Space product assurance

Software dependability and safety

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 select and apply software dependability and safety methods and techniques.

This handbook has been prepared by the ECSS-Q-HB-80-03A Rev.1 Working Group, reviewed by the ECSS Executive Secretariat and approved by the ECSS Technical Authority.

Disclaimer

ECSS does not provide any warranty whatsoever, whether expressed, implied, or statutory, including, but not limited to, any warranty of merchantability or fitness for a particular purpose or any warranty that the contents of the item are error-free. In no respect shall ECSS incur any liability for any damages, including, but not limited to, direct, indirect, special, or consequential damages arising out of, resulting from, or in any way connected to the use of this document, whether or not based upon warranty, business agreement, tort, or otherwise; whether or not injury was sustained by persons or property or otherwise; and whether or not loss was sustained from, or arose out of, the results of, the item, or any services that may be provided by ECSS.

Published by: ESA Requirements and Standards Division
ESTEC, P.O. Box 299,
2200 AG Noordwijk
The Netherlands
Copyright: 2017 © by the European Space Agency for the members of ECSS

Change log

ECSS-Q-HB-80-03A 26 January 2012	First issue
ECSS-Q-HB-80-03A Rev.1 20 November 2017	First issue, Revision 1 The main changes with respect to the previous version are: • Alignment with Rev. 1 of ECSS-Q-ST-30, ECSS-Q-ST-40, ECSS-Q-ST-80, especially in sections 5.2.3.2 and 5.2.3.3. • Update of section 6.2 on SFMEA, addressing in particular: – Removal of any reference to SFMECA (only SFMEA) and clarification about the use of "criticality" in system- and software-level FMEA – Use of "severity", instead of "criticality", for software failure modes – Removal of discussion about potential use of SFMEA at source code level • Update of section 6.6.1 to address maintainability of software against obsolescence of development and operational environments. Introduction of sections 6.7 on "Software failure propagation prevention" and 6.8 on "Defensive programming". Changes with respect to the previous version are identified with revision tracking.

Table of contents

Change log ... **3**

Introduction ... **6**

1 Scope ... **7**

2 References ... **8**

3 Terms, definitions and abbreviated terms ... **9**

3.1 Terms from other documents ... 9

3.2 Abbreviated terms ... 9

4 Principles ... **10**

4.1 General concepts ... 10

4.1.1 Software failures and faults ... 10

4.1.2 Software reliability ... 10

4.1.3 Software maintainability ... 11

4.1.4 Software availability ... 11

4.1.5 Software safety ... 12

4.1.6 System level and software level ... 12

4.1.7 Fault prevention, removal, tolerance, and forecasting ... 12

4.2 Relation to other ECSS Standards and Handbooks ... 13

5 Software dependability and safety programme ... **14**

5.1 Introduction ... 14

5.2 Software dependability and safety workflow ... 14

5.2.1 General ... 14

5.2.2 Software dependability and safety requirements ... 15

5.2.3 Software criticality classification ... 16

5.2.4 Handling of critical software ... 22

5.2.5 Hardware-Software Interaction Analysis ... 22

6 Software dependability and safety methods and techniques ... **24**

6.1 Introduction ... 24

6.2 SFMEA (Software Failure Modes and Effects Analysis)..........24
6.2.1 Purpose..........24
6.2.2 Procedure..........25
6.2.3 Costs and benefits..........28
6.3 SFTA (Software Fault Tree Analysis)..........29
6.3.1 Purpose..........29
6.3.2 Procedure..........29
6.3.3 Costs and benefits..........30
6.4 SCCA (Software Common Cause Analysis)..........30
6.5 Engineering methods and techniques supporting software dependability and safety..........31
6.6 Software availability and maintainability techniques..........31
6.6.1 Software maintainability..........31
6.6.2 Software availability..........33
6.7 Software failure propagation prevention..........34
6.8 Defensive programming..........37

Annex A Software dependability and safety documentation..........39
A.1 Introduction..........39
A.2 Software criticality analysis report..........39
A.2.1 Criticality classification of software products..........40
A.2.2 Criticality classification of software components..........41
A.2.3 Software dependability and safety analysis report..........41

Bibliography..........43

Figures

Figure 5-1 – Software dependability and safety framework..........15
Figure 5-2 – Software dependability and safety requirements..........15
Figure 5-3 – System-level software criticality classification..........17
Figure 5-4 - Software-level software criticality classification..........20
Figure 5-5 – Feedback from software-level to system-level analyses..........21
Figure 5-6 – Hardware-Software Interaction Analysis..........23
Figure 6-1: Fault, error, failure propagation..........35

Introduction

Dependability and safety are issues of paramount importance in the development and operations of space systems. The contribution of software to system dependability and safety is a key factor, especially in view of the growing complexity of the software used in space critical applications, together with the increasing cost and schedule constraints. Hence, the need for more dependable and safe software has led to the publication of this Handbook, meant to provide guidelines on the implementation of the software dependability and safety requirements defined in ECSS-Q-ST-80C and on the application of some methods and techniques for software dependability and safety.

Analyses and activities aiming at assessing and ensuring the system dependability and safety are carried out since the early stages of the development, and software needs to be properly addressed by these system-level activities. Hardware and software products are classified based on their criticality, in order to focus engineering and product assurance activities on the most critical items. At later stages, the inherent complexity of software calls for application of specific methods and techniques, aiming at refining the software criticality classification and supporting the implementation and verification of measures for critical software handling.

This handbook provides an overall description of the entire software dependability and safety workflow, considering the different activities at system and software level, the lifecycle phases and the customer-supplier relationships, with reference to the dependability and safety requirements defined in ECSS-Q-ST-80C. Some individual software RAMS techniques are also presented. They have been selected from the list of methods and techniques mentioned in different national and international standards and literature, from which a choice has been made based on their relevance to the requirements defined in the ECSS Standards.

1 Scope

This Handbook provides guidance on the application of the dependability and safety requirements relevant to software defined in ECSS-Q-ST-80C.

This Handbook provides support for the selection and application of software dependability and safety methods and techniques that can be used in the development of software-intensive space systems.

This Handbook covers all of the different kinds of software for which ECSS-Q-ST-80C is applicable. Although the overall software dependability and safety workflow description is mainly targeted to the development of spacecraft, the described approach can be adapted to projects of different nature (e.g. launchers, ground systems).

The methods and techniques described in the scope of this Handbook are mainly focused on assessment aspects, though specific development and implementation techniques for dependability and safety (e.g. software failure propagation prevention, defensive programming) are addressed.

2 References

For each document or Standard listed, a *mnemonic* (used to refer to that source throughout this document) is proposed in the left column, and then the *complete reference* is provided in the right one.

[ECSS-Q-30]	ECSS-Q-ST-30C – Space product assurance - Dependability
[ECSS-Q-30-02]	ECSS-Q-ST-30-02C – Space product assurance – Failure modes, effects and criticality analysis (FMECA/FMEA)
[ECSS-Q-30-09]	ECSS-Q-ST-30-09C – Space product assurance – Availability analysis
[ECSS-Q-40]	ECSS-Q-ST-40C – Space product assurance – Safety
[ECSS-Q-40-12]	ECSS-Q-ST-40-12C – Space product assurance – Fault tree analysis – Adoption notice ECSS/IEC 61025
[ECSS-Q-80]	ECSS-Q-ST-80C – Space product assurance – Software product assurance
[ECSS-Q-80-04]	ECSS-Q-HB-80-04A – Space product assurance – Software metrication programme definition and implementation
[ECSS-E-HB-40]	ECSS-E-HB-40A – Space engineering – Software guidelines
[ECSS-S-ST-00-01]	ECSS-S-ST-00-01C – ECSS – Glossary of terms

3 Terms, definitions and abbreviated terms

3.1 Terms from other documents

a. For the purpose of this document, the terms and definitions from ECSS-S-ST-00-01 and ECSS-Q-ST-80 apply.

3.2 Abbreviated terms

For the purpose of this document, the abbreviated terms from ECSS-S-ST-00-01 and the following apply:

Abbreviation	Meaning
LEOP	launch and early orbit phase
RAMS	reliability, availability, maintainability and safety
FMECA	Failure Modes, Effects and Criticality Analysis
(S)FMEA	(Software) Failure Modes and Effects Analysis
(S)FTA	(Software) Fault Tree Analysis
HSIA	Hardware/Software Interaction Analysis
ISVV	Independent Software verification and Validation
CPU	Central Processing Unit

4 Principles

4.1 General concepts

4.1.1 Software failures and faults

Several definitions of failure, fault and error in relation to software exist in literature (see e.g. [NASA-8719], [Laprie92]).

The most common approach is the following: a human mistake made in requirements specification, design specification or coding can result in a *fault* to be present (latent) in a software item. This hidden defect, under particular circumstances, can manifest itself as an *error* (a discrepancy between an expected value or action and the actual one) which, in turn, can lead to a *failure*, i.e. an unexpected/unintended behaviour of the system.

Different terms are used in different Standards. [ECSS-S-ST-00-01] defines "failure" as the "event resulting in an item being no longer able to perform its required function", while "fault" is the "state of an item characterized by inability to perform as required". According to [ECSS-S-ST-00-01], a fault can be the cause of a failure, but it can also be caused by a failure. [ECSS-Q-80] mentions both "software faults" and "software failures", while [ECSS-Q-40] uses the term "software errors". [NASA-8719] refers to "software hazards". [JPL D-28444] and [ED-153] use "software failures". This handbook uses the terms "software fault" and "software error" as defined above, but "software-caused failures" rather than just "software failures", for better clarity.

Software-caused failures differ from hardware-caused failures in many respects. For instance, software is not subject to wear or energy-related phenomena which are often the cause of hardware failures. This could lead to think that "software failures" do not exist as such, because the software always does what it is programmed to do. Actually, software-caused failures occur. They are due to conceptual mistakes in requirements definition, design or coding that result in faults residing in the code, ready to manifest themselves in specific conditions. Therefore, software faults are systematic, but the conditions that lead them to cause failures are extremely difficult to predict. Software-caused failures, much as hardware ones, often appear to occur randomly.

In addition, it is worth noting that software-caused failures lead sometimes to catastrophic consequences, as the space community has experienced several times in the past decades.

4.1.2 Software reliability

Software reliability, i.e. the capability of the software to perform a required function under given conditions for a given time interval, is a key issue for space applications. Software-caused failures can result in critical degradation of system performance, up to complete mission loss.

Activities meant to increase the reliability of the software encompass the whole system and software life cycle. Software reliability requirements are derived from system reliability requirements, and responded to through the implementation of specific engineering and product assurance measures.

System reliability requirements for space applications are stated both in qualitative and quantitative terms. While consolidated reliability models exist for hardware that allow to analytically demonstrate compliance with quantitative reliability requirements, the validity of the software reliability models proposed by field engineers over the years is subject to an on-going debate within industry, academia and international standards community. One major obstacle to the usability of software reliability models is the number and nature of the assumptions to be made in order to apply the mathematical calculations on which the models are based. Those assumptions have proven to be not fully justified for the vast majority of bespoke software like space application software (see [IMECS2009]), therefore invalidating the models themselves. As a conclusion, the use of software reliability models to justify compliance with applicable reliability requirements is not advisable, hence software reliability models are not addressed in this handbook.

4.1.3 Software maintainability

Software maintainability is the capability of the software to be retained or restored to a state in which it can perform a required function, when maintenance is performed. In other words, the maintainability of the software relates to the ease with which the software can be modified and put back into operation.

Software maintainability is an issue for all space application software, but it is of extreme importance for on-board software involved in critical functions. Most of the flight software is nowadays modifiable and uploadable from ground, but in case of software faults the time needed to upgrade the software can be a major concern. An example can be a failure caused by a satellite software component that leads the satellite into safe mode. Often, the safe mode can only be maintained for a limited period of time, up to some days. This means that the maintenance team has only limited time to identify the cause of the failure, update the software, validate and upload it. It is clear that in these cases the easiness of analysing the software documentation and code, in order to quickly find the fault and correct it, is a key issue.

Software maintainability is also linked to the potential obsolescence of development and operational platforms and tools. For software with a long planned lifetime, it is important to consider from the very beginning of the development the capability to operate and modify the software at the latest stages of the operational life, possibly in different, evolved environments.

4.1.4 Software availability

Software availability is the capability of the software to be in a state to perform a required function at a given instant of time or over a given time interval. Software availability is a function of software reliability and maintainability: frequent software-caused failures and long maintenance periods reduce the probability that the software be available and operational at a certain time or for a certain period of time.

System availability depends on software availability, and this is especially true for software intensive systems, such as ground applications (like e.g. mission control systems or payload data ground systems). The dependability requirements for this kind of systems are often expressed in terms of their availability, e.g. availability of generated data.

4.1.5 Software safety

Safety is a system property. Software in itself cannot cause or prevent harm to human beings, system loss or damage to environment. However, software is a main component of the system, and therefore contributes to its safety. The contribution of software to the system safety is called software safety [NASA-8719].

It is worth recalling that reliability and safety are two different concepts, although related. Reliability is concerned with every possible software-caused failure, whereas safety is concerned only with those that can result in actual system hazards. Not all software-caused failures can raise safety problems, and not all software that functions according to its specification is safe. A boot software that fails to start up the satellite's main instrument can be judged as totally unreliable, but perfectly safe. On the other hand, since hazards are often caused by system failures, safety and dependability are linked, and [ECSS-Q-30] requires the project dependability programme to be established in conjunction with the safety programme.

[ECSS-Q-40] defines "safety-critical" the software whose "loss or degradation of its function, or its incorrect or inadvertent operation, can result in catastrophic or critical consequences". The activities that lead to the identification of safety-critical software are addressed in this handbook.

4.1.6 System level and software level

Software is an integral component of space systems. Software requirements are derived from system requirements, and the software is eventually qualified when integrated into the target system. Software is also a complex artefact that follows a dedicated life cycle and for whose development particular skills are required and specific methods and techniques are applied.

System development and software development typically occur at different engineering and management levels, also from the organizational and contractual point of view. For instance, the development of the platform application software of a satellite is often subcontracted to a different company than the Prime contractor.

This also affects software dependability and safety aspects, much like for the relation between system and subsystems/equipment. It is true that software RAMS cannot be considered in isolation from the system, and this holds in particular for software safety. However, there are safety- and dependability-related activities performed at system level by system-level engineers, considering the system as a whole, and there are other activities carried out at software level by software experts, focusing on software characteristics. The different roles, skills and organizational levels cannot be neglected or denied. Rather, a sufficient level of iterations and integration between system- and software-level dependability and safety activities at all development stages is a key to sound software development.

4.1.7 Fault prevention, removal, tolerance, and forecasting

The means to achieve software dependability and safety can be classified into four major groups of techniques: fault prevention, removal, tolerance, and forecasting.

Fault prevention aims at preventing the introduction of faults in the final product. This is mainly achieved through the application of strict rules during the requirements definition, design and production of the software (e.g. application of well-established development methods, and rigorous design and coding standards).

Fault removal is meant to detect faults and remove them from the software. Verification and validation are the principal - although not unique - means applied for fault removal.

Fault tolerance is intended to avoid that the presence of software faults leads to system failure, by tolerating the errors caused by faults and keeping the system functioning. It relies primarily on error detection and recovery, attained by different means, such exception handling or failure propagation avoidance.

Fault forecasting (or *fault analysis*) aims at estimating the present number, the future incidence, and the likely consequences of faults. In general terms, this estimation can be either qualitative, e.g. through techniques such as FMEA and FTA, or quantitative (e.g. by statistical models) even though it is recalled that only qualitative estimation is addressed in this handbook (see 4.1.2).

Within the ECSS framework, the product assurance Standards address software fault prevention (e.g. [ECSS-Q-80] requires the application of development methods and standards), fault tolerance (e.g. [ECSS-Q-30] and [ECSS-Q-40] address the definition and verification of fault tolerance requirements), and fault forecasting (e.g. [ECSS-Q-30], [ECSS-Q-40] and [ECSS-Q-80] require the application of FTA and FMEA at different levels). Fault removal is mainly covered by the engineering Standards, which define requirements for the verification and validation of software products.

4.2 Relation to other ECSS Standards and Handbooks

This handbook interfaces with other ECSS documents through ECSS-Q-ST-80. Main purpose of this handbook is to provide guidance on the implementation of ECSS-Q-ST-80 requirements on software dependability and safety.

The software RAMS requirements of ECSS-Q-ST-80 are strictly linked to the dependability requirements defined in ECSS-Q-ST-30 and the safety requirements defined in ECSS-Q-ST-40. As a consequence, this handbook makes reference to the implementation of ECSS-Q-ST-30 and ECSS-Q-ST-40 requirements relevant to software, within the boundaries imposed by the ECSS-Q-ST-80 requirements.

ECSS-Q-ST-80 interfaces with ECSS-E-ST-40, which defines software engineering requirements. This handbook makes reference to software engineering activities relevant to the verification and validation of software dependability and safety requirements, in accordance with ECSS-E-ST-40 requirements.

5 Software dependability and safety programme

5.1 Introduction

This section 5 describes the overall software dependability and safety programme, i.e. the complete set of activities performed to ensure that dependability and safety requirements applicable to software are specified and complied with.

Focus is on product assurance analyses and activities performed at software level, although system-level and software engineering activities related to software RAMS are addressed as well for completeness.

Different types of space applications exist (e.g. spacecraft, launchers, ground segments) which call for specific software dependability and safety programmes. The discussion presented in this clause leaves to the users the task to fully adapt the described processes to the development of different kinds of space applications.

5.2 Software dependability and safety workflow

5.2.1 General

Figure 5-1 provides an overview of the whole software dependability and safety framework, as derived from the requirements defined in the ECSS-Q-ST-30, ECSS-Q-ST-40, ECSS-Q-ST-80 and ECSS-E-ST-40 Standards.

NOTE The connections drawn in Figure 5-1 do not represent the entirety of the information flow relevant to the software dependability and safety framework. For instance, software requirement specification and software design are obviously input to SW RAMS analyses; however, this link is not shown in order not to confuse the diagram.

In the following sections, the different activities depicted in Figure 5-1 are described, with reference to the relevant ECSS requirements.

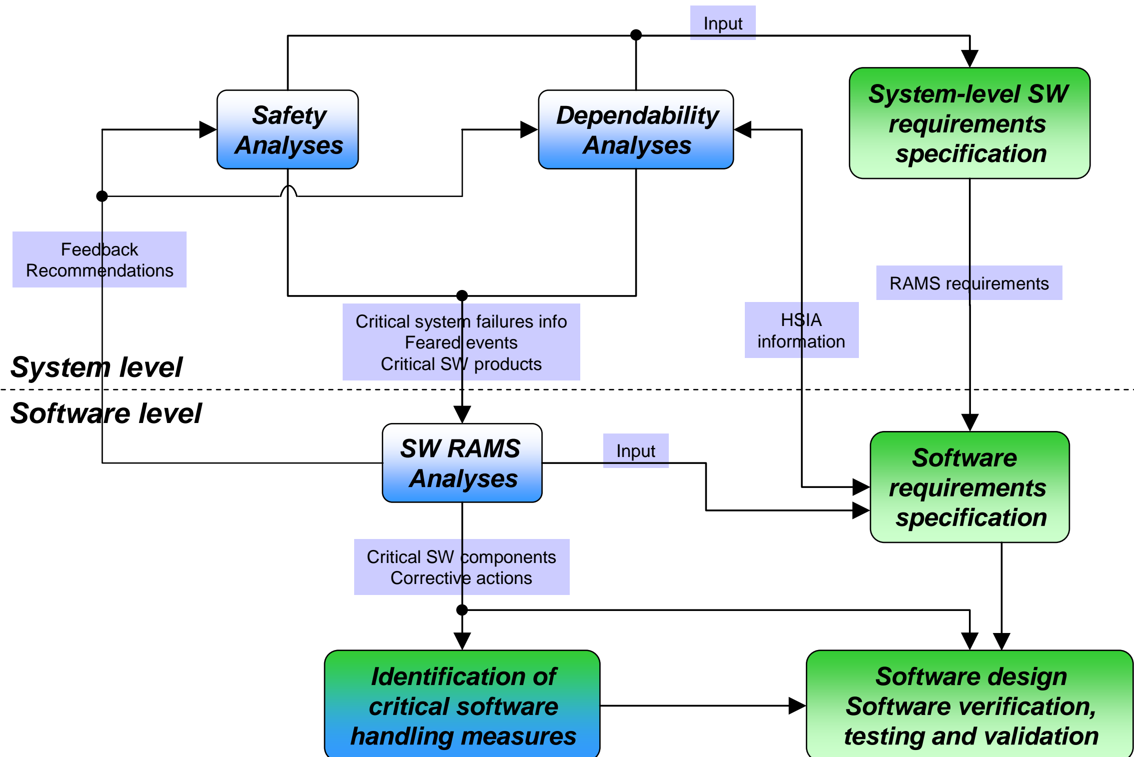

Figure 5-1 – Software dependability and safety framework

5.2.2 Software dependability and safety requirements

5.2.2.1 General

System-level dependability and safety requirements for software are defined as part of the specification of system requirements allocated to software ([ECSS-E-40] clause 5.2.2.1). They are derived from the intended use of the system and from the results of the system dependability and safety analyses.

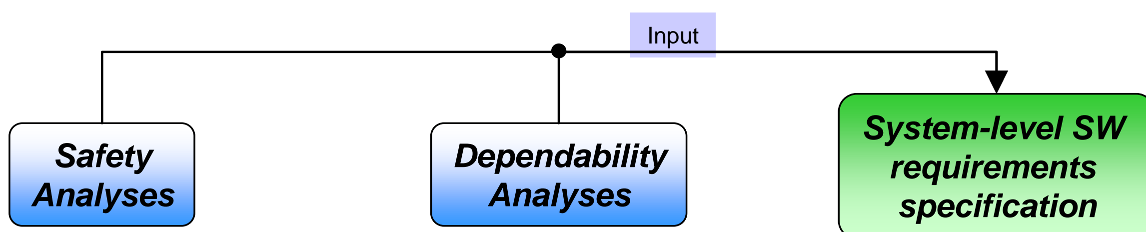

Figure 5-2 – Software dependability and safety requirements

5.2.2.2 System-level safety requirements for software

[ECSS-Q-40] requires that system-level safety requirements be defined during the early phases of the project ([ECSS-Q-40] clauses 5.7.1.1, 5.7.1.2 and 5.7.1.3). These requirements are derived from preliminary and advanced safety analyses, as well as from lessons learnt and safety requirements associated with similar previous missions. The system-level safety requirements include or need to be translated into safety requirements for the software.

The system safety requirements can be of programmatic type, e.g. addressing the need to perform specific analyses at subsystem level, and of technical type, imposing e.g. probabilistic safety targets in conformance with the requirements given by launch safety authorities. The system safety requirements allocated to software mostly correspond to the need of eventually identifying safety-critical software components, through software criticality classification, and apply to them the suitable engineering and product assurance measures (see 5.2.3).

[ECSS-Q-40] is a source itself of safety-related requirements for software. It requires that no software error cause additional failures with hazardous effects, or propagate to cause the hazardous operation of interfacing hardware (clause 6.4.2.3).

5.2.2.3 System-level dependability requirements for software

[ECSS-Q-30] clause 5.2 requires that dependability requirements be specified as part of the overall project requirements in the technical specification. These requirements are then apportioned, in a top-down process, to establish dependability requirements for lower level elements, including software.

[ECSS-Q-30] clause 5.2 also lists a set of system-level requirement types relevant to dependability. Several of these types of requirements affect software; in particular:

- quantitative dependability requirements and qualitative dependability requirements (identification and classification of undesirable events), under specified environmental conditions;
- the degree of tolerance to hardware failures or software malfunctions;
- the detection, isolation, diagnosis of system failures and related system recovery, i.e. its restoration to an acceptable state;
- the requirement for the prevention of failures crossing interfaces with unacceptable consequences (failure propagation avoidance).

While safety requirements express a set of mandatory needs, because e.g. related to human life and health, dependability requirements are traded off with other characteristics of the system, such as mass, size, cost and performance. The result is a set of requirements, both qualitative and quantitative, that correspond to the customer's acceptable level of risks for mission success.

Software dependability requirements reflect this approach. In particular, concerning quantitative software dependability requirements, the available methods and techniques do not often allow to demonstrate compliance with those requirements in an analytical fashion, contrary to what happens with hardware requirements (see discussion on software reliability models in section 4.1.2 of this document). [ECSS-Q-30] requirement 6.4.1b points out that it is not possible to quantitatively assess the dependability of software functions, and only a qualitative assessment can be made, as the dependability of software is influenced by the software development processes. Therefore, in line with the concept of acceptable risk that lies behind the specification of the dependability requirements, the supplier is expected to overcome this obstacle by demonstrating that suitable measures have been put in place to minimize the risk for the software of not meeting the applicable dependability requirements.

5.2.3 Software criticality classification

5.2.3.1 General

[ECSS-Q-40] and [ECSS-Q-30] require that system functions be classified based on their criticality (see clauses 6.4.1 and 5.4.1 respectively).

The criticality classification required by [ECSS-Q-40] and [ECSS-Q-30] is based on the severity of the consequences of system hazardous events and system failures. The severity categories Catastrophic, Critical, Major and Minor/Negligible are defined in a table which is shared by the two Standards ([ECSS-Q-40] Table 6-1, [ECSS-Q-30] Table 5-1). Safety is only concerned with the two highest levels of the severity scale, whereas dependability spans across the three lowest severity categories.

System functions and products are classified for the following reasons:

- From the safety point of view, the criticality classification corresponds to the identification of the safety-critical items, i.e. those items who can lead to hazardous events of Catastrophic and Critical severity, and to which hazard elimination, minimization and control is applied, in accordance with the [ECSS-Q-40] requirements.
- From the dependability point of view, the classification is used to focus efforts on the most critical areas during the different project phases ([ECSS-Q-30] clause 5.4).

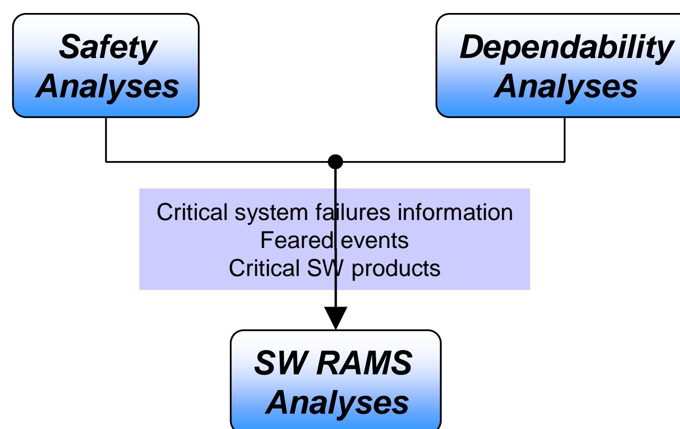

Figure 5-3 – System-level software criticality classification

5.2.3.2 System-level software criticality classification

[ECSS-Q-40] clause 6.5.6 and [ECSS-Q-30] clause 5.4.2 require the assignment of criticality categories to software products. The software criticality category is assigned to a software product based on the criticality of the function(s) implemented, taking into consideration the overall system design.

If a software product implements different system functions, the one among them having the highest criticality is considered for software criticality classification.

The association between criticality of function implemented and software criticality category to be assigned is described in [ECSS-Q-40] Table 6.3 and [ECSS-Q-30] Table 5.3. If a software product is the only system element implementing a certain function, then the criticality category to be assigned is directly linked to the function criticality: software category A for function category I, software category B for function category II, and so on.

In case compensating provisions exist, the software can be assigned a lower criticality category w.r.t. the one corresponding to the (most critical) function implemented. Compensating provisions are items that can prevent or mitigate software-caused failures, e.g. inhibits, monitors, back-ups, operational procedures. Examples of compensating provisions are, for instance:

- For thermal control software, hardware thermostats that prevent the commanded temperature from exceeding certain thresholds, independently of the temperature commanded by the software.

- A software monitor ("safety bag") that constantly checks the outputs of a certain software product and prevents specified output values from being sent to the interfacing system elements.
- An operational procedure that, through specific and detailed checks, prevents dangerous commands from being sent by a software product to a space system.

Compensating provisions allow to reduce the software criticality category of one level with respect to the case where the software is the sole means to implement the function. This reduction is broadly referred to as "software criticality downgrading", although the expression is not fully correct, because the criticality assignment is performed based on an overall assessment of the system design.

In case the compensating provisions are implemented (also) in software, the software contained in the compensating provision keeps the criticality category corresponding to the criticality of the function implemented (in other words, its criticality is not downgraded).

The notes to [ECSS-Q-40] Table 6.3 and [ECSS-Q-30] Table 5.3 stress the need for a sound function decomposition, which is key to a sensible software criticality classification.

[ECSS-Q-40] and [ECSS-Q-30] impose clear conditions on the consideration of compensating provisions as a means for software criticality downgrading. In particular, a demonstration of the effectiveness and timeliness of the compensating provisions in preventing or mitigating software faults consequences is required. This is to avoid simplistic justifications for criticality reduction. For instance, "the software can be patched" is often adduced as an explanation for certain criticality classifications. With reference to this specific type of compensating provision, possible in-flight modifications of software can only be considered as a valid method for the sake of software criticality reduction if the supplier can reasonably demonstrate that the software patching can be performed in all conditions and in time to prevent that the hazards and feared events leading to the initial software criticality categorization materialize. For example, if a postulated failure of the software product under analysis can cause a satellite to go into safe mode, and the satellite can survive in safe mode for 7 days only, the patching of that software product can be considered as a valid compensating provision for the postulated failure only if it can be demonstrated that in all cases the identification of the failure root cause, the generation of the corrected software, its verification and upload to the satellite can be performed within 7 days from the failure occurrence.

It is worth noting that the initial and then ultimate criticality classification of software products is performed at system level. Software is part of the system, and the severity of the consequences of software-triggered hazardous events and software-caused failures are evaluated at system level.

This concept is reflected in [ECSS-Q-80] clauses 6.2.2.1, where reference is made to [ECSS-Q-30] and [ECSS-Q-40] for the system-level analyses leading to the criticality classification of software products based on the severity of failures consequences.

Attention is drawn to the fact that the criticality classification of the software at system level is performed on software "products", i.e. without decomposing the software into components. There are several reasons for that:

- the initial criticality classification of system functions and products is performed during the early phases of the project, when details about the software architecture are typically not available;
- in general, there is an organizational separation between the personnel who perform the system-level RAMS analyses and the software development team(s), often including different contractual levels;
- in many cases, system-level RAMS engineers do not have sufficient knowledge of software to perform the criticality classification at software design component level.

The system-level software criticality classification has particular implications for the customer-supplier relationship. [ECSS-Q-80] clause 5.4.4 requires that, at each level of the customer-supplier chain, the customer provide the supplier with the criticality classification of the software products to be developed, together with information about the failures that can be caused at higher levels by those software products. This is because the supplier needs to know, in particular:

- the criticality category of the software being procured, in order to correctly estimate the costs of its development/procurement;
- the severity of the software potential failures identified at system level, and any useful information about those failures, so that software-level analyses can use these data for a correct criticality classification at software component level.

The timing aspects of system-level software criticality classification with respect to the customer-supplier relationship need also to be carefully considered. Failing to identify the software criticality category in the procurement documentation can result, for the customer, in a subsequent cost increase and development delay, therefore it is indispensable that the criticality category of the software items being procured be determined before the issuing of the procurement documents. With this respect, the need might arise to establish a provisional software criticality classification used in the bidding phase and then confirmed when the system-level RAMS analyses are consolidated.

5.2.3.3 Software-level software criticality classification

[ECSS-Q-80] clause 6.2.2.2 requires the supplier to perform a software criticality analysis of software products in order to determine the criticality of individual software components.

The terms "software product" and "software component" are used to designate different levels of software decomposition. Software products are normally made of several software components and sub-components (see definition in [ECSS-Q-80] clause 3.2.28). As an example, the criticality analysis at system level classifies the thermal control software and the star tracker software of a satellite as software products of criticality category B. Each of these software products is made of several components, and not necessarily all of them contribute to the overall product's criticality.

The software-level criticality classification is principally meant to identify which software components are critical, i.e. which software components can fail in such a way to cause, or contribute to cause, critical failures at system level. This allows to focus the project effort on those critical components. In accordance with [ECSS-E-40] and [ECSS-Q-80] requirements, the critical software components are subjected to more demanding engineering and product assurance activities (e.g. Independent Software Verification and Validation) that would normally be very expensive, in terms of budget, manpower and time, if applied to the whole software product. With reference to the example above, the software-level criticality classification can result in categorizing e.g. only one component of the star tracker software, in charge of bootstrap and memory management, as software of criticality category B, thereby reducing significantly the effort spent to comply with the project requirements applicable to critical software, if compared with the system-level result "star tracker software = software of criticality category B".

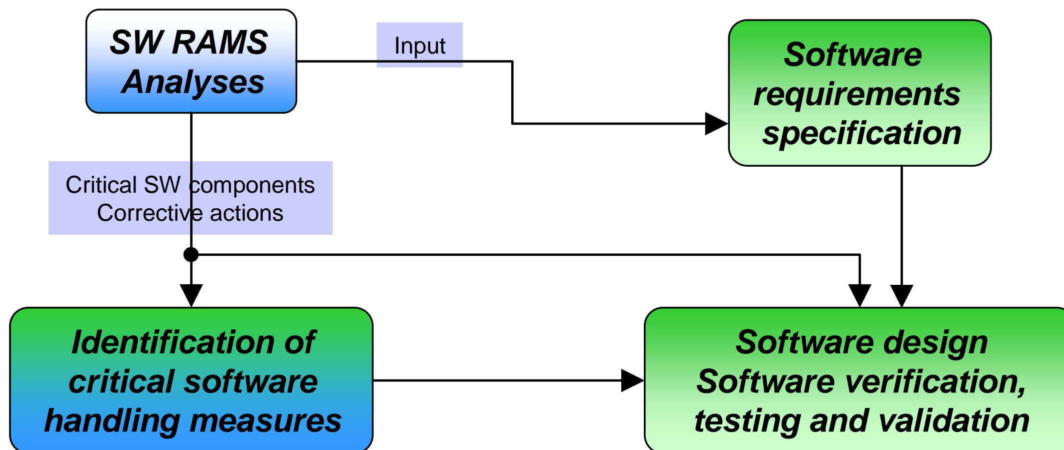

Figure 5-4 - Software-level software criticality classification

[ECSS-Q-80] clause 6.2.2.3 requires that the criticality classification of software be performed by applying a set of methods and techniques that are identified by the supplier and agreed with the customer. Guidance for the selection and application of suitable methods and techniques at software development stages is provided in section 6.

[ECSS-Q-80] clause 6.2.2.4 requires that, once the critical software has been identified, the supplier endeavour to reduce the number of critical software components, by applying suitable engineering techniques. Less critical components means less effort spent in risk mitigation measures.

For this concept to hold, [ECSS-Q-80] clause 6.2.2.10 requires that measures be defined and implemented to prevent software components of lower criticality from causing failures of software components of higher criticality (see [ECSS-Q-80] clause 6.2.2.5; [Ozarin]). If it cannot be prevented that a certain behaviour of a software component, initially classified as non-critical (based on the analysis of its functional failures), causes a software component classified as critical to fail, then both software components are classified to the criticality category of the critical one.

It is worth noting that in this case the low-criticality software component might cause the high-criticality component to fail, either because of failure propagation (e.g. in case the two components share the same memory address space and the first component unduly overwrites data belonging to the second), or because of a usage of shared resources that is "nominal" (i.e. compliant with requirements) for the low-criticality component, but might cause malfunctions in the high-criticality one (e.g. burst of data transmission by the first component on a shared network, causing the second component not to meet a certain deadline).

To be noted that, in case a low-criticality component, behaving according to its requirements, causes a failure of a high-criticality component due to shared resources, then the raising of its criticality category would not in itself solve the problem. However, the problem could be solved either by:

1. Amending the requirements of the low-criticality component to constrain its behaviour with respect to the shared resources, or
2. Putting in place proper measures (either at software or at system level) to prevent the high-criticality one from failing.

Other types of failure propagation might not be relevant to the software criticality classification requirements ([ECSS-Q-80] clause 6.2.2.10), as for instance the case where the first component generates wrong data and provides them to the second component, without causing the latter to fail or not to fulfil its requirements.

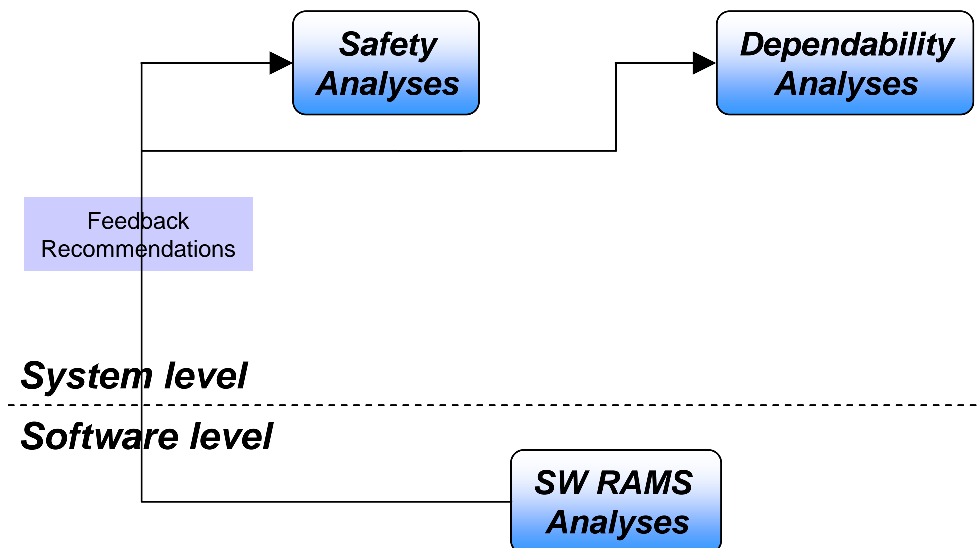

Figure 5-5 – Feedback from software-level to system-level analyses

The results of the software criticality classification are documented in a software dependability and safety analysis report. [ECSS-Q-80] clause 6.2.2.7 requires that this report be fed back and integrated into the system-level analyses, similarly to what happens for FME(C)As performed at subsystem or equipment levels w.r.t. higher level FME(C)As (see [ECSS-Q-30-02] clause 4.5). Two types of information contained in the report can be of prominent interest for the system:

- The software-level criticality analysis can lead to the identification of software-caused failures that might have a critical impact on the system, and that have not been considered by the system-level analyses. This can result e.g. in a change of criticality of the software products, or in a modification of the system design.
- The software-level criticality analysis can lead to specific recommendations for the system-level activities. For instance, it can result that the majority of the software components is classified as critical, because the system architecture does not allow to segregate the critical software. A recommendation could then be made to modify the system design, e.g. introducing two separate processor boards, one for the critical software and one for the non-critical software.

[ECSS-Q-80] clause 6.2.2.6 requires that the software dependability and safety analysis report be verified at each milestone, to confirm that the criticality categories assigned to software components are still valid. It could happen, for instance, that a modification of the system or software design, or the results of the feeding of the report to system level, call for an update of the software criticality analysis.

The need of having a provisional criticality classification of software at system level for contractual reasons (see section 5.2.3.2) can materialize also at software level, in particular for the Independent Software Verification and Validation (ISVV) business agreement (see section 5.2.4), which is required by [ECSS-Q-80] for software of criticality A and B. It is crucial to have the ISVV supplier involved in the early phases of the software development, and this can lead to a situation where, at the point in time when the business agreement is drawn up, the software criticality classification is not finalized yet, and most likely the software-level criticality classification is still to come. Since the amount of highly critical software to be subjected to ISVV is clearly a driver for the estimation of effort by the ISVV supplier, a preliminary software criticality categorization can be used in the ISVV business agreement, to be subsequently refined by a full-fledged software-level criticality analysis. This approach, of course, implies the need for a significant level of flexibility on the ISVV supplier side.

As depicted in Figure 5-4, the activities performed as part of the software criticality analysis can provide direct input for the software technical specification. The identification of some potential software failure modes by a SFMEA, for instance, can trigger the definition of new or modified software requirements, meant to overcome or reduce the effects associated with those potential failure modes.

In some cases, the effort required to perform a software-level criticality classification, together with the one necessary to prevent failure propagation between software of different criticality categories, can lead the supplier to make the decision to skip these activities and keep the criticality category assigned at system level for the entire software product. This choice can be agreed with the customer, provided that the project requirements applicable to critical software are applied to the whole software product.

5.2.4 Handling of critical software

According to [ECSS-Q-80] clause 3.2.7, critical software is software of criticality category A, B or C. As mentioned, critical software is subjected to specific engineering and product assurance measures, aiming at assuring its dependability and safety, i.e. at reducing the risks associated with its criticality to an acceptable level.

[ECSS-Q-80] Table D-2, the tailoring matrix based on software criticality, identifies the full set of [ECSS-Q-80] requirements that are applicable to category A, B and C software, therefore to critical software. Similarly, [ECSS-E-40] Table R-1 lists the software engineering requirements applicable to the different criticality categories.

In addition, the supplier is required to propose and agree with the customer a set of activities to be performed on the critical software. [ECSS-Q-80] clause 6.2.3.2 provides a list of potential critical software handling measures, including software prevention techniques, software fault tolerance mechanisms, etc. Those measures can be suitable for certain types of software, but not for others; therefore, the list included in the Standard is not mandatory.

On the other hand, there are other requirements in [ECSS-Q-80] clause 6.2.3 that are mandatory for the critical software, relevant for instance to regression testing, unreachable code and testing of instrumented code.

Two major requirements for highly critical software are contained in [ECSS-Q-80] clauses 6.2.6.13 and 6.3.5.28: Independent Software Verification and Validation, performed by a third party, is required for category A and B software (see applicability matrix, [ECSS-Q-80] Table D-2). These challenging requirements are partly softened by their respective notes, where the possibility of having a less rigorous level of independence, e.g. an independent team in the same organization, is contemplated. The notes clarify that any relaxation of those requirements is a customer’s prerogative.

The criticality classification of software components is also a driver for the definition of the test coverage goals. [ECSS-E-40] clause 5.8.3.5 provides a list of code coverage goals based on software criticality. For instance, 100 % code statement and decision coverage is mandatory for category A and B software. This requirement needs to be read together with [ECSS-Q-80] clause 6.3.5.2, which requires that customer and supplier agree upon a set of test coverage goals, based on software criticality, for the different test levels (unit, integration, validation against technical specification and against requirement baseline).

5.2.5 Hardware-Software Interaction Analysis

[ECSS-Q-30] clause 6.4.2.3 defines the purpose of the Hardware-Software Interaction Analysis (HSIA): to ensure that the software reacts in an acceptable way to hardware failures. This system-level analysis

can be seen as a complementary activity with respect to the software criticality analysis: while the former is focused on hardware failures and the corresponding software reactions, the latter looks at the effects of software-caused failures on the system, and therefore on the hardware.

Since [ECSS-Q-30] requires that the Hardware-Software Interaction Analysis be executed at the level of the software technical specification, the software supplier is actively involved in the performance of the HSIA. As the owner of the technical specification, the supplier is required to ensure that, for each potential hardware failure that might affect software, a set of requirements exist to define the software behaviour in the occurrence of that hardware failure ([ECSS-Q-80] clause 6.2.2.8).

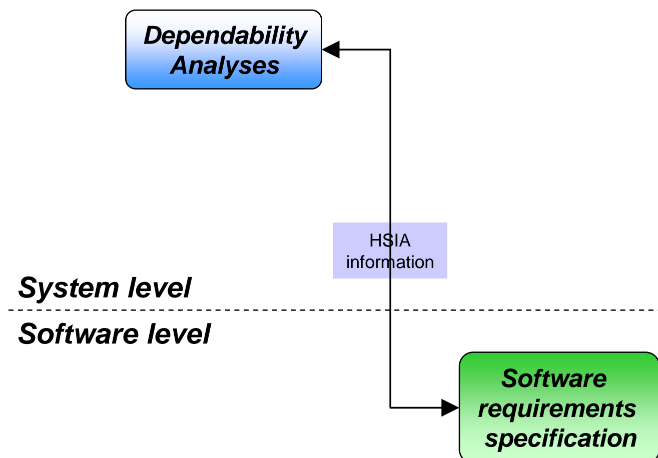

Figure 5-6 – Hardware-Software Interaction Analysis

The bi-directional arrow depicted in Figure 5-6 is meant to represent the flow of information between the HSIA, which is a system level analysis, and the specification of software requirements. On one hand, software requirements, as well as potential hardware failures, are input to the HSIA; on the other hand, the HSIA can trigger the definition of new software requirements, in case it is verified that the software reaction to specific hardware failures is not or not adequately specified.

[ECSS-Q-30-02] clause 7 provides detailed requirements for the performance of the HSIA.

[ECSS-Q-80] clause 6.2.2.9 is meant to stress the need that, as part of the software verification and validation, the results of the Hardware-Software Interaction Analysis be verified, i.e. that the software requirements specifying the software behaviour in the event of each hardware failure actually ensure that the software reaction is as expected and does not lead to any additional system failure.

The HSIA is a system-level analysis that requires a significant support from software developers. The situation where different software suppliers contribute to the development of a software product deserves special consideration. A typical case is the platform software of a satellite, which is split into basic software, developed by the supplier of the CPU hardware, and application software, developed by a software development organization. The ultimate responsibility for the production of the HSIA is with the system developer, i.e. Prime contractor. It is often the case, however, that the Prime organization has not enough visibility on the hardware and software features to generate a thorough and detailed HSIA. Most likely, the basic and application software suppliers are required to massively contribute to the analysis. In order to avoid misunderstandings and discharge of responsibility, it is important that projects define clear requirements for the production of a Hardware-Software Interaction Analysis in a multi-layer, multi-supplier set up, so that the contribution to the analysis required from the individual suppliers is clearly identified, and the Prime is aware of his task to ensure full integration of suppliers’ contributions.

6 Software dependability and safety methods and techniques

6.1 Introduction

This clause provides support and background information regarding some software dependability and safety methods and techniques.

Several techniques exist that can be applied to increase the contribution of the software to the dependability and safety of the system. As described in section 4.1.7, these techniques can be grouped into fault prevention, removal, tolerance, and forecasting.

The methods and techniques described in this handbook are the ones which are directly or indirectly linked to the ECSS-Q-ST-80C requirements on software dependability and safety. Engineering techniques, such as software dependability and safety testing, are described in [ECSS-E-HB-40] and therefore they are not extensively described in this handbook. Similarly, fault prevention methods, such as the establishment of design and coding standards, or code metrics thresholds, and software fault tolerance methods, such as partitioning and safety bags, are not described in this clause.

6.2 SFMEA (Software Failure Modes and Effects Analysis)

6.2.1 Purpose

The SFMEA (Software Failure Modes and Effects Analysis) is an adaptation of the FMEA (Failure Modes and Effects Analysis) to software.

The main purpose of the SFMEA is to identify potential software-caused failures, through a systematic and documented analysis of the credible ways in which a software component can fail, the causes for each failure mode, and the effects of each failure (including the severity of its consequences). It can also be used as a support to the allocation of criticality category to the analysed software components, based on the severity of the consequences of the potential failure modes (a software failure mode corresponds to a software error, according to the terminology introduced in section 4.1.1). However, it is important not to confuse this notion of "criticality" with this same term used in FMECA, where it is applied not to the item under analysis, but to its failure modes, and where it is a combination of the severity of consequences and probability of occurrence (the latter being excluded from the notion of criticality applied to software).

The SFMEA can be carried out at different levels of the software development for different purposes: at requirements/architectural design level for a Software Criticality Analysis, at lower level as a fault forecasting technique (see section 6.2.2.2; [Lutz-Shaw]).

6.2.2 Procedure

6.2.2.1 General

As mentioned in the note of [ECSS-Q-80] clause 6.2.2.3, [ECSS-Q-30-02] provides implementation requirements and a step-by-step procedure for the implementation of a Failure Modes and Effects Analysis. Therefore, the procedure is not repeated in this handbook.

However, in order to apply [ECSS-Q-30-02] for the execution of a SFMEA, specific considerations need to be taken into account. The following sections address these considerations and complement the [ECSS-Q-30-02] approach for a SMFEA.

6.2.2.2 Software level

The detailed requirement 4.6.f for the system FMEA contained in [ECSS-Q-30-02] specifies that the software be analysed using only the functional approach.

The SFMEA is instead a software-level analysis, and as such it is performed at different levels of software decomposition, from the specification down to the architectural design, detailed design and even source code, if necessary.

If the SFMEA is one of the techniques applied in a Software Criticality Analysis (ref. [ECSS-Q-80] clause 6.2.2.3) to determine the criticality of individual software components, then it is normally performed at specification/architectural design level. The level of software decomposition is then the lowest one for which it makes sense to assign different criticality categories to different software components. It is also important that the potential failure propagation between components of different criticality (ref. [ECSS-Q-80] clause 6.2.2.10) and the possibility to introduce segregation and partitioning between components are taken into consideration when deciding the software decomposition level to which the SFMEA is performed. In fact, there is no point in classifying two software components at different criticality categories if failure propagation between them cannot be prevented (see section 5.2.3.3).

The SFMEA can be executed at a lower level (e.g. detailed design) as a means to improve and/or to verify the dependability and safety of critical software (ref. [ECSS-Q-80] clause 6.2.3.2). In this case, if SFMEA is applied, the severity classification of failures can be used to prioritize them. For the performance of this detailed SFMEA, [FMEA-IEEE-2000] requires existence of a design description of the software at least at the level of pseudo-code. The fault of each variable and algorithm of the software can then be postulated, and the effect of the corresponding errors through the code and the output signals can be traced, in order to determine whether the propagation of these errors to the system can have intolerable consequences. The ultimate goal of this detailed analysis is to ensure that all potential software failure modes of the already designed critical software are analysed, and that corrective actions are implemented to eliminate software errors or reduce their likelihood of occurrence, and/or to compensate/mitigate their consequences on the system.

For highly critical applications, it might be necessary to confirm assumptions made during early phases of the project by cross-checking with the actual implementation of the code.

6.2.2.3 Failure modes

Although not always straightforward, it is expected that an effort is made to identify independent items within the software, and to analyse them. Failure modes are identified in relation with the functionalities of the item being analysed, i.e. no/wrong implementation of a software requirement, or no/wrong implementation of a functionality of an architectural software component.

In general, failure modes are postulated considering at least the following events:

- functionality not performed;
- functionality performed wrongly (e.g. wrong/null data provided, wrong effect);
- functionality performed with wrong timing (e.g. too late).

The following aspects need to be considered when analysing the different failure modes:

- The persistence of the error causing the failure mode. The cause can be transient or permanent. Transient errors (and thereby failures) are errors that can show up in particular situations, and that do not recur if the software is restarted (e.g. due to overloads or deadlocks). Transient errors are more difficult to discover and analyse than permanent errors (e.g. a division by zero). Corrective actions and recommendations for transient and permanent errors can differ. Separate failure mode analyses are produced in order to reflect these differences.
- Nature of the software application to be analysed. As an example, data driven systems are carefully analysed focusing on the data that can be the cause of common mode failures (data inconsistency, common databases). The causes of failures of data driven systems are different from service based ones. The SFMEA analysis covers any data interfering with the software application under analysis, whether the data comes from external sources (often called interface analysis) or from other applications.

[Neufelder] presents an approach using SFMEA at different levels of software development. Although some concepts expressed in this book might not be strictly pertinent to the performance of SFMEA as laid down in ECSS Standards, the approach of addressing failure modes and systematically postulating possible failure causes for those failure modes could support the performance of software failure mode analysis.

6.2.2.4 Failure effects

[ECSS-Q-30-02] 4.6b and 4.6c specify how failure effects and failure propagation are documented in the system FMEA. An interpretation of these requirements is necessary for the SFMEA.

In general, the SFMEA can be seen as a lower level FMEA (ref. [ECSS-Q-30-02] clause 4.5) performed by a different organization than the one at system level (level 0), in a customer-supplier relationship. According to [ECSS-Q-30-02] requirements 4.5b and 4.5c, the customer is expected to provide the software supplier with information about the critical system failures that can be caused by procured software, and the supplier is required to use this information when performing the analysis at his level. This is reflected in [ECSS-Q-80] clause 5.4.4.

When filling up the SFMEA column related to local effects and end effects, the software supplier can identify failure effects at software component and product level, but in a general case he is not able to establish what effects can that failure have on the system, because he could be unaware of prevention or mitigation devices implemented at system level. Therefore, the software supplier is expected to proceed as follows:

a. If the effects of the software failure mode do not cross the boundaries of the software product being analysed, then only local effects are documented.

b. If the effects of the software failure mode propagate through the interfaces of the software product, and it can be established that the software failure mode under analysis can be a cause of a system failure identified by the customer ([ECSS-Q-80] requirement 5.4.4a.2), then that system failure can be used to document the end effects of the software failure mode.

c. If the effects of the software failure mode propagate through the interfaces of the software product, but no match can be established with any system failure identified by the customer ([ECSS-Q-80] requirement 5.4.4a.2), then the final effects of the software failure mode is denoted as "Propagates through interfaces". This information is provided back for integration

into the system-level FMEA (see [ECSS-Q-30-02] requirement 4.5a, and [ECSS-Q-80] clause 6.2.2.7). For practical reasons, the results of this integration, i.e. the identification of any critical end effects that the software failure mode under analysis could cause at system level, are supposed to be reflected also in the SFMEA, so that it can provide a complete picture of the software-caused failure effects and be directly used for software criticality classification (see section 6.2.2.5).

6.2.2.5 Severity and criticality

For a system FMEA, [ECSS-Q-30-02] clause 4.2 requires that the severity of each failure mode be classified based on its consequences on the system, disregarding any existing compensating provisions.

For the failure analysis of software in a SFMEA, suppliers are expected to take the following into account, leading to a slightly different approach.

As described in section 6.2.2.3 above, the final effects of a software error on the system cannot always be determined at SFMEA level. The performer of the SFMEA, who is often at a lower organizational level than the system level, might not be aware of prevention or compensation measures that could mitigate the final consequences of the software failure. Therefore, an interaction with the system level is generally necessary.

With reference to the bullets of section 6.2.2.3 and to [ECSS-Q-80] Annex D.1, the severity classification of software failure modes is performed as follows:

a. If the effects of the software failure mode do not cross the boundaries of the software product being analysed and do not affect the system, then the software failure mode can be classified as level 4.

b. If the effects of the software failure mode propagate through the interfaces of the software product, and it can be established that the software failure mode under analysis can be a cause of a system failure identified by the customer (see [ECSS-Q-80] requirement 5.4.4a.2, and section 5.2.3.2 of this handbook), then the severity level assigned at system level is used to assign a severity level to the corresponding software failure mode, according to [ECSS-Q-80] Annex D.1.

c. If the effects of the software failure mode propagate through the interfaces of the software product, but no match can be established with any system failure identified by the customer ([ECSS-Q-80] requirement 5.4.4a.2), then the severity level of the software failure mode is denoted as "Propagates through interfaces". This information is then provided back for integration into the system-level FME(C)A (see [ECSS-Q-30-02] requirement 4.5a, and [ECSS-Q-80] clause 6.2.2.7), for an overall evaluation of failure consequences at system level, and criticality category assignment as specified in sections 5.2.3.2 and 5.2.3.3 of this handbook .

It is recommended that the results of the integration of the SFMEA with system-level analyses, i.e. the severity classification of the software failure modes based of their final effects at system level, considering any prevention/compensation mechanism, be documented in the SFMEA itself.

The SFMEA can then be directly used for software criticality classification, by assigning to each software component the criticality category corresponding to the severity of the failure mode, among all the potential failure modes of that software component, which has the highest severity level. Considerations of potential failure propagation between software components are essential to the final assignment of criticality category (see section 6.2.2.2).

6.2.2.6 Observable symptoms

The observable symptoms of the failure are documented in the SFMEA. If the software or system does not provide any mechanism to observe the corresponding failures, then some remarks and recommendations are preliminarily noted. A description by which occurrence of the failure mode is detected or observed is recorded. This can represent a trigger for defining maintainability requirements for the software.

6.2.2.7 Recommendations

Recommendations are an integral part of the analysis, and depend on its purpose.

If the SFMEA is primarily performed to identify critical software components, then recommendations are related to the means to reduce the risks associated with the software criticality. To this end, recommendations can include addition of new requirements, or suggestions for design measures at system level or software level (e.g. for segregation and partitioning). These recommendations are used when integrating the SFMEA with the system-level analyses (see [ECSS-Q-80] requirement 6.2.2.7a.2).

When the SFMEA is applied (generally to critical software) as a measure for fault forecasting, then the recommendations aim at the actual elimination of the consequences of the critical failures, and therefore the recommendations are uniquely identified and traced, in order to verify that suitable design and verification measures have been put in place in order to eliminate the failure consequences or reduce them to an acceptable level. To be noted that, if the consequences are actually eliminated, this may have an impact on the allocation of the criticality categories.

6.2.3 Costs and benefits

The main advantages of the SFMEA are:

- It is systematic.
- It can reveal potential system failures caused by software, not detected by system-level analyses.
- It can be used to identify the critical software components and thereby give guidance on where to focus the development, verification and product assurance effort.
- It can be used to drive and justify design and verification decisions.
- The same tools available for FMEA can be used (with the adaptations described in section 6.2.2).

The main disadvantages of the SFMEA are:

- It does not consider multiple failures.
- It can be time consuming and tedious; its difficulty increases with the complexity of the software being analysed.
- To be correctly and usefully performed, it requires specific skills, spanning from RAMS to software engineering to thorough understanding of the system operation.

6.3 SFTA (Software Fault Tree Analysis)

6.3.1 Purpose

The SFTA (Software Fault Tree Analysis) is an extension of the FTA (Fault Tree Analysis), applied to software.

A fault tree is an organized graphical representation of the conditions that cause, or contribute to, the occurrence of a defined undesirable outcome, referred to as the "top event", or "feared event".

The main purpose of the SFTA is to identify potential deficiencies in the software requirements, design or implementation that could lead to undesired events at the next software integration level.

The SFTA can be carried out at different stages of software development for different purposes: at requirements/architectural design level to support the Software Criticality Analysis, at lower level as a fault forecasting technique (similarly to SFMEA, see section 6.2.2.2; [Towhidnejad]).

6.3.2 Procedure

The SFTA can be performed by applying the procedure described in [IEC-61025], adopted by [ECSS-Q-40-12].

The top events constituting the root of the trees are selected among the system level failures which can be caused by software. These can be for instance the results of a FTA or FME(C)A executed at system level. Since the SFTA, as compared to the SFMEA, does not help identify new potential system failures, the scope of the analysis can be limited, addressing only top events corresponding to critical system failures.

Once system-level analyses have established that a software product is a contributor to a potential system failure (top event), then the SFTA can be used to identify the software components, within that software product, whose behaviour can lead to (i.e. can be the cause of) the occurrence of the top event. If the top event is a critical failure (i.e. severity level 1, 2, or 3, as classified by system-level analyses, see section 5.2.3.2), then the software components whose behaviour can contribute to that top event are classified as critical software.

The events which make up the tree are decomposed into lower-level events that can be combined through the logical gates defined in the method. The decomposition and combination of events terminates when the lowest granularity of the analysis is reached, depending on the scope and purpose of the SFTA. For a software criticality analysis, the SFTA is normally performed down to the level of events corresponding to errors of the software components whose criticality is being assessed.

The "leaf" events of a SFTA can in general be matched with a subset of the failure modes identified by a SFMEA executed on the same software product at the same decomposition level.

In case the software under analysis is of data-driven nature, the SFTA addresses all the data which are processed by the software, whether internally or externally generated; special consideration is given to data that can cause common-cause failures.

It is important that the SFTA, as opposite to the SFMEA, analyses both individual events and combination of events. At system level, the analysis of combination of failures through the FTA is typically required when potential catastrophic consequences of system failures are envisaged ([ECSS-Q-40] clause 6.4.2.1), or in general to ensure that the design conforms to the failure tolerance requirements for combinations of failures ([ECSS-Q-30] clause 6.4.2.5). At software level, SFTA is applied regardless of system failure tolerance requirements, because the method is suitable to support

the identification of critical software components and design weaknesses also in absence of safety implications.

Recommendations for the software and system design and validation can be directly drawn from the results of the SFTA, concerning for example the isolation of software components that can contribute to system critical failures, or indications of areas to be subjected to robustness testing.

6.3.3 Costs and benefits

The main advantages of the SFTA are:

- It is systematic.
- It can be used to identify the critical software components and thereby give guidance on where to focus the development, verification and product assurance effort.
- It considers multiple failures.
- It can be used to drive and justify design and verification decisions.
- A fault tree does not need to be fully developed in all its branches for the analysis to be useful.
- Tools exist to support the development of a SFTA.

The main disadvantages of the SFTA are:

- It cannot reveal potential system failures caused by software, not detected by system-level analyses (therefore, it is recommended in combination with other techniques).
- It is difficult to introduce timing and causal dependency in the analysis (dynamic behaviour).
- It can be time consuming and tedious; its difficulty increases with the complexity of the software being analysed.
- To be correctly and usefully performed, it requires specific skills, spanning from RAMS to software engineering to thorough understanding of the system operation.

6.4 SCCA (Software Common Cause Analysis)

The note to [ECSS-Q-80] clause 6.2.2.3 mentions the Software Common Cause Analysis (SCCA) as one of the methods and techniques that can be applied for a software dependability and safety analysis.

The Software Common Cause Analysis is an adaptation of the Common Cause Analysis to software. The Common Cause Analysis is required by [ECSS-Q-40] and [ECSS-Q-30] as a means to verify that the root cause of potential failure modes does not affect or negate the effectiveness of failure tolerance measures, such as redundancy. For instance, in a redundant configuration, if both the main and the redundant items are thermally coupled with the same source of heat, a failure of the heater could damage both items, invalidating redundancy as a failure tolerance mechanism for that configuration.

The Common Cause Analysis is performed in association with other analyses (e.g. FTA, FME(C)A) through the definition and application of check-lists (see [ECSS-Q-30] Annexes I and L).

Software is per se a source of common-cause failures: if the same software is used in a redundant set-up, a software fault that causes a failure in the main item, under the same circumstances, causes the same failure in the redundant item. This also applies to the typical spacecraft configuration of Nominal Mode software and Safe Mode software: any component which is common to the Nominal and Safe Mode software is a potential initiator of common-cause failures.

The need for a Common Cause Analysis at software level (SCCA) could arise in case of N-version programming, i.e. when diverse versions of the same software, developed independently, are used to perform a certain function. The use of N-version programming as a fault tolerance mechanism is controversial (see [IEEE-Trans-89], [DASC-93]) and it is not listed among the methods recommended by the ECSS Standards.

In the need of performing a Software Common Cause Analysis, the same approach described in [ECSS-Q-40] clause 7.5.4.4, and [ECSS-Q-30] clause 6.4.2.6, can be applied, with the creation of a specific check-list for common-cause failures (see e.g. [SCCA-SAFECOMP]).

6.5 Engineering methods and techniques supporting software dependability and safety

In the ECSS framework, several of the actual methods and techniques that are used to improve software dependability and safety belong to the software engineering discipline, and are therefore described in the [ECSS-E-HB-40] handbook.

Testing is one of the primary means to ensure the dependability and safety of the software. The amount of testing performed on a software item is related to its criticality. [ECSS-E-40] clause 5.8.3.5 and [ECSS-Q-80] clause 6.3.5.2 require the achievement of different test coverage goals, depending on the software criticality category.

The tests required by [ECSS-E-40] and [ECSS-Q-80] are performed to verify the correct implementation of the design and requirement specifications. Additional types of test exist that can be used to improve the dependability and safety of the software. These include, for instance, robustness testing, which is used to demonstrate the ability of the software to function correctly in the presence of invalid inputs or stressful environmental conditions.

Among the engineering methods and techniques that can be applied to assess the dependability and safety of the software, it is worth mentioning the data flow analysis, control flow analysis and schedulability analysis.

A detailed description of the mentioned engineering methods and techniques supporting software dependability and safety can be found in the [ECSS-E-HB-40] handbook.

6.6 Software availability and maintainability techniques

6.6.1 Software maintainability

Software maintainability is mainly related to the quality of software documentation and source code. [ECSS-E-40] and [ECSS-Q-80] require that maintainability requirements be defined and implemented (see e.g. [ECSS-E-40] clause 5.4.2.1; [ECSS-Q-80] clause 6.3.2.4). Maintainability is one of the characteristics of a software quality model used to specify the software quality requirements (see [ECSS-Q-80] clause 5.2.7).

In order to ensure software maintainability, it is good practice to carry out the following activities during software development:

a. System-level software maintainability requirements are specified by the customer in the requirements baseline.

Sometimes the translation of system maintainability requirements into software ones is not straightforward, especially for what concerns quantitative requirements, which are often specified for ground segments. A typical system maintainability requirement could be, for instance: "*in case of malfunction, the Mean Time To Restore shall be less than X time*". The fulfilment of this requirement by the software components of the system is hard to demonstrate. While the time needed e.g. for a switchover to a redundant hardware subsystem can be estimated and then measured, the isolation, identification and correction of a software fault, including testing and installation of the updated software, could require a time which cannot be established a priori. However, the system developer is expected to specify, in the software requirements baseline, the level of maintainability that the software is required to exhibit, considering in particular the expected operational lifetime and environment.

b. Design and coding standards are defined, submitted for customer approval and enforced.

The application of suitable design and coding standards is a primary means to achieve software maintainability. The design and coding standards reflect the applicable maintainability requirements and are enforced in the supplier organization since the beginning of the development.

c. The maintainability characteristic and the corresponding sub-characteristics (see [ECSS-Q-80-04]) are included in the software quality model, together with the related metrics, in line with the system maintainability requirements.

The quality model maintainability sub-characteristics and metrics deal, for instance, with size and complexity of the code, adherence to coding standards, testability of requirements and modularity of design. All these factors influence the capability of the software to be modified with reasonable effort and in a limited amount of time. The thresholds for the maintainability-related metrics are specified taking into account of the system maintainability requirements: the higher the degree of maintainability required, the more stringent the metric thresholds.

d. Maintainability requirements are included in the software technical specification (see [ECSS-Q-80] clause 6.3.2.4), and their implementation verified.

The quality model and metrics, described in the product assurance plan, together with the design and coding standards represent key programmatic elements for the software development, whose application needs to be verified. However, the core of the requirements against which the software is validated is defined in the technical specification, and in particular in the software requirements specification. This is why [ECSS-Q-80] requires that non-functional requirements, and in particular maintainability requirements, be defined in the SRS. In accordance with [ECSS-Q-80] clause 5.2.7.1, these software maintainability requirements (see Note below) are specified by using the elements of the project quality model(s). If maintainability is a requirement for the system, and the relevant characteristics, sub-characteristics and metrics are included in the quality model(s), then the achievement of the maintainability metrics thresholds becomes a software requirement. In general, metrics are used only as indicators. However, if the supplier can demonstrate that the software requirements specifying the achievement of certain metrics threshold goals are fulfilled, then this would help support the claim that the system maintainability requirements are also met by the software. Commercial and open-source tools exist that support the verification of the correct implementation of those software requirements.

NOTE The term "Quality requirements" is used in [ECSS-Q-80] clause 5.2.7.1 in a broad sense, and includes dependability requirements.

e. The maintainability of the software against the potential obsolescence of the development and operational environments, as well as the obsolescence of the software itself with respect to the target platform, are ensured, especially for software with a long planned lifetime.

[ECSS-Q-80] addresses this issue in clause 7.2.2.3, highlighting the need to design the software with minimum dependency on the underlying platform, in order to aid portability. This can be achieved, for instance, by applying measures such as introducing a layer between the body of the software and the underlying operating system/hardware, enforcing coding rules that prevent the use of language features dependent on a certain environment, not re-using libraries available only for specific environments.

The impact of obsolescence on system and software maintainability appears as a rather new topic in literature (see [IEEE-Trans-07], [CIRP-37-10]), often driven by military needs (e.g. the DMSMS issue: Diminishing Manufacturing Sources and Material Shortages). It is important to consider, besides the design techniques mentioned above, the following measures to mitigate the effects of obsolescence on software maintainability:

- Select and set up the development environment to be as much as possible flexible against changes in the target environment.
- Extend the maintenance business agreements with hardware and COTS software vendors, possibly resorting to software license downgrade (expand or extend the authorized use of an older product by purchasing additional licenses of the latest version and applying those licenses to the older products).
- Ensure that proper documentation (including source code) is available for maintenance, possibly including escrow agreements with original suppliers to cover proprietary information.
- Plan for maintaining the skills and knowledge necessary for the software maintenance within the organization and, as appropriate, fully transfer the knowledge to a third-party maintenance organization.

The viability of the measures to mitigate software obsolescence is to be considered at the early stages in the development cycle of space software, both for flight and ground segment, especially for long-life applications.

f. Software maintainability is supported by several other activities required by the [ECSS-Q-80] Standard, which are meant to improve the quality of the documentation and of the development and testing environment.

Although "maintainability" is not explicitly mentioned, several [ECSS-Q-80] requirements have the effect of increasing the maintainability of the software products (e.g. [ECSS-Q-80] clauses 6.3.3.7, 6.3.5.3, 6.3.5.12, 7.2.3.1, 7.2.3.4). If maintainability is a primary goal characteristic of the software being developed, then a correct and exhaustive implementation of those requirements is crucial.

In addition, a thorough and accurate configuration management of the source software code, development/testing environment and documentation, in accordance with [ECSS-Q-80] clause 6.2.4.1, is necessary in order to ensure the maintainability of the software.

6.6.2 Software availability

As mentioned in section 4.1.3, software availability is a function of software reliability and maintainability. The more reliable the software and the shorter the time needed for its maintenance, the higher the probability that the software is available when required.

Software availability requirements derive from system availability requirements. As described in [ECSS-Q-30-09] clause 5.2, there are different ways to specify system availability requirements, e.g. availability during mission lifetime for a specified service, or percentage of successfully delivered products. [ECSS-Q-30-09] 5.1.2c. requires that each availability requirements be specified in a quantitative form. While analytical and statistical methods can be applied for the availability assessment of hardware products, the bespoke nature of most of space software items does not allow the use of these methods to assess the compliance of software with quantitative availability

requirements. As a matter of fact, the availability assessment documents for spacecraft do not usually cover the on-board software.

Nevertheless, the contribution of software to system (un)availability can be significant, especially for software intensive systems. For example, for a Payload Data Ground Segment, which is mainly composed of software and commercial workstations, a primary dependability requirement can be: "*The PDGS shall deliver 98,5 % of the processed data […] in time and complete. The above requirements shall not be degraded for more than one day*". To claim compliance with such a requirement, the supplier in the first instance is expected to demonstrate that suitable engineering and product assurance measures have been put in place to provide confidence in the capability of the software to meet the imposed availability requirements. The assessment of the suitability of those measures can only be left to engineering judgement and customer approval, but certain criteria can be used to guide the software availability assessment.

a. The consequences of the software unavailability are to be evaluated. This would basically correspond to analyse the effects of software-caused failures leading to the unavailability of the function(s) implemented by software, and to estimate the worst potential consequences (see [ECSS-Q-30-02] requirement 4.1f.4). If the time needed to restore the failed software function(s) is considerable, this could lead to major mission degradation (e.g. data processing facility not available to provide emergency- or security-related Earth observation data within a useful time). In some cases, the software unavailability in particular system states or modes (e.g. for a ground segment during satellite's LEOP) could jeopardize a mission. The classification of software unavailability based on its worst potential consequences can be used to determine the criticality of the software, and therefore to select the applicable engineering and product assurance requirements.

b. Reliability is a major contributor to availability. Improving the reliability of the software increases its availability. The engineering methods mentioned in section 6.5 are to be applied to an extent that is commensurate with the required degree of system availability. It is worth noting that, for a given system, even though the worst consequences of any individual system failures are negligible, frequent software-caused failures (requiring analysis, correction, verification and installation) might have a significant impact on system availability.

The other major contributor to availability is maintainability. The considerations made in section 6.6.1 are to be applied in connection with the applicable availability requirements. For example, if a high degree of system availability is required, the supplier is expected to show that the software maintainability metrics (see [ECSS-Q-80-04]) are in a good shape, and that the configuration management of the software allows to readily rolling back to a functioning software configuration, in case a software upgrade does not work properly.

6.7 Software failure propagation prevention

The propagation of failures through software interfaces is one of the main concerns for software dependability and safety. Besides the propagation to the interfacing hardware, which is the way software errors can manifest themselves as system failures, the propagation to other software components or systems is a major threat to system dependability and safety, and suitable measures are to be put in place to prevent this from happening.

As mentioned in section 5.2.2.3, software failure propagation has a direct influence on the possibility of classifying software components at different criticality categories (see Figure 6-1).

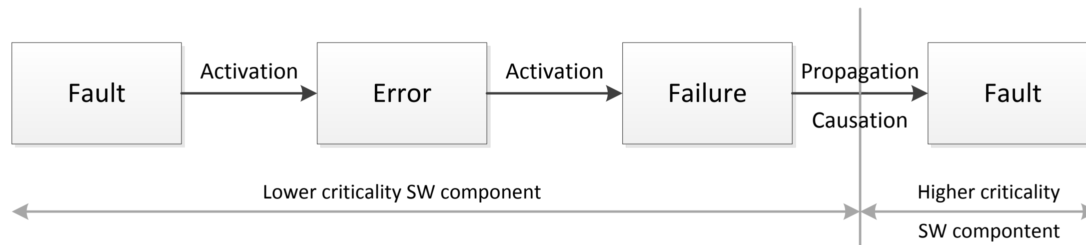

Figure 6-1: Fault, error, failure propagation

In fact, [ECSS-Q-80] clause 6.2.2.10 clearly requires that, if a software component can cause the failure of a software component of high criticality, then both components are classified at the high criticality category. The reason for this is pretty obvious: it would make little sense to concentrate the development and verification effort on the higher-criticality components only, knowing that the low-criticality components can anyhow make the higher criticality components fail and possibly lead to the same consequences which the applied effort is trying to prevent.

Table 6-1: Software failure propagation taxonomy

Type of interaction	General failure type in low-criticality software	Impact of propagation in high-criticality software
Synchronous component's interaction	Value Failure	Silent propagation unless software-dependent measures are applied
	Timing Failure	Tolerated or causes deadline miss
	Exception (unhandled or raised)	Exception propagation
	Aborted	Abortion of calling thread
Asynchronous component's interaction	Value Failure	Silent propagation unless software-dependent measures are applied
	Timing Failure	Tolerated or causes deadline miss or timeout exception
	Exception (unhandled)	No returned value (if one expected)
	Aborted	No returned value (if one expected)
Resource sharing	Resource violation	Arbitrary (including system crash)
	Resource hogging	Tolerated or causes deadline miss

Table 6-1 provides a taxonomy of potential software failure propagation events related to software criticality classification, based on the type of interactions between software items, the potential failure types and the impact on the software towards which the failure propagates. A detailed description of these failure propagation events is provided in the outputs of the [SFPP] study.

In order to be able to classify the two software components at different criticality categories, the supplier is required by [ECSS-Q-80] clause 6.2.2.10 to demonstrate that failure propagation from low-criticality component to the high-criticality component is prevented. To this end, the supplier can analyse the failure types listed in Table 6-1 and identify corresponding failure propagation prevention measures. These can be grouped in two main categories:

a. External factors: factors related to the environment in which the program executes (e.g. hardware, operating system, run-time support system) that can support the handling of some failures (e.g. their prevention, detection, etc.).

b. Application: pure software-based mechanisms that can handle (e.g. detect and support recovery from) some of the failure types identified.

The selection of the external factors plays a fundamental role in the possibility to prevent failure propagation across software components, especially if these are meant to be located on the same processing platform (e.g. single processor board). The potential influence of external factors on failure manifestation/propagation can be summarised as follows:

a. Programming language:
    1. Value failure detection (e.g. strong typing).
    2. Timing failure detection (e.g. timeouts on inter-thread communication).
    3. Exception handling.
    4. Asynchronous notification.

b. Operating system/Hypervisor:
    1. Memory management (e.g. separate memory spaces for different processes).
    2. Exception handling (e.g. division by 0).
    3. Partitioning support.

c. Processing platform (e.g. processor, mass memory unit, etc.):
    1. Support to operating system (e.g. for memory management).
    2. Identification of resource conflicts.

These external factors, to be used in combination, can influence both the software failure propagation effects and the software-based mechanisms suitable for the propagation prevention. The latter can include techniques such as:

a. Safety bag

b. Exception handlers

c. Timeout/watchdog on return

d. WCET overrun detection

It is important to note that, as documented in [SFPP], typical on-board environments used in current European space applications (e.g. processor board based on ERC32/LEON2/LEON3 + RTEMS operating system + C programming language) do not support failure propagation prevention to an extent allowing to claim compliance with the relevant [ECSS-Q-80] requirements. In particular, neither resource violation nor resource hogging failures can be prevented in these kinds of configuration.

A promising solution to ensure failure propagation prevention across different-criticality software components is the use of a hypervisor compliant with the ARINC 653 Standard, or anyway a separation kernel that allows different partitions to run on the same processor platform and be fully separated in time and space (memory). In this case, of course, the hypervisor or separation kernel are developed and validated at the highest criticality category of the software they are intended to support.

6.8 Defensive programming

"Defensive programming" is mentioned in clause 6.2.3 of [ECSS-Q-80] as a potential handling measure for critical software. There is no unanimous definition of defensive programming in the international literature and Standards.

The NASA Software Safety Guidebook [NASA-8719-GB], for instance, defines defensive programming as "a collection of methods, techniques, and algorithms designed to prevent faults from becoming failures", and the faults (defects) to be considered "can be in your own software (e.g. incorrect logic), another program in the system (e.g. sends invalid message), hardware faults (e.g. sensor returns bad value), and operator error (e.g. mistyping command)".

According to [EN-50128], the goal of defensive programming is "to produce programs which detect anomalous control flow, data flow or data values during their execution and react to these in a predetermined and acceptable manner".

It is clear from the above excerpts that the scope of defensive programming is subject to interpretation and, depending on its wideness, requires involvement of different project roles and system knowledge. In addition, defensive programming is much dependant on the programming language used (e.g. applications written in C language require a proactive check on access to array elements, whereas Ada programs can rely on the language-defined exception mechanisms in case of array index constraint violations).

Defensive programming measures can be specified and implemented with no specific knowledge of the behaviour of the software and surrounding environment. For instance, for a C function that receives a pointer as input and it is required to use the content of the variable referenced by that pointer, a defensive programming technique would be to check that the pointer is not NULL and, if so, return an error code.

Defensive programming techniques "at code level" can also be applied to make the software more robust against control flow disruptions. For instance, specific checks can be injected in software units to ensure that they are only executed when certain conditions are met (e.g. bit pattern filled in by higher-level units), and, if not, error codes are returned. This would prevent critical software functions from being wrongly invoked because of issues in the software control flow.

The above examples would be effective in avoiding that faults become failures (e.g. software crash), but the relevant techniques cannot be used in isolation. The calling functions are expected to be able to handle the returned error codes and cope with the fact that the called functions could not perform their processing. This implies that an overall strategy needs to be arranged to manage abnormal situations, convey error codes to the correct handling levels and determine the system reaction to errors detected by defensive programming techniques (e.g. graceful degradation).

This is even more relevant when, as in [NASA-8719-GB], the scope of defensive programming is extended to e.g. hardware faults, like a sensor returning a bad value. Defensive programming techniques at code level cannot cope with this kind of events alone. What sensor values are "bad"? Where are they documented? What is the software expected to do if a "bad" value is returned? Besides defensive programming, this is a matter of hardware-software ICD, HSIA, FDIR and relevant requirements contained in technical specifications.

Some basic defensive programming techniques that could be applied in the development of critical software for space applications are:

a. Check procedure parameters for range (and type and dimension, when applicable) before using them.

b. Range-check variables before using them.

c. Check variables for plausibility before using them.

It is worth noting that many good coding practices, not specific to critical software handling (e.g. use macros instead of hardcoded numbers in C, or ensure that no warnings are generated when compiling and linking the code), are part of coding standards for space applications.

For the definition of the applicable coding standards or "safe subsets" of language, it is useful to understand what the vulnerabilities of the programming language to be used are. [TR24772] contains a taxonomy of language-independent software vulnerabilities. For each vulnerability, the mechanisms of failure are presented, guidance on how to avoid or mitigate the effects of the vulnerability is provided, and the characteristics of the language for which this vulnerability could be applicable are explained. This reference also provides a cross reference to commonly used coding standards of different languages covering each vulnerability.

NOTE The term "vulnerability" is used here to address any potential software source code flaws that could affect security, safety and dependability.

Annex A Software dependability and safety documentation

A.1 Introduction

[ECSS-Q-80] clause 6.2.2 requires the generation of the following documentation:

a. criticality classification of software products;

b. criticality classification of software components;

c. software dependability and safety analysis report.

No DRD (Document Requirement Definition) for this documentation is included in the Standard. The following sections provide an example of how to organize the information required by [ECSS-Q-80] clause 6.2.2 into actual project documents.

A.2 Software criticality analysis report

The [ECSS-Q-80] requirements are generally made applicable to the first level supplier (system level), who in turn makes them applicable to the lower level suppliers through the Software product assurance requirements for suppliers (see [ECSS-Q-80] clause 5.4.2). This means that the requirements of [ECSS-Q-80] clause 6.2.2 are made applicable to the software suppliers at different organizational levels, and all of them have to produce the relevant documentation. However, depending on their responsibilities in the customer-supplier chain, different information needs to be provided.

In particular, the first level supplier (generally defined "the customer" in [ECSS-Q-80] and [ECSS-E-40] or the "Prime", or the "system-level supplier"), i.e. the organizational unit which manages, assembles and delivers the system to the final customer, is responsible for the expected output of [ECSS-Q-80] clause 6.2.2.1 ("Criticality classification of software products"). The lower level suppliers ("software-level suppliers"), who develop and deliver software products to their customers and, in the last instance, to the first level supplier, are responsible for the expected output of [ECSS-Q-80] clauses 6.2.2.2 to 6.2.2.7 ("Software dependability and safety analysis report" and "Criticality classification of software components"; see also the process described in section 5.2.3 of this handbook).

All the above mentioned documentation is relevant to the criticality classification of the software and to the measures applied to handle this criticality. This means that all the suppliers have to generate a Software Criticality Analysis Report (SCAR), with different information depending on their position in the organizational structure.

[ECSS-Q-80] clause 6.2.2.6 requires that this report is updated at each milestone, taking into consideration the interactions between system and software level.

The following sections correspond to the three main chapters of a possible Software Criticality Analysis Report in a typical contractual situation, where a first-level supplier is in charge of the system development and one or more lower-level suppliers develop different software products.

On top of the chapters described in the following sections, the SCAR is expected to contain an introductory part describing:

- the system context;
- the applicable documents;
- the requirements driving the generation of the Report;
- the criticality categories applied;
- any other relevant information.

A.2.1 Criticality classification of software products

A.2.1.1 System-level supplier

The system-level supplier provides in this section the information relevant to the classification of software products performed at system level. This includes:

a. Reference to the system-level analyses that have been carried out for the criticality classification of the software products.

b. For each software product:

   1. The identification of the individual software product.

   2. The criticality category assigned.

   3. The list of (or reference to) the potential system failure modes or undesired events that led to the criticality classification of the software product.

      NOTE A reference to the original analyses is made only if the retrieval of information relevant to the software products criticality classification is straightforward. It is anyway preferable to provide this information, suitably grouped, directly in the SCAR.

   4. For each system failure mode or undesired event:

      (a) A reference to the system level function implemented by the software product where this very failure mode can manifest itself.

      (b) The description of the failure/undesired event.

      (c) Its effects on the system.

      (d) Severity classification.

      (e) Existing prevention/compensation provisions that mitigate the consequences of the failure/undesired event.

A.2.1.2 Software-level supplier

The software-level supplier includes in this chapter (a reference to) the information received from his customer about the criticality classification of the product(s) to be developed. This information is in the form described in A.2.1.1.

A.2.2 Criticality classification of software components

A.2.2.1 System-level supplier

The system-level supplier provides in this chapter a summary of the results of the software component criticality classification performed by the software-level suppliers. This includes:

a. A collation of the results of the criticality classification of software components performed by the software-level suppliers (e.g. a table with the criticality classification of the individual software components, grouped by software product).

b. Summary of the actions undertaken to reduce the number of critical software components and mitigate the risks associated with the critical software.

A.2.2.2 Software-level supplier

The software-level supplier provides in this chapter the results of the software component criticality classification. This includes:

a. A description of the approach taken for the software component criticality classification, including analyses performed and assumptions made.

b. The criticality classification of the software components belonging to the product(s) being developed.

c. The actions undertaken to reduce the number of critical software components.

d. The description of the engineering measures applied to reduce the number of critical software components and mitigate the risks associated with the critical software.

A.2.3 Software dependability and safety analysis report

A.2.3.1 System-level supplier

The system-level supplier provides in this chapter:

a. A summary of the software dependability and safety analyses performed by software-level suppliers.

b. A detailed reference to the documents containing the results of the analyses described in A.2.3.1a.

c. All the information relevant to the interactions with software-level suppliers for the integration of their software dependability and safety analyses into the system-level analyses, including:

    1. Severity classification of the software-caused failures identified by software-level suppliers which cross software boundaries, based on their final effects on the system, taking into account any interactions with other system elements.

    2. Rationale for the severity classification provided in bullet 1.

    3. Disposition and verification of any recommendations contained in the software-level dependability and safety analyses that might affect the system.

A.2.3.2 Software-level supplier

The software-level suppliers include in this chapter the results of the software dependability and safety analyses performed.

The main results and conclusions of the analyses are to be provided first, including major recommendations for corrective and preventive actions at software and system level, followed by the bulk of the analysis documentation (e.g. SFTA or SFMEA worksheets). The analyses include the input deriving from the interactions with system-level suppliers for the integration of these software dependability and safety analyses into the system-level analyses.

Bibliography

[IEC-61025] CEI/IEC 61025:2006 – Fault Tree Analysis (FTA)

[Laprie92] J.C. Laprie, Dependability: Basic Concepts and Terminology Springer-Verlag, 1992. ISBN 0387822968

[NASA-8719] NASA-STD-8719.13B w/Change 1 – Software Safety – 8 July 2004

[JPL D-28444] Jet Propulsion Laboratory - Pasadena, California - Software Fault Analysis Handbook (Software Fault Tree Analysis (SFTA) & Software Failure Modes, Effects and Criticality Analysis (SFMECA))

[Eurocontrol] Eurocontrol Experimental Centre – EEC Note 01/04 – Review of techniques to support the EATMP safety assessment methodology

[ED-153] EUROCAE – ED-153- Guidelines for ANS software safety assurance

[Ozarin] Nathaniel Ozarin - The Role of Software Failure Modes and Effects Analysis for Interfaces in Safety- and Mission-Critical Systems - SysCon 2008 - IEEE International Systems Conference

[Lutz-Shaw] Robyn R. Lutz and Hui-Yin Shaw - Applying Adaptive Safety Analysis Techniques - Jet Propulsion Laboratory - California Institute of Technology

[Towhidnejad] Massood Towhidnejad1, Dolores R. Wallace, Albert M. Gallo - Fault Tree Analysis for Software Design - Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop

[IMECS2009] Khaled M. S. Faqih - What is Hampering the Performance of Software Reliability Models? A literature review - Proceedings of the International MultiConference of Engineers and Computer Scientists 2009

[IEEE-Trans-89] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson - The Consistent Comparison Problem in N-Version Software - IEEE Transactions on software engineering. Vol. 15. No. 11, November 1989

[DASC-93] Laura L. Pullum - A new adjudicator for fault tolerant software applications correctly resulting in multiple solutions - Digital Avionics Systems Conference, 1993. 12th DASC., AIAA/IEEE

[SCCA-SAFECOMP] Rainer Faller - Specification of a Software Common Cause Analysis Method - SAFECOMP '07 Proceedings of the 26th International

Conference on Computer Safety, Reliability, and Security

[FMEA-IEEE-2000] Peter L. Goddard - Software FMEA Techniques - Proceedings of Annual reliability and maintainability symposium, 2000

[SFPP] SWC/12-040/SFPP/PR - Guidelines And Recommendations For Prevention Of Software Failure Propagation – October 2013 - ESA Study Contract Report

[NASA-8719-GB] NASA-GB-8719.13 – Software Safety Guidebook – 31 March 2004

[EN-50128] IEC 62279: Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems (EN 50128) – March 2001

[IEEE-Trans-07] Peter Sandborn Software Obsolescence – Complicating the Part and Technology Obsolescence Management Problem - IEEE Trans on Components and Packaging Technologies, Vol. 30, No. 4, pp. 886-888, December 2007

[CIRP-37-10] Key Challenges in Managing Software Obsolescence for Industrial Product-Service Systems (IPS2) - CIRP IPS2 Conference 2010

[Neufelder] Ann Marie Neufelder - Effective Application of Software Failure Modes Effects Analysis. Quanterion, Inc. 2014

[TR24772] ISO/IEC TR24772 - Information Technology – Programming language – Guidance to avoiding vulnerabilities in programming languages through language selection and use.