, data inputs and outputs, and hardware components such as busses, memories, and processors. The language can also be used to model dynamic aspects of the system such as operational modes and mode transitions. Each AADL component comes with a set of predefined properties that can be used by tools for system analysis (e.g. schedulability analysis) and this set can be extended for specific purposes. AADL is generally well suited to design and verification of the avionics architecture and to be complemented by other modelling languages and tools that focus on functional analysis or logical models in general, such as SCADE, SDL, Simulink or ASN.1. It supports annexes to extend the language, among which is predefined the Error Model Annex suitable for dependability modelling and analysis.

Reference: www.aadl.info

B.5 Data Description Languages

Data description languages are formal languages with a set of symbols defined by a formal grammar, as opposite to natural languages.

Formal languages can be used to describe in an unambiguous manner software interfaces, including data that are exchanged between subsystems. Formal languages usually offer the capability to define simple types (e.g. integer, real, string, enumerated) and complex types (e.g. structure, choice, tables). Simple types may be constrained in size and/or allowed values. Complex types may include boolean expressions to specify if a component of the complex type is present or not, or may include

mathematic expressions to specify the number of items composing a table. Formal languages are also able to carry semantics.

The main advantage in using a data description language to specify interfaces is that the interface definition (Interface Control Document) is clear and unambiguous (there is no reason to have a different or diverging understanding of the specification). The consistency of the data w.r.t. the formal definition can be proven. Indeed, the formal data definition can be interpreted in an automated fashion so that the data can be automatically checked.

There are different data description languages available, e.g.:

a. XTCE (a CCSDS standard for the XML Telemetric and Command Exchange)

b. XML Schema (a W3C recommendation for XML data Exchange)

c. ASN.1: The Abstract Syntax Notation One is a standardized notation to represent data types that was developed as ISO and ITU-T standards by the telecommunication industry. It is a simple text notation that allows precise definition of data types, and it is supported by many tools. It is widely used in many areas such as air traffic control, telecommunications, and space domain at ESA and NASA.

   Reference: http://www.itu.int/ITU-T/asn1/introduction/index.htm .

d. EAST (a CCSDS standard for any kind of data, incl. binary data)

Whatever the data description language used to specify the interface, it is recommended to use a dedicated editor to help the user in describing the interface without requiring any specific knowledge of the chosen syntax.

Since data description languages are sometimes unintelligible to users, it is also required to translate automatically the formal definition into a human-readable but faithful description. The formal definition remains the contractual specification of the interface but a more readable definition is also provided.

Data description languages allow the use of code generators that make the reading and the writing of data a smooth activity (code generators use the formal definition in order to provide a library dedicated to I/O). The update of data descriptions has less impact on the software because the I/O library is the result of an automatic process.

B.6 State machine modelling languages

State machines are extensively used to express the part of the software behaviour that is based on states and states transitions, such as protocols, mode management or fault management.

State machines are expressed by graphical languages, such as the state diagram of UML, statecharts.

Some languages are formal enough to allow for model-checking.

a. SDL (Specification and Description Language) is a modelling language that can be used to describe communicating systems. It was originally created by the telecommunication industry to solve the issue of specifying and verifying stacks of protocols prior to implementation. However, the semantics of the language is appropriate to represent all kinds of real-time systems such as on-board space systems. SDL allows for automatic code generation and automatic test generation.

   Reference: http://www.sdl-forum.org/

b. Petri Nets are a formal and graphical language for the specification, design and analysis of systems that are characterized as being concurrent, asynchronous, distributed, parallel,

nondeterministic, and/or stochastic. PN are mathematically defined but can be used as a visual communication aid, similar to data and control flow charts, interaction, state and activity diagrams. It is also an executable technique, and allows analysis methods to prove properties about the specifications. Petri Nets were studied and applied for safety critical software (Class A and B) in European human space flight infrastructure projects.

c. Reference: http://en.wikipedia.org/wiki/Petri_net

d. LTSA is a verification tool for concurrent systems. It mechanically checks that the specification of a concurrent system satisfies the properties required of its behaviour. In addition, LTSA supports specification animation to facilitate interactive exploration of system behaviour. A system in LTSA is modelled as a set of interacting finite state machines. The properties required of the system are also modelled as state machines. LTSA performs compositional reachability analysis to exhaustively search for violations of the desired properties. More formally, each component of a specification is described as a Labelled Transition System (LTS), which contains all the states a component may reach and all the transitions it may perform. However, explicit description of an LTS in terms of its states, set of action labels and transition relation is cumbersome for other than small systems. Consequently, LTSA supports a process algebra notation (FSP) for concise description of component behaviour. The tool allows the LTS corresponding to a FSP specification to be viewed graphically. LTSA has an extensible architecture which allows extra features to be added by means of plugins.

   Reference: http://www.doc.ic.ac.uk/ltsa/

B.7 ITIL ®

To maintain software systems the current best practice is to use the Information Technology Infrastructure Library (ITIL) Version 3 as a framework for the processes to be used. ITIL is the most widely accepted approach to IT service management in the world. ITIL provides a cohesive set of best practice, drawn from the public and private sectors internationally. ITIL is being widely applied to software projects as a way of managing and maintaining software.

For ground systems ITIL is extensively used as a framework. For on board flight software the maintenance is mainly focussed on providing new releases of the software. This usually means the processes are based around an extension of the Software Development Plan and ITIL process wrapper is not required.

There are three main phases that are applicable to ECSS-E-ST-40C. In order to resolve defects, or improve the software through change, a new version needs to be first designed and developed. Then this new version is transitioned into service. The software is then used within operations. Operations may identify incidents and problems with the software that will need fixing by developing a new version of the software. And so the cycle continues.

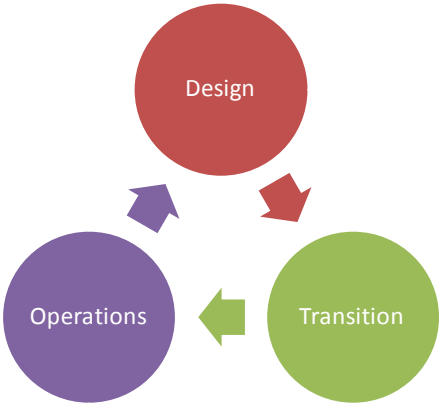


Figure 7-3 : Maintenance Cycle

Further details of how the ITIL processes can be implemented using best practice can be found on the ITIL web site.

Reference: www.itil-officialsite.com/

Annex C (normative) "Software Maintenance Plan (SMP) – DRD"

This annex provides an example for the contents of a Software Maintenance Plan (SMP) which is not currently covered in the standard ECSS-E-ST-40C.

C.1 DRD identification

C.1.1 Requirement identification and source document

The Software Maintenance Plan (SMP) document is called from the normative provisions summarized in Table 1.

Table 1. SMP traceability to ECSS-E-ST-40 and ECSS-Q-ST-80 clauses

ECSS Standard	Clauses	DRD section
ECSS-E-ST-40	5.4.3.1	<4.1>, <4.2>, <4.3>, <5.1>, <5.2>, <5.3>

C.1.2 Purpose and objective

The software maintenance plan is a constituent of the management file (MF). Its purpose is to provide the definition of organizational aspects and management approach to the implementation of the maintenance tasks.

The objective of the software maintenance plan is to describe the approach to the implementation of the maintenance process for a software product.

C.2 Expected response

C.2.1 Scope and content

<1> Introduction

a. The SMP shall contain a description of the purpose, objective, content and the reason prompting its preparation.

<2> Applicable and reference documents

a. The SMP shall list the applicable and reference documents to support the generation of the document.

<3> Terms, definitions and abbreviated terms

a. The SMP shall include any additional terms, definition or abbreviated terms used.

<4> Scope and Purpose

a. The SMP shall describe the processes and procedures necessary to provide the corrective and preventive maintenance of a software product (i.e. corrections, enhancements, improvements).

b. The SMP shall define the boundaries of the maintenance process, its starting point (receipt of the request) and the end actions (delivery and sign-off).

c. The SMP shall address the difference between maintenance and development.

<5> Application of the Plan

a. The SMP shall describe the overall flow of work including descriptions of each process step and their interfaces, and the data flow between processes.

b. The SMP shall ensure that each step in the process is controlled and measured, and that expected levels of performance are defined.

<6> General Requirements

a. The SMP describes the policies and responsibilities of the program/project team as it plans for software maintenance. Policies and responsibilities are usually spelled out in an organisation policy/directives manual and may be referenced here. It is highly recommended, however, that such doctrine be summarised in the plan as illustrated below.

<6.2> System

a. The SMP shall describe the mission of the system including mission need and employment, identification of interoperability requirements and system functions description.

b. The SMP shall describe the system architecture, components and interfaces, hardware and software.

<6.3> Status

a. The SMP shall identify the initial status of the system and the complete identification of the system with formal and common names, nomenclature, identification number and system abbreviations.

<6.4> Support

a. The SMP shall describe why support is needed.

NOTE During the projected life of period of the system, corrections and enhancements will be required. Corrective maintenance accommodates latent defects as reported by users. Enhancements or improvements are submitted in order to

improve performance and provide additional functionality for the users. As a result, maintenance support is required.

<6.5> Maintainer organisation

a. The SMP shall identify the maintainer, including the description of the maintainer team, roles and responsibilities.

NOTE This could be the separate Software Maintenance Staff or the Software Development Staff if there is no transition to a separate maintenance organisation.

<6.6> Contracts

a. The SMP shall describe any contractual protocols between next-higher-level Contractor and Contractor.

<7> Maintenance Concept

<7.1> Concept

a. The SMP shall describe the maintenance concepts, including:

1. the scope of software maintenance;
2. the tailoring of the post- delivery process;
3. the designation of who will provide maintenance;
4. an estimate of life-cycle costs.
5. the activities of post-delivery software maintenance

NOTE 1 The contractor develops it early in the development effort with help from the maintainer. Defining the scope of maintenance helps the contractor determine exactly how much support the maintainer will give to the next-higher-level Contractor. Scope relates to how responsive the maintainer will be to the users.

NOTE 2 Different organisations often perform different activities in the post-delivery process. An early attempt is made to identify these organisations and to document them in the maintenance concept. In many cases, a separate maintenance organisation performs the maintenance functions.

NOTE 3 Responsiveness to the user community is the primary consideration in determining the scope of software maintenance. The scope of software maintenance is tailored to satisfy operational response requirements. Scope relates to how responsive the Maintainer will be to proposed changes. For example, a full scope, software maintenance concept suggests that the Maintainer will provide full support for the entire deployment phase. This includes responding to all approved software change categories (i.e.: corrections and enhancements) within a reasonable period. Software maintenance concepts that limit the scope of software maintenance are referred to as limited scope concepts. Limited scope concepts limit the support period, the support level or both.

<7.2> Level of support

a. The SMP shall describe the level of support for the system, ensuring that the following requirements are met:

NOTE 1 All corrective and enhancements approved by the Software Change Control Board are included in releases.

NOTE 2 Tracking of all change requests is done.

NOTE 3 A Help Desk is maintained and technical support is provided as needed.

<7.3> Support period

a. The SMP shall describe the support period from pre-delivery to post-delivery support, on an on-call basis, to review in particular requirements, and plans.

<7.4> Tailoring the maintenance process

a. The SMP shall describe the tailoring of the maintenance process by referring to the maintainer's software maintenance process manual.

<8> Maintenance Activities

a. The SMP shall specify the specific maintenance activities.

NOTE General software engineering activities are performed during pre-delivery and post-delivery. The role of the user is defined as well as any interfaces with other organisations.

<9> Resources, methods and standards

<9.1> Resources

a. The SMP shall analyse the hardware and software most appropriate to support the organisation's needs, including:

1. The definition of the development, maintenance, and target platforms
2. The description of the differences between the environments.
3. Identification and provision of the tools sets that enhance productivity, of the way the tools are accessible, and the sufficient level of training to users.
4. Description of planning of design, implementation and testing (including associated documentation).

<9.2> Methods and standards

a. The SMP shall describe the methods and standards to be used for maintenance, in accordance with the ones applied during development process and with the applicable standards.

<10> Maintenance Process

a. The SMP shall describe how modification request are evaluated to determine its classification and handling priority and assignment for implementation as a block of modifications that will be released to the user.

b. The SMP shall describe the Software Configuration Control Board (i.e. participants, roles, activities).

c. The SMP shall describe the Maintenance process phases, including
   1. Analysis phase;
   2. Design phase;
   3. Implementation phase;
   4. Acceptance test phase;
   5. Delivery phase.

<10.2> Analysis phase

a. The SMP shall describe the analysis phase including the activities to:
   1. Determine if additional problem analysis/identification is required;
   2. Record acceptance or rejection of the proposed modification(s);
   3. Develop an agreed-upon project plan;
   4. Evaluate any software or hardware constraints that may result from the changes and that need consideration during the design phase;
   5. Document any project or software risks resulting from the analysis to be considered for subsequent phases of the change life cycle.

NOTE The analysis phase for a modification request can generate several system-level functional, performance, usability, reliability, and maintainability requirements. Each of these may be further decomposed into several software, database, interface, documentation, and hardware requirements.

<10.3> Design phase

a. The SMP shall describe the design phase.

NOTE 1 Actual implementation begins during this phase, while keeping in mind the continued feasibility of the proposed change. For example, the engineering staff may not fully understand the impact and magnitude of a change until the design is complete, or the design of a specific change may be too complex to implement.

NOTE 2 The phase can describe portions of current design specification, software development files, and entries in software engineering case tool database. Other items that may be described during this phase include a revised analysis, revised statements of requirements, a revised list of elements affected, a revised plan for implementation, and a revised risk analysis.

<10.4> Implementation phase

a. The SMP shall describe the implementation phase.

NOTE The primary inputs to this phase are the result of the design phase. Other inputs to describe for successful control of this phase include the following:

- Approved and controlled requirements and design documentation;
- Any design metrics/measurement that may be applicable to the implementation phase;
- A detailed implementation schedule, nothing how many code reviews will take place and at what level;
- A set of responses to the defined risks from the previous phase that are applicable to the testing phase.

Risk analysis and review is performed periodically during this phase rather than at its end, as in the design and analysis phases. This is recommended because a high percentage of design, cost, and performance problems and risks are exposed while modifying the system.

<10.5> Acceptance phase

a. The SMP shall describe the acceptance test phase, ensuring that the products of the modification are satisfactory to the next-higher-level Contractor, and including the description of:

1. the software system and the documentation necessary to support it.
2. the tests to validate that faults are not introduced as a result of changes.
3. the use of simulations where it is not possible to have the completely integrated system in the test facility.

NOTE The test organisation is responsible for reporting the status of the criteria that had been established in the test plan for satisfactory completion of acceptance testing.

<10.6> Delivery phase

a. The SMP shall describe the delivery phase including how replacing the existing system with the new version, how to reduce the risk associated with installation of the new version of a software system.

b. The SMP shall plan the necessary documentation, the user training, and the backup of the existing system version as well as the new version.

NOTE The backup consists of source code, requirement documentation, design documentation, test documentation and the support environment.

<11> Training Requirements

a. The SMP shall identify the training activities necessary to meet the needs of the maintenance activities.

<12> Software Product Assurance Activities

a. The SMP shall describe the software product assurance activities for maintenance if not covered by SPAP, including NCRs handling.

b. The SMP shall describe how qualification status w.r.t. the applicable Standards requirement is maintained and provided.

<13> Software Configuration Management

a. The SMP shall describe the software configuration management activities for maintenance, including SPRs/SMRs handling, NCR's, CRs and RFWs handling.

<14> Records and Reports

a. The SMP shall describe the recording, tracking and implementing maintenance including the management and description of the various forms.

NOTE The maintainer documents the problem/modification request, the analysis result and implementation options through the problem analysis report.

b. The SMP shall describe the rules for submission of maintenance reports.

c. The SMP shall describe the maintenance records content, including as a minimum the following information:

1. list of request for assistance or problem reports that have been received and the current status of each;
2. Organisation responsible for responding to requests for assistance or implementing the appropriate corrective actions;
3. priorities that have been assigned to the corrective actions;
4. result of corrective and preventive actions
5. statistical data on failure occurrences and maintenance activities.

NOTE The record of maintenance activities may be utilised for evaluation and enhancement of the_software product and for improvement of the quality system itself.

<15> Sample Request Form

a. The SMP shall describe all template modules necessary to compile the reports, taking into account longer lifetime, evolution or change of the development environment and transfer of the documentation and configuration information to the next project.