ECSS-Q-HB-80-01A
5 December 2011

Space product assurance

Reuse of existing software

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

Foreword

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

The material in this handbook is defined in terms of description and recommendation how to organize and perform the work of selecting and reusing any existing software in a space project (including the use of tools for its development).

This handbook has been prepared by the ECSS-Q-HB-80-01 Working Group, reviewed by the ECSS Executive Secretariat and approved by the ECSS Technical Authority.

Disclaimer

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

Change log

ECSS-Q-HB-80-01A 5 December 2011	First issue

Table of contents

Figures

Tables

Introduction

This handbook provides guidance on the approach that can be taken when defining the implementation of activities for the reuse of existing software within a space project.

Existing software is defined in ECSS-Q-ST-80 as follows:

- Any software from previous developments that is used for the project development as is or with adaptation. It also includes software supplied by the customer for use in the project development.
- Any software including any software developed outside the contract to which ECSS software standards are applicable.
- Any software including products such as freeware and open source software products.

In the development of software systems or products, different types of existing software artefacts can be reused, such as:

- Requirements, when reused early in the software product requirements definition.
- Components, when reused early in the software product architecture definition.
- Modules, when reused at detailed design level.
- Libraries and source code, when reused at coding level.
- Documents, plans, tests, and data are other software items that can be reused.

This handbook adopts a broader interpretation of the term 'existing software', and assumes that it can comprise the 'reuse' of tools for the development of any space software product.

Furthermore, the effective reuse existing software is based on the possibility to fully understand it with respect to properties such as functionality, quality, performance, dependability or safety and to find and adopt it to the development faster than it otherwise can be constructed.

However, whatever is the level of reuse, the quality of the reused existing software is of utmost importance, as low quality can easily lead to system failure and thus loss of mission even for the lowest reuse level. Consequently, significant analyses should be carried out when using existing software. Furthermore, policies that favour reuse of existing software should be adopted with an understanding of the complex impacts of using the already available software.

1
Scope

This handbook provides recommendations, methods and procedures that can be used for the selection and reuse of existing software in space software systems.

This handbook is applicable to all types of software of a space system, including the space segment, the launch service segment and the ground segment software (including EGSEs) whenever existing software is intended to be reused within them.

This handbook covers the following topics:

- Software reuse approach including guidelines to build the Software Reuse File
- Techniques to support completion of existing software qualification to allow its reuse in a particular project
- Tool qualification
- Risk management aspects of reusing existing software

Existing software can be of any type: Purchased (or COTS), Legacy-Software**,** open-source software, customer-furnished items (CFI's), etc**.**

NOTE Special emphasis is put on guidance for the reuse of COTS software often available as-is and for which no code and documentation are often available.

Legal and contractual aspects of reuse are in principle out of scope; however guidelines to help in determine the reusability of existing software from a contractual point of view is provided in [ESA/REG/002].

Any organization with the business objective of systematic reuse may need to implement the organizational reuse processes presented in [ISO12207]. These processes will support the identification of reusable software products and components within selected reuse domains, their classification, storage and systematic reuse within the projects of that organization, etc. But these processes are out of scope of this handbook as the handbook is centred on the specific project activities to reuse an existing software product, not part of those organizational reuse processes more oriented to 'design for reuse' processes.

In addition, this handbook provides guidelines to be used for the selection and analysis of tools for the development, verification and validation of the operational software.

2 References

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

ECSS-S-ST-00-01	ECSS - Glossary of terms
ECSS-Q-ST-80	Space product assurance – Software product assurance
ECSS-E-ST-40	Space engineering – Software
BSCC(2004)	ESA software Intellectual Property Rights and Licensing
DO178B	Software considerations in airborne systems and equipment certification. RTCA DO178B/EUROCAE ED-12B. Radio Technical Commission for Aeronautics/European Organization for Civil Aviation Equipment. 1992.
ECSS-Q-HB-80-04	Space product assurance - Software metrication programme definition and implementation
ECSS-Q-HB-80-02	Space product assurance - Software process assessment and improvement
ESA/REG/002	General clauses and conditions for ESA contracts (clauses 41-43).
FAA-COTS	DOT/FAA/AR-01/26 COTS avionics Study, May 2001
FAA-DOT-handbook	DOT/FAA/AR-01/116 Software Service History Handbook. January 2002. FAA.
FAA-DOT-report	DOT/FAA/AR-01/125 Software Service History report. January 2002. FAA.
FAA-N8110.91	FAA Notice N 8110.91. Guidelines for the qualification of software tools using RTCA/DO-178B. 16/01/2001
GSWS	GAL-SPE-GLI-SYST-A/0092. Galileo Software Standard
IEC 61508	Functional safety: safety-related systems. (Parts 1-7) Ed 2.0. 2010
IEEE 1517	IEEE Standard for Information Technology - Software Life Cycle Processes-Reuse Processes
ISO 12207	Systems and software engineering -- Software life cycle processes. Edition: 2. 2008. ISO.
ISO FDIS 26262	Road vehicles -- Functional safety. FDIS Parts 1-10. 2010. ISO.

3
Terms, definitions and abbreviated terms

3.1 Terms from other documents

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

3.2 Terms specific to the present document

3.2.1 asset

item that has been designed for use in multiple contexts

[ISO 24765]

NOTE 1 an asset may be such as design, specification, source code, documentation, test suites or manual procedures.

NOTE 2 "asset" is used in this handbook as synonym of "existing software".

3.2.2 domain engineering

reuse-based approach to defining the scope (i.e., domain definition), specifying the structure (i.e., domain architecture), and building the assets for a class of systems, subsystems, or applications

[ISO 24765]

3.2.3 operational software

software product which contributes directly to the mission

[GSWS]

NOTE Contractual aspects are not considered in this definition.

3.2.4 reuse

building a software system at least partly from existing pieces to perform a new application

[ISO 24765]

NOTE Traditionally achieved using program libraries. Object-oriented programming offers reusability of code via its techniques of inheritance and genericity. Class libraries with intelligent browsers and application generators are under development to help in this process. Polymorphic functional languages also supports reusability while retaining the benefits of strong typing.

3.2.5 reuse software

see "existing software" in ECSS-Q-ST-80.

3.3 Abbreviated terms

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

Abbreviation	Meaning
ESA	European Space Agency
FAA	U.S. Federal Aviation Authority
PSH	product service history
SCMP	software configuration management plan
SDP	software development plan
SFMECA	software failure mode effect and criticality analysis
SFTA	software fault tree analysis
SQA	software quality assurance
SRF	software reuse file
SVVP	software verification and validation plan
V&V	verification and validation

4
Overview of the handbook

4.1 Introduction

This clause 4 contains an introduction of the content of this handbook, the intended audience and how to use this handbook.

The organization of this handbook is reflected in detail in Figure 4-1. This handbook is organized in ten main parts:

- Section 1. Scope
- Section 2: References
- Section 3: Terms, definitions and abbreviated terms
- Section 4: Overview of the handbook
- Section 5: Software reuse approach
- Section 6: Tool qualification
- Section 7: Techniques to support qualification when reusing existing software

Annexes include detailed information about:

- Annex A: Content of Software Reuse File (SRF)
- Annex B: Content of the Product Service History file
- Annex C: Risk management considerations

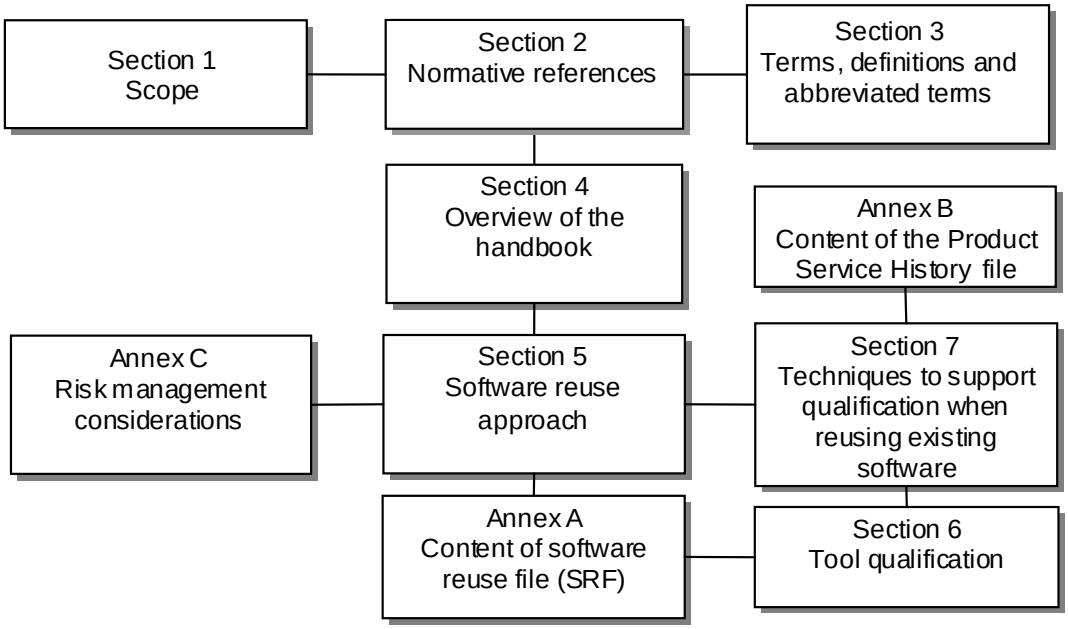


Figure 4-1: Organization of the handbook

4.2 Relation to other ECSS Standards

4.2.1 General

Section 4.2 discusses how this handbook interfaces with other ECSS series, namely the ECSS-Q series of standards (product assurance), ECSS-E series of standards (engineering) and the ECSS-M series of standards (management).

The interface of this handbook to the ECSS-Q branch is via ECSS-Q-ST-80; equally, the interface of this handbook to the ECSS-E branch is ECSS-E-ST-40.

The ECSS-M branch defines the requirements to be applied to the management of space projects. ECSS-E-ST-40 and ECSS-Q-ST-80 describe how the ECSS-M standards apply to the management of software projects. In addition, requirements that cannot be found in the M-branch because they are specific to software product assurance are defined in ECSS-Q-ST-80.

Therefore, this clause contains an analysis of ECSS-E-ST-40 and ECSS-Q-ST-80 requirements related to the reuse of software in space systems.

4.2.2 Software engineering

The interface of this handbook to the ECSS-E branch is via ECSS-E-ST-40; in turn, the interface of ECSS-E-ST-40 to this handbook is via the ECSS-Q-ST-80.

ECSS-E-ST-40 covers all aspects of space software engineering from requirements definition to retirement. It defines the scope of the space software engineering processes, including details of the verification and validation processes, and their interfaces with management and product assurance, which are addressed in the management (-M) and product assurance (-Q) branches of the ECSS system.

ECSS-E-ST-40 contains some specific clauses applicable to projects that intend to reuse software products from other space projects and third-party "commercial off-the-shelf" products to be part of the software product

ECSS-E-ST-40 clauses 5.4.2.1 and 5.4.3.7, respectively, invokes clause 6.2.7 of ECSS-Q-ST-80 for requirements on the use of existing software. Clause 5.4.3.7 of ECSS-E-ST-40 requires the evaluation of the reuse potential of the software to be performed through the identification of the reuse components with respect to the functional requirements baseline.

ECSS-E-ST-40 contains a DRD for the Software Reuse File (SRF) as a constituent of the design justification file (DJF). Its purpose is to document the identification and analysis to be performed on existing software intended to be reused.

This handbook also provides guidance for gaining confidence of the qualification status of any tool used for the development, verification or validation of the space operational software. This handbook will explicitly complement the implementation of ECSS-E-ST-40 tool related clauses, such as: 5.3.2.1 with requirements about development techniques (often supported by the use of tools) and testing environment, 5.3.2.4 containing requirements about supporting tools for automatic code generation, 5.6.2 mentioning validation tools, 5.8.2.1 mentioning verification tools.

4.2.3 Software product assurance

ECSS-Q-ST-80 Standard defines software product assurance requirements for the development of software in space projects in order to provide confidence to the customer and to the suppliers that developed or reused software satisfies the requirements throughout the system lifetime. In particular, ECSS-Q-ST-80 specifies requirements to ensure the software is developed to perform as expected and safely in the operational environment meeting the quality objectives agreed for the project.

Clause 6.2.7 in ECSS-Q-ST-80 contains requirements about reuse of existing software and specifies the term reuse software as it is used in the handbook. This handbook supports the implementation of the requirements contained in ECSS-Q-ST-80 Clause 6.2.7.

Assessing the impact and deriving extra requirements to ensure any deactivated code or configurable code, potentially happening when reusing existing software, do not harm the operational software and system (as required by requirements 6.2.6.5 and 6.2.6.6 of ECSS-Q-ST-80) is also mentioned in this handbook.

This handbook also provides guidance to cope with the selection of suppliers of existing software as required ECSS-Q-ST-80 in Clause 5.4.1.2.

As this handbook also provides guidance for gaining confidence in the qualification status of any tool used for the development, verification or validation of the operational space software, it supports the implementation of clause 5.6 in ECSS-Q-ST-80, about tools and supporting environment detailing development environment requirements.

4.2.4 Project management

The ECSS-M branch defines the requirements to be applied to the management of space projects. ECSS-E-ST-40 and ECSS-Q-ST-80 describe how the ECSS-M series of standards apply to the management of software projects. In addition, requirements that cannot be found in the M-branch because they are specific to software product assurance are defined in ECSS-Q-ST-80.

These management-related processes are directly handled in this handbook through the interfaces to ECSS-E-ST-40 and ECSS-Q-ST-80.

5
Software reuse approach

5.1 Introduction

Different existing software artefacts can be considered for reuse in each application engineering processes: requirements analysis, design, coding, integration, testing, installation, maintenance and operations. Therefore, there are specific activities that should be performed at a very early phase of the project in order to ensure that the most suitable existing software is considered for reuse in the current application. The suppliers should assess different options relevant to reuse and new development, evaluating them with respect to criteria such as risks, cost and benefits. These options include:

a. Purchase an off-the-shelf, COTS software (no source code available) that satisfies the requirements

b. Develop the software product or obtain the software service internally

c. Develop the software product or obtain the external software service through contract

d. Use open source software products that satisfies the requirements

e. A combination of a, b, c and d above

f. Enhance an existing software product or service

Clause 5 describes the activities to be performed and considerations to be made when reusing existing software in a project. Choosing between creating the software in-house or reusing existing software is not an easy decision. This choice should be made very early in the project, when often there is still no information about the full set of functionalities nor the design. Only when systematic reuse is an established policy in an organization, reusing existing software can be the starting approach in any project. The organization would have the reuse-related processes deployed (see [ISO12207]) and any project would first access the library of existing reusable products to check for any one that could fit into the project concerned. Nevertheless, no matter what the organizational situation is, a systems approach should be taken to consider how the existing software will fit into the new software application to be developed.

The aim of this clause is to define a chronology of events and activities in order to correctly document the selection, justification and qualification of the existing software to be reused in the current application. As presented at the Figure 5-1 the phases that should be performed for each reused existing software item are the following:

- Requirements phase – definition of the requirements to be fulfilled by the existing software candidates by requirements identification, gap analysis and definition of derived requirements.
- Assessment phase –selection and justification of the best choice according to the identified requirements from the previous phase and identification of corrective actions.
- Integration Phase – acquisition, inspection, adaptation, configuration management of the selected reused existing software into the system software of the project.

- Qualification Phase – qualification activities performed on the existing software reused in parallel to current software development.

NOTE Throughout this clause special considerations are made when the existing software to be reused is what is often identified as COTS: black box commercially available software for which neither its source code nor any other development information is available when acquired. COTS software usage may require considerations of glue code, architectural mitigation techniques, derived requirements and COTS software specific integration issues for checking consistency. Any supplemental software due to COTS software incorporation in software systems is considered developmental software to which all of the objectives and requirements of the project apply.

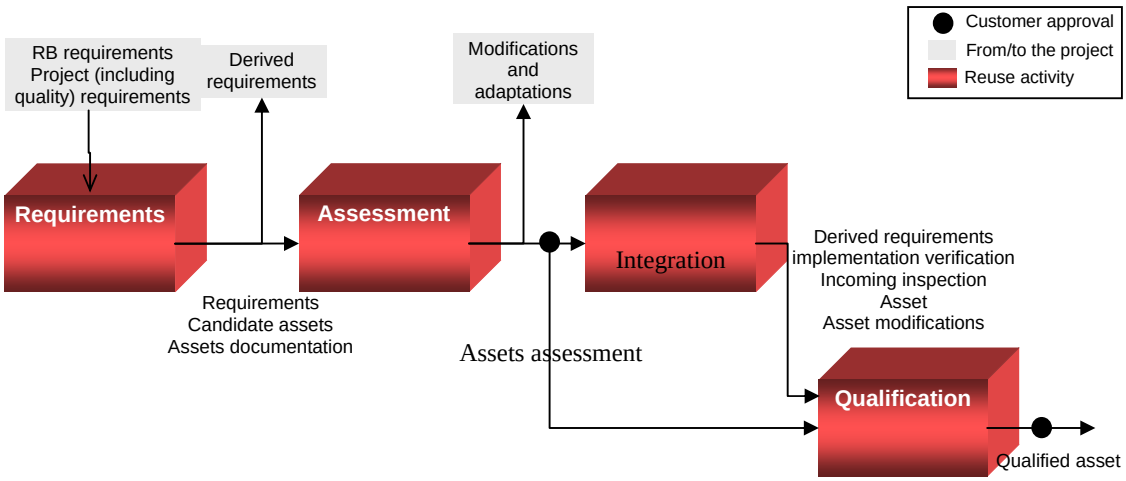


Figure 5-1: Specific reuse activities within project

All results are documented in the so-called Software Reuse File (SRF), sections of which will be referred throughout the performance of the different reuse activities detailed below. Annex A of this handbook presents guidelines following the required SRF DRD content as defined in the Annex N of ECSS-E-ST-40.

Generally a single SRF document is expected to cover ALL the reused existing software foreseen in the context of a software project, but it may be appropriate to produce specific SRFs targeted to individual existing software products when there are large, numerous or complex. It should be noted that documentation concerning reused or existing software is not simply limited to the (Software Reuse File) SRF. There are software requirements that call for adequate planning concerning reused software to be present in the SDP, the SCMP, the SPAP, the SVVP, and the SMP. Furthermore, it is clear that reused software considerations should be properly covered (if needed) in a wider set of documents (e.g. SDD, IR, SPR, etc.). Reporting on activities concerning existing software is also expected in the SPAMR and SW progress reports. Lastly, software supplier should comply also with all SW requirements relating with configuration management and control of reused software and tools as described in 5.4.3.

5.2 Requirements phase

5.2.1 Overview

The system's software requirements definition process identifies software requirements that existing software should satisfy in compliance with clause 5.2 of the ECSS-E-ST-40 with the aim to put the existing software in the context of the system requirements allocated to the software (e.g. any FDIR related concepts/measures/mechanisms built into the reused existing software need to be validated for compliance with the system-level dependability analysis and requirements). Existing software can contain more features than the requirements needed by the system under development. A definition of these features may be available from the supplier or derived from the user manuals, technical materials, product data etc. Conversely, due to the use of existing software, there can be derived requirements (e.g. platform dependent requirements, interrupt handling, interface handling, resource requirements, usage constraints, error handling, partitioning) to be added to the system software requirements. This is more likely true for COTS.

The following are the activities a project should perform and document within this reuse phase:

a. The requirements to be fulfilled by the reusable existing software should be identified from project requirements

b. Gap analysis

c. Identification of derived requirements

5.2.2 Requirements identification

The aim is to concentrate on producing a detailed functional description in terms of functional requirements and features needed for the reused existing software. This could for example be based on:

- specific next-higher level/customer requirements that map directly to a potential existing software component (e.g. upper level requirements related specifically to the operating system).
- additional high-level requirements identified directly by the software supplier (e.g. during specification, architectural or detailed design phase).
- design constraints, performance, budget and timing requirements, target environment constraints, etc.

The requirements analysis for the reuse of the existing software should be established taking into account:

- the resources and performances in the operational environment of the original project of the existing software and the operational functionality to be provided by the current application.
- rules for error handling including the propagation of errors from the existing software.
- non functional requirements, including criticality category level, are identified for the existing software.
- the description of the operational environment of the application that uses the existing software.
- qualification methods (by inspection, analysis, test, etc.) are detailed.
- maintenance and support requirements.

The software supplier should clearly identify the critical functions from the safety and dependability analysis that determine the quality level of the reused existing software with respect to the project requirements. The qualification campaign will need to concentrate on demonstrating that these identified functions are successfully realized by the reused existing software. When the existing software is used to cover functionality across various software criticality categories, the requirements for error-handling and propagation need to be clearly specified, as well as appropriate protection/partitioning requirements. Qualification activities will need to be performed to avoid that errors in the existing software at a lower software criticality category do not affect the required higher criticality category functionality.

Much of the work to achieve a proper functional description (i.e. requirements definition) would be performed in parallel with the software architectural design activities. The software supplier, faced with upper level requirements to satisfy through a software (+hardware) solution can choose to apportion parts of that architecture directly to existing software or to SW components that contain existing software. It should therefore be entirely possible to identify the functional requirements that will be needed to be realized by the existing software (as well as other non-functional requirements).

The results of the requirements analysis specifying whether the existing software fulfils the project requirements should be documented in section <4.1> of the Reuse Software File (SRF) for all criticality categories, as presented in Annex A of this handbook. It should be provided at SRR and PDR reviews.

As part of this requirements activity, potential existing software candidates that can satisfy the project requirements should be identified.

All technical and management information available for each existing software candidate should be collected. The suppliers should ensure also that they collect and document in the SRF the information related to all identified candidates to be reused (including license agreements, costs, supplier support policy, maintenance and warranty conditions, and more).

The detailed information about each candidate and its supplier(s) should be documented in section <4.2> of the Reuse Software File (SRF) for all criticality categories, as presented in Annex A of this handbook. This information should be provided at SRR and PDR reviews.

5.2.3 Gap analysis

Each candidate existing software documentation - especially specification, if exists - should be examined, and a coverage analysis should be conducted against the requirements identified (see clause 5.2.2). The purpose of this analysis is to determine up to which extension the existing software requirements match the project software requirements and to aid in the comparison of candidates.

All available life cycle data from each candidate should be assessed. A gap analysis should be performed to analyse the quality level of the existing software with respect to the project requirements, according to the criticality of the functions intended to be implemented, taking into account the detailed aspects referred in ECSS-Q-ST-80 clause 6.2.7.4.

In addition, in this gap analysis, the comparison between the software development standards applied during the development of the existing software (if known) and the software standards applicable to the current project is to be performed to aid on the later reuse qualification activities to be done.

The gaps identified during this phase should be translated into constraints that can be used as inputs for a new iteration of the requirement analysis and a further analysis of the impact on the overall software architecture and design.

The gap analysis for each candidate should be documented in section <5.1> of the Reuse Software File (SRF) for all criticality categories, as presented in Annex A of this handbook. This information should be provided at SRR and PDR reviews.

5.2.4 Derived requirements identification

Analysis should be conducted to identify derived requirements. This analysis should include all existing software functions, needed and unneeded. Derived requirements can be classified as follows:

— Requirements to prevent adverse effects of any unneeded functions of any existing software. This can result in design techniques to work around the integration of the existing software. The aim is to ensure that any deactivated code is not activated or that its accidental activation does not harm the operation of the software system as required in clause 6.2.6.5 of ECSS-Q-ST-80.

— Requirements that the selected existing software can impose on the system including those for preventing adverse effects of needed existing software functions (e.g. input formatting, call order, initialisation, data conversion, resources, range, checking). This can result in interface code, coding directives, architecture considerations, resource sizing, glue-code etc.

— Requirements for software containing configurable code. The aim is to ensure that any unintended configuration cannot be activated at run time or included during code generation as required in clause 6.2.6.6 of ECSS-Q-ST-80.

These derived requirements should be documented in section <5.2> of the Reuse Software File (SRF) for all criticality category levels, as presented in Annex A of this handbook. This information should be provided at SRR and PDR reviews.

5.3 Assessment phase

5.3.1 Overview

The focus of section 5.3 is on the assurance of quality and technical aspects when acquiring existing software. The existing software assessment phase is comprised of assessment and selection activities. The results of these activities will be used to full-fill the justification of the choice in the software reuse file.

5.3.2 Assessment

After requirement phase, the existing software candidate is assessed for their capability to implement the software requirements, for feasibility and the identification of compensation measures (e.g. delta-qualification) and for their effects (e.g. cost, schedule) The impact of any unneeded features present in the existing software should be assessed as well (assessing the impact of any deactivated code or configurable code, to ensure no harm will be made to the operational software and system, etc.) as previously mentioned in clause 5.2.4.

The following existing software artefacts should be considered for assessment if available (see clause 6.2.7.4 of ECSS-Q-ST-80):

- system requirements and their corresponding models;
- software requirements documentation;
- software architectural and detailed design documentation;
- forward and backward traceability between system requirements, software requirements, design and code;

- unit tests documentation and coverage;
- integration tests documentation and coverage;
- validation documentation and coverage;
- verification reports;
- performance;
- operational performances;
- residual non-conformances and waivers;
- user documentation;
- code quality (adherence to coding standards, metrics)

Additionally, for COTS software (often black box with little information about it) the following aspects should be added to the previous list:

- product availability;
- availability of lifecycle data;
- ease of integration and extent of additional efforts such as glue code, architecture techniques to mitigate impact integration etc.;
- product service history;
- existing software supplier qualifications such as use of standards, history and length of service, technical support etc.;
- configuration control including visibility into COTS software supplier's product version;
- maintenance issues such as patches, retirement, obsolescence and change impact analysis.

The assessment of the potentially reusable existing software should also include the following activities:

- evaluating the qualification status of these existing software towards projects requirements, including reusability, safety and dependability;
- performing a risk analysis on re-using these components (see Annex C of this handbook);
- analysing compliance with the organization's reuse standards;
- analysing consistency with project designs (e.g. software architecture design, database design);

The assessment is performed to evaluate whether the software system can incorporate the existing software, whether design containment techniques to be used will tolerate identified risks or whether the implementation of corrective measures will be so costly that the reused existing software candidate should be rejected. These adaptations and modifications should be well identified when assessing the candidate reusable products.

The assessments will permit to define what evidence is not provided, and to define the associated actions to conform to the requirements (including quality requirements). The assessment will support the definition of the corrective actions required to complement the verification and validation information of the candidate existing software to meet the applicable project requirements.

The corrective actions can use or include one or more of the techniques introduced in clause 7. When reverse engineering is not possible to use and life cycle data from previous development are not available, PSH technique should be used (see clause 6.2.7.8 in ECSS-Q-ST-80). In the existing software qualification plan (see section <8.1> of SRF) these techniques should be listed (see clause 5.5 for the qualification phase).

The assessments will also permit to define, as early as possible, what modifications should be added to the project software (external to the reused one) and their impact for each of the possible options. These set of modifications should be provided to the project for approval and eventual implementation.

The cost of acquisition and analysis and recovery of the evidence, as well as the cost of modification to the current application where the existing software will be integrated, when required, should be estimated for each option.

All this assessment results should be recorded in section <7> of the software reuse file together with an assessment of the possible level of reuse and a description of the assumptions and the methods applied when estimating the level of reuse as required in the clause 6.2.7.5 of ECSS-Q-ST-80. They should be reported for all criticality category levels, as presented in Annex A of this handbook. This information should be provided at SRR and PDR reviews.

The corrective actions should be recorded in section <8> of the software reuse file for all criticality category levels, as presented in Annex A of this handbook, and presented at SRR and PDR reviews.

5.3.3 Selection

The selection is a process based on the results from the assessment activity and on the comparison of existing software assessment results. The selection of the most appropriate existing software candidate should be made based on what was identified during the assessment. Specific criteria for the selection include, but are not limited to, the following aspects:

a. Ability to provide required capabilities and meet required constraints (applicable requirements including non-functional requirements);

b. Compliance with the project requirements applicable to the criticality level of the function provided (see clause 5.2.2);

c. Acceptance (demonstration of correct operation) status and warranty conditions;

d. Interoperability with other systems and system-external elements;

e. Copyright and licensing issues, (see [ESA/REG/002]);

   1. Restrictions on copying/distributing the software or documentation,

   2. License or other fees appropriate to each copy (often complicated to handle when the existing software is open source or public)

f. Maintainability, including:

   1. Likelihood that the software product needs to be changed,

   2. Feasibility of accomplishing that change,

   3. Availability and quality of documentation and source files,

   4. Likelihood that the current version continues to be supported by the supplier,

   5. Impact on the system if the current version is not supported, and likelihood of uncontrolled versions are available (e.g. for COTS software),

   6. Maintenance responsibility,

   7. The maintenance conditions, including the possibilities of changes and upgrades to new releases.

g. Technical, cost, and schedule risks and tradeoffs in using the software product;

h. Conditions of installation, preparation, training and use;

i. Identification and registration by configuration management;

j. Ordering criteria (e.g. versions, options and extensions);

k. Back-up solutions if the product becomes unavailable;

l. Contractual arrangements for the development and maintenance phases;

m. Exportability constraints;

n. Trustability and stability prospects of the existing software supplier and the existing software itself.

The justification of choice should be explained, with the following information:

- the identification of the existing software to be reused;
- a justification of reuse versus development;
- in case of reuse existing software that needs modification, identification of the baseline technical requirements (gap) to which the existing software is partially or non-compliant and a technical description of the modification needed and its impact (e.g. cost, schedule);
- in case of reused existing software that needs additional verification and validation, identification of the baseline quality requirements to which the existing software is partially or non-compliant and a specification of the extra work needed(e.g. update SDP, SVVP, test, analysis).

The justification could include also an analysis or inspection of the existing software known characteristics (e.g. its supplier, any published data), existing industry knowledge or previous experience that the software supplier has with the existing software, etc. in order to demonstrate suitability versus the identified requirements.

For each reused software, an evaluation table should be defined (to be documented in the section <6> of the SRF File) summarizing the collected supplier information and showing the final status of the assessment of each solution. Selection of existing software supplier should be performed based on the comparison of existing software assessment results of each solution as required in ECSS-Q-ST-80 clause 5.4.1.2.

Both the evaluation table and the justification of the choice should be provided in the SRF file at SRR and PDR reviews.

5.4 Integration phase

5.4.1 Overview

The objectives of the integration phase of the approach are to provide support to the integration of the existing software into the target software project taking into account the results of the analysis and assessments performed in the previous phases.

5.4.2 Incoming inspections

The supplier should perform incoming inspections after the existing software becomes available, using corporate or project specific incoming inspection procedures. Incoming inspection reports should be produced and references should be added to the SRF.

5.4.3 Configuration management

The activities associated with configuration management of existing software should consider that:

a. An identification method should be established to ensure that the existing software configuration and data items are uniquely identified.

NOTE The identification method can be based on existing software identification from the existing software supplier and any additional data such as release or delivery date.

b. The software project's problem reporting should include management of problems found in existing software, and a bi-directional problem reporting mechanism with the existing software supplier should be established.

c. The software project's change control process should be established for the incorporation of updated existing software versions. Complications for existing software-based systems can be caused by patches, upgrades, and products that may seem innocuous but can interfere with the functioning and performance of the system for which the configuration management activity is responsible. An impact analysis of changes to the existing software baseline should be performed prior to incorporation of new releases of existing software.

NOTE The list of changes (problem fixes and new/changed/deleted functions) implemented in each new release may be available from the existing software supplier.

d. The software project's archive, retrieval and release should include existing software specific configuration and data items.

Many times, the configuration management policy used by the existing software supplier is not under the control of the current software project management (e.g. COTS software). This requires the project configuration management system to include control of the existing software versions. In addition to traditional configuration management responsibilities, the configuration management baseline for an existing software-based system tracks such things as product slices, version slices, license information, product patches, and dependencies between products (products that are dependent on one another in such a way that a change or upgrade of one is likely to force a corresponding change or upgrade in one or more others); aspects that are not as a rule a part of configuration management for normal software developments.

The configuration baseline of the reuse existing software should be included in the SRF in section <9> for each release (documentation and code should be consistent) and provided to the customer for acceptance.(see ECSS-Q-ST-80 clause 6.2.7.11).

5.4.4 Adaptation of the existing software

5.4.4.1 Changes to existing software

A description of the assumptions and the method of calculate level of reuse, that is, all changes to the existing software made as part of its incorporation in the related software project should be provided by the supplier in the software reuse file. ECSS-Q-HB-80-04 clause A.3.3.25 defines reuse modification rate metric that keeps track of the amount of estimated and/or performed modifications applied to every existing software component. The metric gives a guideline for assessing whether to treat the existing software as entirely new software based on the ratio of modified/added lines of code. If the ratio is greater than an estimated percentage of modifications, then the software is considered as

newly software. No specific target value is identified for the estimated percentage of code modified/added, being considered as strictly project dependant.

The calculation of the effort needed to incorporate the changed existing software baseline should follow the same process as incorporating changes in any software product. This includes but is not limited to:

- Impact on re-verification effort,
- Process effort to implement and verify the existing software upgrade (if selected) into the software application. The assumption made is that the existing software has been previously incorporated into a software application.

The decision to change the baseline version of existing software is based on the same considerations used in the original selection of the existing software. A new accept/reject decision should be made before going forward with the change. It is important to gain access to the existing software supplier's problem reports that were incorporated into the change, as well as the revised software requirements (new features) as a roadmap to incorporation and re-verification of the new version of the application software. It is important to capture any information related to problems fixed in between versions. Any life cycle data pertaining to the existing software should be obtained. The problem reports should be provided, even for COTS software.

Configuration considerations for existing software should ensure baseline and subsequent updates are controlled by the supplier (see clause 5.4.3).

Analysis activities for proposed modifications to existing software should include as a minimum:

a. Review of the outputs of the system and software safety and dependability assessment process taking into account the proposed modifications.

b. Revision of the requirements for reused existing software according to the revised software criticality category of the current software

c. Both the impact of the application software requirements changes and the impact of application software architecture changes should be analyzed, including the consequences of software requirement changes upon other requirements and coupling between several software components that can result in re-verification effort involving more than the modified area.

d. Determination of the area affected by a change. This can be done by data flow analysis, control flow analysis, timing analysis and traceability analysis.

e. Re-verification of the areas affected by the change.

The effects of any changes to existing software should be analyzed with respect to:

- previously accepted qualification package
- interface specifications to other modules (not being re-verified)
- application software requirements
- safety and dependability analysis (e.g. driving tests)
- any other documentation (e.g. concerning architecture and interface design)

Reuse of existing software, especially COTS software, implies specific effort to be planned and dedicated to integrate and test these components. If the reused existing software changes, this implies a required level of effort of suppliers that use that existing software.

5.4.4.2 Change of application or development environment

The use and modification of existing software can involve a new development environment, a new target processor or other hardware, or integration with other software than that used for the original application.

Guidance for change of application or development environment includes:

- The rigor of the evaluation of an application change should particularly consider the complexity and sophistication of the programming language. For example, the rigor of the evaluation for Ada generics is greater if the generic parameters are different in the new application.
- If a different compiler or different set of compiler options are used resulting in different object code, the results from a previous software verification process activity using the object code may not be valid and should not be used for the new application. In this case, previous test results may no longer be valid for the structural coverage criteria of the new application. Similarly, compiler assumptions about optimization may not be valid.
- If a different processor is used then:
    1. The results from a previous software verification process activity directed at the hardware/software interface should not be used for the new application.
    2. The previous hardware/software integration tests should be executed for the new application.
    3. Reviews of hardware/software compatibility should be repeated.
    4. Additional hardware/software integration tests and reviews can be required.
- Verification of software interfaces should be conducted where existing software is used with different interfacing software.

5.5 Qualification phase

This phase can start when the final selected candidate is approved by the customer. The qualification is performed along with the existing software integration and concerns the qualification of the existing software in the frame of the current project.

The objectives of the reuse qualification phase are to provide support to the collection and documentation of verification and validation evidence so as to demonstrate the achievement of the project requirements, including the software product assurance, needs by the selected existing software regarding the software system in which it is integrated and to the same criticality category.

The set of corrective actions identified at gap analysis and assessment activities become the inputs to the qualification plan.

The intention of this qualification stage is to ensure that all existing software utilized in the project provides the same level of confidence as bespoke software developed from scratch. This aims at subjecting all reused existing software to exactly the same verification & validation requirements as the new software of the same criticality is subjected to. Clearly on a commercially available COTS product this is difficult to achieve. The source code is not generally available, there is limited documentation, little evidence that a proper and documented software lifecycle was followed in its development, and no evidence of the kind of verification and validation that the COTS has been subjected to. For this kind of software, based on the reused software phase's outcome described above, and according to the criticality of the functions intended to be implemented by the COTS, a

certification data package may be required as defined in section <10> of the SRF in Annex A of this handbook.

For other kinds of existing software (e.g. previously developed software, developed to other standards) it is as well difficult to assess this software against current SW V&V requirements since they use to be developed to different requirements for documentation, design and development process, testing, PA, configuration management etc. The effort required to align the existing software exactly to the SW requirements of the current project might be difficult and expensive to achieve. Furthermore, achieving total compliance with verification/validation objectives might be impossible without the re-engineering of certain design/development outputs that might not be available, or were simply not required by the original development standard.

With the above considerations in mind, it is clear that especially for highly critical software category levels the reuse of existing software is difficult. Appropriate consideration should be paid to define the reuse qualification plan in the SRF (see section <8.1> in Annex A of this handbook) to fully qualify the existing software to the required SW criticality category, since high criticality of the software demands the same absolute confidence in the existing software as for the newly developed one.

The software supplier should elaborate any evaluation evidence that was not obtained from the existing software supplier. Especially, if existing software proposed for reuse was developed with less rigour than specified by the project standards for the criticality level of its intended use or if the analysis of available data indicates remaining risks, the supplier should provide the evidence of the product's suitability or perform additional verification tasks.

Reverse engineering techniques is a method that can be applied to generate missing documentation and to reach the required verification and validation coverage. PSH method should be used for existing software whose life cycle data from previous development are not available and reverse engineering techniques are not fully applicable.

Methods to support the reuse qualification are provided in clause 7 to be used as required in clause 6.2.7.8 of ECSS-Q-ST-80.

Section <8> of the SRF is dedicated to document this reuse qualification activity. As already mentioned in Annex A of this handbook, Section <8.1> contains the qualification plan that should be always provided for software criticality categories A, B and C. For software criticality category D its delivery should be agreed with the customer on a case by case basis. This qualification plan should be provided at SW-PDR.

Section <8.2> contains the results of the execution of the reuse qualification plan with all corrective actions implemented. This information should be provided for all criticality categories and at PDR, DDR, TRR, CDR, QR.

6
Tool qualification

6.1 Introduction

Tools are often used in projects to simplify, support or automate activities and tasks required for the development, verification or validation of software. But as already required in ECSS-Q-ST-80 tools should be assessed for their appropriateness for each specific project. Compilers, code generators, etc. are example of tools that should be very robust since they even automate development activities directly affecting the quality and robustness of the resulting operational software.

In addition to the applicable project requirements, they should be subject to specific incoming inspections and configuration management requirements as for any existing software (i.e. the ones mentioned in clauses 5.4.2 and 5.4.3 respectively).

Whether a tool needs to be qualified is independent of the type of the tool (development, verification or validation). Tool qualification should be carried out when the results of the tool are relied upon to provide the sole evidence that one or more SW requirements are satisfied. If, instead, the output of the tool is verified by some other means (e.g. manual inspection or review) the tool does not need to be qualified.

A tool needs qualification if the answer to all of the three following questions is "Yes":

- Can the tool introduce an error into the software, or fail to detect an existing error?
- Will the output of the tool not be later verified as defined in the requirements of the current project?
- Are processes of the current project eliminated, reduced or automated by the use of the tool? To be more precise, will the output of the tool be used to either meet or replace an objective (requirement/process) of the current project?

6.2 Tool qualification level

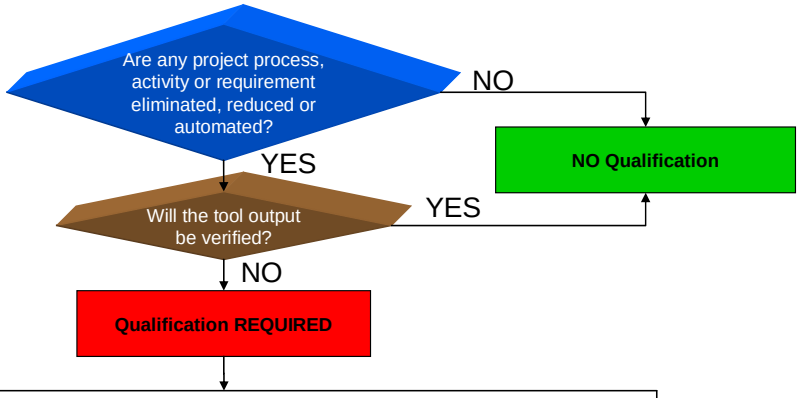
Qualification of a tool is needed when processes or activities of the project requirements and standards are eliminated, reduced or automated by the use of a software tool without its output being verified. The objectives of the tool qualification process is to ensure that the tool provides confidence at least equivalent to that of the process(es) eliminated, reduced or automated.

As said above, if the results of the tool are relied upon to provide the sole evidence that one or more SW requirements are satisfied, the tool requires qualification. If instead the output of the tool is verified by some other means (e.g. manual inspection or review) the tool does not need to be qualified

Clause 6.2 provides guidance for the level of confidence needed for the different tools and development environment depending on the amount of potential danger to the software under

production as well as on the criticality level of that software (the higher the criticality level the more confidence is needed on the tool).

Tool qualification needs depend on the software life cycle process or activity impacted by the use of the tool. It will determine the amount of confidence required for the tool. Based on this, different 'Tool Qualification Levels' (TQL) will be defined following the logic presented in Figure 6-1 (heavily based on [DO178]):



Software criticality category	Criteria		
	A tool whose output is part of the operational software and thus could insert an error	A tool that automates verification or validation process(es) and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of: • verification or validation process(es) other than that automated by the tool, or • development process(es) which could have an impact on the operational software	A tool that, within the scope of its intended use, could fail to detect an error
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

Figure 6-1: Tool qualification levels

The impact of the tool on the quality of the software application to be produced using that tool should be assessed in order to determine its tool qualification level. The following tool assessment criteria should be used:

1. A tool whose output is part of critical software and thus could insert an error.
2. A tool that automates testing, verification or validation process(es) or activities and thus could fail to detect an error, and whose output is used to justify the elimination or reduction of:
   - testing, verification or validation process(es) or activities other than that automated by the too, or
   - engineering process(es) which could have an impact on the application software.
3. A tool that, within the scope of its intended use, could fail to detect an error.

When tools are involved in the production of the executable software product, tool qualification should be applied. The tool qualification level (TQL) should satisfy the same product assurance and engineering requirements as those that would apply to the software developed according to the project standards, when:

1. the project requirements or the applicable [ECSS-E-ST-40] or [ECSS-Q-ST-80] activities are reduced, automated or eliminated and
2. the output of the tool is not verified nor validated

Five tools qualification levels are identified: from TQL 1 (for tools that need the most stringent qualification activities) to TQL 5 (less rigorous tool qualification level).

An example of a tool not needing to be qualified is a test case generator tool, when its outputs are reviewed to ensure that a certain test coverage goals to be achieved.

Following this guideline therefore, it is possible to determine also, for example, that a software compiler will in many cases not need to be qualified (in the tool sense) since a) its output (object code/executable) is subjected to numerous verification requirements and b) it is not automating any specific process of the current project. However, if for example a feature of the compiler is used with particular switches/options in order to enforce certain coding-standard rules (e.g. identify violations of the rule to use variable names uniquely in the source code), the compiler is here being used to automate a specific verification objective of the current project which, if not verified by some other means, requires that the compiler be "qualified" for the verification task that it is performing.

Compilers also need to be considered in terms of the run-time libraries that are linked into the final executable software product, since these inherently can affect the quality of the operational software and that cannot necessarily be specifically verified through the normal verification/validation.

6.3 Tool qualification

Based on the qualification level assigned, the qualification plan for a tool should be established depending on: the processes or activities eliminated, reduced or automated and whether the tool outputs are verified and/or validated as requested in the project applicable standards; and the criticality category level of the operational software.

The following tables provide the different methods to be used in combination to qualify the tool (when reverse engineering cannot or is not feasible to be used to qualify the tool) depending on the assigned tool qualification level (based on what is provided in [ISO FDIS 26262]).

Table 6-1: Example of combination of classes of methods

Method for TQL-1	CRITICALITY CATEGORY			
	A	B	C	D
Development in compliance with safety and dependability related requirements or standard	HR	0	0	0
Validation(i.e. testing) of the software tool	HR	0	0	0
Development process examination	R	0	0	0
Product service history	0	0	0	0

Method for TQL.2	CRITICALITY CATEGORY			
	A	B	C	D
Development in compliance with safety and dependability related requirements or standard	0	R	0	0
Validation (i.e. testing) of the software tool	0	HR	0	0
Development process examination	0	HR	0	0
Product service history	0	HR	0	0

Method for TQL-3	CRITICALITY CATEGORY			
	A	B	C	D
Development in compliance with safety and dependability related requirements or standard	0	0	R	0
Validation (i.e. testing) of the software tool	0	0	HR	0
Development process examination	0	0	HR	0
Product service history	0	0	HR	0

Method for TQL-4	CRITICALITY CATEGORY			
	A	B	C	D
Development in compliance with safety and dependability related requirements or standard	R	R	0	R
Validation (i.e. testing) of the software tool	HR	R	0	R
Development process examination	HR	HR	0	R
Product service history	HR	HR	0	R

Method for TQL-5	CRITICALITY CATEGORY			
	A	B	C	D
Development in compliance with safety and dependability related requirements or standard	R	R	R	R
Validation (i.e. testing) of the software tool	R	R	R	R
Development process examination	HR	HR	R	R
Product service history	HR	HR	R	R

KEY: HR: Highly recommended;
R: Recommended
0: Not to be applied.

A combination of the above methods should be used and always previously agreed with the customer and should be presented for the approval by the customer. The HR ones not proposed when applicable should be explicitly justified.

For tools involved in the production of the executable software product following above criteria 1, and for the upper criticality categories, a comparison between the software development standards applied during the development of the reused tools and the software standards applicable to the current project should be performed and documented. This comparison allows focusing on the qualification activities to be done. The qualification plan should cover the missing specification, design, implementation, integration, verification and validation activities for the tool as defined in the project standard for the specific software criticality category level of the application software in which the tool is involved.

Instead, the qualification plan for tools involved in the production of the application software falling under criteria 2 and 3 of clause 6.2 should focus more on a set of tests to demonstrate compliance under abnormal and normal operational conditions with the functional and operational requirements of the tool. These tests should aim at showing that all the tool requirements have been verified and/or validate. The complexity of these tests will vary depending on the complexity of the tool, its purpose and how it is used.

Guidelines for these methods and techniques can be found in section 7 of this handbook.

A risk analysis should be also conducted, taking into account the planned use of the tool and the planned qualification activities to be conducted, in order to assess the effect of faults of the tool, the effect of errors introduced by users of the tool, the effect of the use of the tool non conformant to the specified use.

The supplier should also perform an analysis of the tool's pedigree focusing on:

- Tool's reputation and usage examples in industry,
- Provider reputation and certification status (if any),
- Quality standards used by the provider in the development of the tool,
- References to previous experience with the tool, problems encountered etc.

Furthermore, the supplier should take the following aspects into account:

- evolution of the tools in relation to the tools that use the generated code as an input (e.g. compilers, code management systems),
- customization of the tools to comply with project standards,
- portability requirements for the generated code,
- collection of the required design and code metrics,
- verification of software components containing generated code,
- configuration control of the tools including the parameters for customization.

Tool qualification activity should be documented in section <8> of the SRF in Annex A. The qualification activities are requested to be completed before the tool is used for its intended purpose in the operational software development phase. This should generally be possible early in the project lifecycle and then qualification activities are requested to be completed before the SW PDR (as required to be demonstrated in requirement 5.6.1.2-2 of ECSS-Q-ST-80). However, there can be particular cases where it is difficult to qualify the tool in a stand-alone fashion without the application code being present, requiring deferral of qualification phase to coincide with the phase in which the tool is actually used operationally – such cases should be justified appropriately.

Section <8.2> of the SRF contains the results of the execution of the tool qualification plan with all TQL related actions implemented. It should be provided at SW-PDR (exceptionally to be completed before the tool is used for its intended purpose).

7 Techniques to support qualification when reusing existing software

7.1 Introduction

Software life cycle data from the previous development should be evaluated to ensure that the software verification and validation process objectives of the software criticality level are satisfied for the new application. The objectives of this qualification phase are to provide support to the collection and documentation of needed verification and validation evidences.

The following clauses detail activities, best practices and methods that help to achieve the (delta)-qualification when reusing existing software.

NOTE 1 The activities when reusing supporting tools (instead of existing software as part of the final software product) are described in clause 6.

NOTE 2 Most of the techniques described in this clause are not specific of the reuse of existing software. They are briefly mentioned here as a reminder.

In addition, clause 7.3 defines some practices and methods for the implementation of the derived requirements intended to prevent adverse effects of any needed or unneeded functions of any existing software. This can result in design techniques around the reused product: isolation, etc. as well as other software fault tolerant techniques implemented in the system software where the existing software will be integrated.

Usually, the reuse verification and validation processes identify verification and validation objectives that cannot be met using traditional means. This clause provides few different methods and techniques that can be used to document and support the qualification process of a project reusing existing software. They are grouped as follows:

- Verification techniques
- SW design techniques
- HW architectural techniques
- Reverse engineering
- Product service history
- Development process examination

7.2 Verification techniques

7.2.1 Black box techniques

The following techniques can be used when the source code is not available:

a. Functional testing

The objective of functional (or black box) testing is to reveal failures during the specification and design phases and to avoid failures during implementation and the integration of software and hardware. During the functional tests, reviews are carried out to see whether the specified characteristics of the software have been achieved. The software is given input data which characterizes the normal expected operation. The outputs are observed and their response is compared with that given by the specification. Deviations from the specification and indications of an incomplete specification are documented

b. Performance testing

The purpose is to verify proper component timing or optimize response times for the best throughput by verifying time under varying loads, throughput, memory utilization, and many other real-time properties.

c. Software fault tree analysis

Software fault-tree analysis is simply the application of static fault-tree analysis to software. Its purpose is assesses the causal relationship between events in a process.

d. Software failure mode and effects analysis

The basic idea is to assess a software product in terms of the services provided by its components, which can be existing software components or bespoke. This allows abstraction away from lower level failures for example a run time error in software can cause: the loss of a service (Omission); the service to do something it shouldn't (Commission); corruption of output data; and even late or early delivery of a service. The exact cause of a service failure is not of interest it is its consequence with respect to system dependability and safety.

e. Interface testing

The objective is to detect errors in the interfaces of subprograms. Several levels of detail or completeness of testing are feasible. The most important levels are tests for:

— all interface variables at their extreme values;

— all interface variables individually at their extreme values with other interface variables at normal values;

— all values of the domain of each interface variable with other interface variables at normal values;

— all values of all variables in combination (this is only feasible for small interfaces);

— the specified test conditions to each call of each subroutine.

These tests are particularly important if the interfaces do not contain assertions that detect incorrect parameter values. They are also important after new configurations of pre-existing subprograms have been generated

f. Boundary value analysis

Its objective is to remove software errors occurring at parameter limits or boundaries. The input domain of the program is divided into a number of input classes. The tests should cover the boundaries and extremes of the classes. The tests check that the boundaries in the input domain

of the specification coincide with those in the program. The use of the value zero, in a direct as well as in an indirect translation, is often error-prone and demands special attention: zero divisor; blank ASCII characters; empty stack or list element; full matrix; zero table entry.

In general, the boundaries for input have a direct correspondence to the boundaries for the output range. Test cases should be written to force the output to its limited values. Consider also, if it is feasible to specify a test case which causes the output to exceed the specification boundary values.

g. Software fault injection

The goal is to insert errors (one error at time) to test the effectiveness of the fault tolerance technique added to the software. Fault inject allows to test the fault tolerance technique implemented.

h. Stress testing

The objective is to exercising the software in order to verify its behaviour and response under stress conditions. There are a variety of test conditions which can be applied for avalanche/stress testing. Some of these test conditions are:

— if working in a polling mode then the test object gets much more input changes per time unit as under normal conditions;

— if working on demands then the number of demands per time unit to the test object is increased beyond normal conditions;

— if the size of a database plays an important role then it is increased beyond normal conditions;

— influential devices are tuned to their maximum speed or lowest speed respectively;

— for the extreme cases, all influential factors should be put to the boundary conditions at the same time.

Under these test conditions the time behaviour of the test object can be evaluated. The influence of load changes can be observed. The correct dimension of internal buffers or dynamic variables or stacks can be checked.

i. Response time and memory constraints

The objective is to ensure that the software system meets its temporal and memory requirements. The requirements specification for the system and the software includes memory and response requirements for specific functions, perhaps combined with constraints on the use of total system resources. An analysis is performed to determine the distribution demands under average and worst case conditions. This analysis requires estimates of the resource usage and elapsed time of each software system function. These estimates can be obtained in several ways, for example comparison with an existing software system or the prototyping and benchmarking of time critical software.

7.2.2 White box techniques

White box evidence should be obtained when black box evidence is not considered sufficient to fulfil the requirements relevant to the current the software criticality level. White box evidences should be used in:

- Software that is non-deterministic
- Software products where reliability modelling is unfeasible
- Time-critical software products

- Control systems where the algorithm and implementation are not known
- New components, without a track record
- Software products where it is complicated to establish the configuration and assess the impact of differences

The following white box techniques can be used when source code is available:

a. Structured based testing

The goal is to apply tests which exercise certain subsets of the program structure. Based on analysis of the program, a set of input data is chosen such that a large (and often pre-specified target) percentage of the program code is exercised. Measures of code coverage vary, depending upon the level of rigour required: statements, branches, modified conditions and so.

b. Control flow analysis

The goal is to detect poor and potentially incorrect program structures. Control flow analysis is a static testing technique for finding suspect areas of code that do not follow good programming practice. The program is analysed producing a directed graph which can be further analysed for inaccessible code, for instance.

c. Data flow analysis

The objective is to detect poor and potentially incorrect program structures. Data flow analysis is a static testing technique that combines the information obtained from the control flow analysis with information about which variables are read or written in different portions of code. The analysis can check for:

— variables that can be read before they are assigned a value – this can be avoided by always assigning a value when declaring a new variable.

— variables that are written more than once without being read – this can indicate omitted code.

— variables that are written but never read – this can indicate redundant code.

A data flow anomaly are not always directly correspond to a program fault, but if anomalies are avoided the code is less likely to contain faults.

d. Code inspections

The objective is to reveal discrepancies between the code and its requirements (including coding standard rules, etc). The code is examined and evaluated to ensure that it conforms to the requirements given in the specification and in other applicable documents and rules. Any points of doubt concerning the implementation are documented for later resolution. In contrast to a walkthrough, the author is passive and the inspector is active during the inspection procedure.

e. Formal proof

The objective is to prove the correctness of a program without executing it, using theoretical and mathematical models and rules. A number of assertions are stated at various locations in the program, and they are used as pre and post conditions to various paths in the program. The proof consists of showing that the program transfers the preconditions into the post-conditions according to a set of logical rules, and that the program terminates

f. Complexity metrics

The objective is to predict the attributes of programs from properties of the software itself or from its development or test history. These models evaluate some structural properties of the

software and relate this to a desired attribute such as reliability or complexity. Software tools are used to evaluate most of the measures.

g. Audits and inspections

The objective is to reveal mistakes and faults in all phases of the software development. Audits and inspections are a means by which one can determine if a process has been performed. Reviews of existing software data in conjunction with Software Quality Assurance (SQA) records substantiating the process applied to these software data can help satisfy these objectives. In some cases, a traditional inspection of the data itself is needed. In other cases, the alternative method of reviewing other evidence of the process activities can help to satisfy these objectives.

h. Walkthrough/design review

The objective is to detect faults in some products of the development as soon and economically as possible. Formal design review is conducted for all new products/processes, new applications, and revisions to existing products and manufacturing processes which affect the function, performance, safety, reliability, ability to inspect maintainability, availability, ability to cost, and other characteristics affecting the end product/process.

A code walkthrough consists of a walkthrough team selecting a small set of paper test cases, representative sets of inputs and corresponding expected outputs for the program. The test data is then manually traced through the logic of the program.

i. Scheduling analysis

The objective is to predict the run-time behaviour of a pre-emptive, priority-based concurrent software product.

Scheduling analysis techniques allow determining if the defined deadlines of a real-time are met. The application is analysed in order to verify if it meets the requirements under the highest load conditions.

j. Model-based verification

The objectives of the model-based verification technique are to demonstrate some given properties through the elaboration of models. Model based techniques covers a wide range of technical approaches to software development and assessment. Its use ranges from requirements analysis through system specification, techniques for "correct" construction of software, and of major concern here, to analytic techniques for demonstrating properties of systems or software.

k. Prototyping

The objective is to check the feasibility of implementing the software product against the given constraints. A subset of software functions, constraints, and performance requirements are selected. A prototype is built using high level tools. At this stage, constraints such as the target computer, implementation language, program size, maintainability, reliability and availability need not be considered. The prototype is evaluated against the customer's criteria and the requirements may be modified in the light of this evaluation. Software prototyping has many variants. However, all the methods are in some way based on two major types of prototyping: Throwaway Prototyping and Evolutionary Prototyping.

The main goal when using evolutionary prototyping is to build a very robust prototype in a structured manner and constantly refine it. The reason for this is that the evolutionary prototype, when built, forms the heart of the new system, and the improvements and further requirements will be built. When developing a system using evolutionary prototyping, the system is continually refined and rebuilt. This technique allows the development team to add

features, or make changes that couldn't be conceived during the requirements and design phase.

Throwaway prototyping is the most commonly used in the assessments of reuse existing software. Throwaway or rapid prototyping refers to the creation of a model that will eventually be discarded rather than becoming part of the final delivered software. After preliminary requirements gathering is accomplished, a simple working model of the system is constructed to visually show how requirements look like when they are implemented into a finished system. Rapid prototyping involved creating a working model after a relatively short investigation. The method used in building it is usually quite informal, the most important factor being the speed with which the model is provided. The model then becomes the starting point from which clarifies requirements. When this has been achieved, the prototype model is 'thrown away', and the system is formally developed based on the identified requirements. The most obvious reason for using throwaway prototyping is that it can be done quickly. If the engineers can get quick feedback on their requirements, they are able to refine them early in the development of the software. Making changes early in the development lifecycle is extremely cost effective since there is nothing at that point to redo. If a project is changed after a considerable work has been done then small changes could require large efforts to implement since software systems have many dependencies. Speed is crucial in implementing a throwaway prototype, since with a limited budget of time and money little can be expended on a prototype that will be discarded. Throwaway prototyping is also strength for its ability to construct interfaces that can be test.

There are many advantages to using prototyping in software development – some tangible, some abstract:

— Reduced time and costs: Prototyping can improve the quality of requirements and specifications provided to developers.

Using, or perhaps misusing, prototyping can also have disadvantages:

— Insufficient analysis: The focus on a limited prototype can distract developers from properly analyzing the complete project. This can lead to overlooking better solutions.

— Developer attachment to prototype: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture.

— Excessive development time of the prototype: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, it can happen that they try to develop a prototype that is too complex

Table 7-1 shows an example of the combinations of classes of methods that are likely to provide the confidence that the existing software complies against each of the requirements and characteristics mentioned above.

Table 7-1: Example of combination of classes of methods

Requirements	Black Box Technique	White Box Technique
Nominal Functionality	Functional testing Interface testing	Design Review Data flow analysis Control flow analysis
Exceptional Functionality	Stress Testing Boundary value analysis/testing Black box testing	Structure based testing Design Review Code inspection Fault injection
Algorithms sufficiency	Stress Testing Boundary value analysis/testing Statistical testing Data recording and analysis	Design review Code inspection Structured based testing Formal proof
Timing	Performance testing Response time and memory constraints Stress testing	Design review Code inspection Scheduling analysis Structured based testing
Memory usage	Stress testing Response time and memory constraints	Design review Code inspection
Availability	Software FMECA Software FTA	Design review Structured based testing
Functional independence	Impacts analysis Boundary values analysis/testing Equivalence-class testing Regression testing	Design review Code inspection Scheduling analysis Structure based testing
Security	Interface testing Stress testing Data recording and analysis	Design review Code inspection Data flow analysis Control flow analysis
Robustness	Stress Testing Fault injection	Design review Code inspection Structured based testing Complexity metrics
Safety	SFMECA SFTA Stress testing Fault injection	Design review Code inspection Structured based testing Audits/inspections

7.3 SW design techniques

Architectural means can be employed to limit the effect of a software error leading to a system-level failure. Unfortunately, architectural means are not always an option to protect against latent errors in software. It can also be that the architectural means actually increases the system's complexity. The following list provides architectural means for limiting the effect of an existing software error and fault tolerance techniques to be used for reusing existing software:

a. Error detecting and correcting codes

   The objective is to detect and correct errors in sensitive information.

b. Safety bag

   The objective is to protect against residual specification and implementation faults in software which adversely affects safety. In this technique, an external monitor, called a safety bag, is implemented on an independent computer using a different specification. The primary function of the safety bag is to ensure that the main system performs safe operations. The safety bag continually monitors the main system to prevent it from entering an unsafe state. If a hazardous state does occur, the system is brought back to a safe state by either the safety bag or the main system.

c. SW partitioning

   The objective of this approach is to prevent error propagation from low trustable and low criticality components to higher criticality components. It then allows the elaboration of a system with components and functions at various criticality levels, without having to use equally trustable components (i.e., at the highest level).

   The partitioning approach consists in architectural design choices at system or global software level, providing isolation of components of different levels of criticality, and providing protection against error propagation between these levels. In practice the approach consists in a combination of physical (hardware isolation) and logical (software isolation): separate circuits, boards, memory management units, software layers with monitoring of software interactions, etc.

   Partitioning provides, on the same resource: (1) independence of software components and processes, and (2) error containment. Partitioning enables the following development process objectives:

   — Supports the implementation of different software processes, components, or functions on the same processor.

   — Supports the implementation of different software processes, components, or functions of different software levels on the same computation resources (e.g., processors).

   — Supports software updates or modifications without re-verification of unmodified software partitions.

   — Isolates and protects a non-modifiable, partitioned software component from a user-modifiable software component that is not under the control of the original developer.

   — Isolates and protects an approved software component from unapproved software components.

   The guidelines of robust partitioning are:

   — A software partition should not be allowed to contaminate another partition’s code, I/O, or data storage areas.

   — A software partition should be allowed to consume shared processor resources only during its period of execution.

— A software partition should be allowed to consume shared I/O resources only during its period of execution.

— Failures of hardware unique to a software partition should not cause adverse effects on other software partitions.

— Software providing partitioning should have the same or higher software level than the highest level of the partitioned software applications.

The main purpose of the verification of the partitioning mechanism is to demonstrate that:

— No errors of any software component can contribute to misbehaviour or failure of any other outside the partition.

— No violation of assignment (e.g., overlapping) of the shared resources can be accepted.

Where partitions are used to isolate different software levels, the partition mechanism is verified to the objectives of the highest software criticality level.

Elements of partitioning criticality demonstration can be validated by exercising the partitioning mechanisms using special test scenarios, simulations, or analysis techniques. Test scenarios are written to stimulate the partitioning mechanism by injecting errors or violation attempts to bypass time and space constraints. An example of additional analysis would be the calculation of worst-case execution times to validate temporal performance.

A verification test suite (containing normal range test cases and abnormal or out of range test cases for all requirements of the partitioning mechanism) is established.

Robustness of the partitioning mechanism can be demonstrated by use of a requirements-based test suite (satisfying requirements-based test coverage).

d. Wrappers

The objectives of the wrapping technique are:

— to provide some isolation between a software component (here, the considered reused existing software) and the rest of the system, and consequently improve the portability, interoperability and maintainability (particularly the capability to evolve);

— to provide some protection with respect to the fault occurrence (or activation) and error propagation, through the filtering of inputs and outputs of a given software component.

Trying to detect all possible faults leading to the corruption of upper layer software is a difficult issue and certainly almost impossible as far as off-the-shelf software components are considered. Anyway, preventing such faults to impact the upper layers can only be achieved by external mechanisms. This claim leads to the notion of *wrapper*, originally introduced to deal with security concerns. Two main types of wrappers can be distinguished: *filter* (screening some information flows at a very low level), and *proxy* (front-end package controlling the interactions).

The wrapping technique can also be used in order to check component's functionality. One can distinguish input and output wrappers, the former preventing certain inputs from reaching the component, the latter verifying constraints on the output before releasing it. Although filters and proxies can be used for wrapping both inputs and outputs, clearly the verification of constraints relates to the notion of proxy, since these checks are performed at a semantic level. Input wrappers can be seen as filters of some syntactically incorrect invocation pattern.

The wrapping technique can also be based on the notion of semantic verification. It then relies on a model of the expected behaviour of the executive functional classes. Both inputs and outputs impact the behavioural model on which constraints are verified at runtime. The target component and the level of abstraction needed to obtain a realistic model. These specifications can lead to the development of a runtime model checker or of executable assertions. The

wrapper can thus be seen as a monitor of the underlying component, providing extended error detection features for errors due to both internal software faults and incorrect calls, which are not detected by the internal checks of the component. The performance penalty introduced essentially relies on the granularity of the model and the complexity of the executable assertions.

The wrapping technique consists in the development and validation of a software layer around a software component, according to various specifications. These specifications can cover functional issues (to adapt the functions of a component according to the needs), interface issues (to adapt the interfaces and make them compliant to a given standard), and dependability issues. The latter point relies on a combination of input filtering (avoid erroneous activation; enforce an agreed subset of functions) and output filtering (detect errors; possibly provide some form of fault tolerance – degraded acceptable safe behaviour). The wrapper itself consists in specific software, developed and validated within the project following the corresponding processes and activities (with, due to its nature, a strong focus on integration and validation: integration with the reused existing software, and integration and validation within the global software product and system).

e. Recovery block programming

The objective is to increase the likelihood of the program performing its intended function. Several different program sections are written, often independently, each of which is intended to perform the same desired function. The final program is constructed from these sections. The first section, called the primary, is executed first. This is followed by an acceptance test of the result it calculates. If the test is passed then the result is accepted and passed on to subsequent parts of the system. If it fails, any side effects of the first are reset and the second section, called the first alternative, is executed. This too is followed by an acceptance test and is treated as in the first case. A second, third or even more alternatives can be provided if desired.

Recovery block programming is a technique where independently written modules check themselves for correctness. The technique applied to the reusable existing software would be to isolate the reusable existing software in a module and, prior to exit, assess the results for any error. If the module detects an error, another module is instantiated, which cleans any side effect from the reusable existing software encapsulated module and proceeds to operate error free. A concern of course is the ability of the module to properly assess its health and recover in a safe manner. The technique can be extended from modules to programs to subsystems if the system requires those levels of redundancy (see [FAA-COTS])

f. Retry fault recovery

The objective is to attempt functional recovery from a detected fault condition by re-try mechanisms. In the event of a detected fault or error condition, attempts are made to recover the situation by re-executing the same code. Recovery by re-try can be as complete as a reboot and a re-start procedure or a small re-scheduling and re-starting task, after a software time-out or a task monitoring action. Re-try techniques are commonly used in communication fault or error recovery, and re-try conditions can be flagged from a communication protocol error (e.g. checksum) or from a communication acknowledgement response time-out (see [IEC61508]).

Fault recovery via retry is often used by communication-related systems and is not a common technique for fast real-time systems. The system monitors itself for a fault and will reset itself to a previous safe state and continue forward. If used in a real-time-related system, assurances need to be made that the recovery will be able to be completed before the fault can externally manifest itself at the system level (see [FAA-COTS]).

g. Graceful degradation

The objective is to maintain the more critical system functions available, despite failures, by dropping the less critical functions. This technique gives priorities to the various functions to be

carried out by the system. The design ensures that if there are insufficient resources to carry out all the system functions, the higher priority functions are carried out in preference to the lower ones. For example, error and event logging functions can be lower priority than system control functions, in which case system control would continue if the hardware associated with error logging were to fail. Further, should the system control hardware fail, but not the error logging hardware, then the error logging hardware would take over the control function.

This is predominantly applied to hardware but also to the total system. It is taken into account from the top-most design phase (see [IEC61508]).

h. Restriction of functionality

The concept "restriction of functionality" involves restricting the use of the existing software to a subset of its functionality. The 'restriction of functionality' of the existing software can be carried out by methods such as, but not limited to, run-time checks, design practices (e.g., company design and code standards), or build-time restrictions. The restricted set of functional requirements can then become the basis for use of the existing product.

This restriction of functionality method makes it possible to show compliance to project requirements objectives, particularly by reducing verification and auditing activities to the restricted subset of functional requirements. The mechanisms by which the restrictions are enforced may need additional verification.

7.4 Hardware architecture techniques

Today, the complexity of building software and hardware intensive systems has certainly grown. Indeed, neither can be developed independently and, as such, to build a system with high criticality requires both to exist in a symbiotic relationship. This fact should be kept in mind when considering either hardware- or software-based architecture schemes, because, in many instances, a combination of hardware and software efforts are needed to complete the accommodation of the dependability and safety requirements. Some hardware architectural techniques in use today are listed below. The applicability of these techniques to COTS components is necessarily established depending on their usage:

- System Partitioning: A technique where the safety-related portions of the system are built with high fidelity and separated from the rest of the system.
- Memory Partitioning: A technique to separate data or code such that one group of address spaces do not have an adverse effect on another group of address spaces.
- Time Partitioning: This is a technique used to separate the effects of the operation of one portion of a program on the timing behaviour of another portion of the program.
- Network Separation: A scheme to separate safety-critical portions of a system by limiting all communication between components with the use of packetization of messages (components communicate with each other by message passing, each of which has its own local memory) versus the use of communication via shared memory space.
- N-Redundant Components (hardware): N-redundant components are a way to accommodate faults in a system whereby the fault in one component is accommodated by another component redundant in functionality. Dissimilar, independent designs utilize this technique. This class of architecture can possibly discover random physical failures and design errors. The design should assure the independence of requirements, algorithm, data, and other potential sources of design error.

- Watch-Dog Timers: Watch-dog timers are components that require regular attention from the operational software to prevent a predefined recovery or shutdown action. Watchdogs are used to be triggered at regular intervals to validate the status of a system or subsystem and to activate a predefined action if not. A watchdog is a computer hardware or software timer that triggers a system reset or other corrective action if the main program, due to some fault condition, such as a hang, neglects to regularly service the watchdog (also referred to as "kicking the dog" or "waking the watchdog"). The intention is to bring the system back from the unresponsive state into normal operation. Watch-dog timers can be dedicated hardware components or a separate computer and associated software designed for this task (see [FAA-COTS]).

7.5 Reverse engineering

Reverse engineering is the process of generating software artefacts at a higher level of abstraction, starting from existing software, usually available in the form of source code or object/executable code.

Reverse engineering should be considered when the software proposed for reuse does not meet the current project requirements concerning documentation, V&V and/or quality, and the developers of the existing software are not in the condition to generate the missing information following a direct development approach. This can happen for a number of reasons, e.g. the software is open-source, or the organization that generated the software does not exist anymore.

An example could be the case where a software library or an operating system is proposed for reuse in a project, but only the source code and user manuals are available, while all the other life cycle documentation is missing, and the extent of verification and validation applied in previous applications of the software is unknown.

In such a case, reverse engineering could be employed as a means to provide an unambiguous and complete definition of what the software does and to verify to a certain level of confidence that the software adheres to its specification. To this end, an analysis of the source code could lead to the definition of the software architectural and detailed design, while the user manuals could be used, with the support of the design, to establish the software requirements, i.e. what the software is expected to do, including performance requirements.

The extent of the delta documentation (w.r.t. the one available) to be generated from reverse engineering is driven by the project requirements, namely the criticality of the function implemented by the software and thereby the engineering product assurance requirements applicable to that criticality category.

The documentation generated through reverse engineering can be used to design and perform different kinds and level of tests (unit, integration, validation; functional, interface, robustness), with specific test coverage goals, in accordance with project requirements, suitable to provide confidence that the reused software can be successfully integrated within the newly developed software.

The use of reverse engineering is subject to interpretation on many points and can not be relied upon to always reproduce the original data, if any. It should be noted that, as an alternative method, it is unlikely to provide a complete mapping to the project requirements. This is particularly true for artefacts which are at a level of abstraction much higher than the available form of the existing software (e.g. software requirements derived from executable code). In deciding the extent of reverse engineering, an evaluation should be made of the development process of the existing software, in particular its general adequacy and conformance to the requirements of the current project. The reverse engineering activities should concentrate on the areas most notably inadequate in satisfying the objectives of the project. Specific issues to be addressed should include:

a. Conformance of the development process of the existing software to the software development standards of the current application - Audit.

b. The methods and tools used during the development of the existing software.

c. The extent of the V&V activities conducted on the existing software.

d. The adequacy of the documentation for the existing software.

However, there can be cases where retrospective application of project requirements in its entirety to the existing software are not practicable. In such cases a reduction of the amount of reverse engineering activities can be done if such a reduction is justified from, for example, the dependability and safety analysis, or taking into account the product service history evidence.

Reverse engineering is expensive. The application of reverse engineering is often limited due to the limitations of the techniques, the need of specialised tools or legal limitations. Any project finding itself in the position to resort to such technique in order to make an existing software item suitable for its use in the current (and any future) application should carefully consider new development as an alternative.

7.6 Product service history

This technique is often applied when reusing existing software since it includes information available from the use of the existing software in real operations. Product Service History (PSH) is beneficial for the gain in confidence on the reused software dependability. Its benefits for safety aspects is however not ensured since system safety should be demonstrated for each system in particular, at system level and cannot be demonstrated by any of its subsystems in isolation.

Product Service History is the utilization of the information about previous in-service experience of the component that is relevant to the new intended application and that can constitute evidence of product dependability.

Annex B of this handbook provides an example of the information to be provided in the PSH file.

Product service history should only be taken into account as evidence of the dependability of the existing software where reliable data exists relating to in-service usage and failure rates.

In addition, more credit is given if the product was used in an application with the same or a higher criticality software level. Previously assessed existing software should be considered to complement the product service history file.

The acceptability of this method depends on:

- Effective configuration management of the existing software and a demonstrably effective problem reporting system. In particular, both the software and the associated service history evidence should have been under configuration management throughout the software's service life.
- Configuration changes during the software's service life should be identified and assessed in order to determine the stability and maturity of the software and to determine the applicability of the entire service history data to the particular version to be incorporated in the current software product. Operating experience data derived from extensive usage of the same version of a product in similar applications can indicate that the product is acceptable for the intended application. Nevertheless, several issues with this assumption should be considered. First, configuration data of the actual version used in the problem report supplied can be difficult to obtain. As such, the statistical validity of the data is unknown.

- The problem reporting system for the existing software should be such that there is confidence that the problems reported incorporate all software faults encountered by users, and that data is recorded from each problem report to enable judgements to be made about the severity and implications of the problems. Error-reporting databases commonly span multiple releases and configurations of a given product. Another issue is that circumstances surrounding the occurrence, monitoring, and recording of failures are often vaguely reported. Even the highest critical software level systems have difficulty in reproducing and diagnosing problems due to the inability to recreate the problem from the information provided. Indeed, some of these activities are not only badly controlled, but in fact, COTS vendors limit the publication of negative experiences, particularly if the COTS product was not originally intended for highly critical systems.
- The operating environments of the existing software should be assessed to determine their relevance to the proposed use in the new application.
- Quantified error rates and failure probabilities should be derived for the existing software, taking into account:
    1. length of service period;
    2. the operational in-service hours within the service period, allowing for different operational modes and the numbers of copies in service;
    3. definition of what is counted as a fault/error/failure.

The suitability of the quantified methods, assumptions, rationale and factors relating to the applicability of the data should be justified in the section <11> of the Software Reuse File in Annex A.

In order for Product Service History of the existing software intended to be reused to provide any degree of objective evidence of integrity, the following attributes should be evaluated in determining the credit that can or cannot be granted for service history [FAA-DOT-Handbook]:

— Service duration length;

— Change control during service;

— Proposed use versus service use;

— Proposed environment to service environment;

— Number of significant modifications during service;

— Number of software modifications during service;

— Number of hardware modifications during service;

— Error detection capability;

— Error reporting capability;

— Number of in-service errors;

— Number of services outages;

— Flawless service duration length;

— Service outage length; and

— Amount and quality of service history data available and reviewed

The value of Product Service History information can be less than it appears initially. This is mainly due to the difficulty of comparing previous use with intended use. Analysis should be used to

determine whether the (often seemingly small) changes in use that occur with reuse have a significant impact on dependability and safety.

However, Product Service History arguments correctly used and understood should give weight in the dependability demonstrations – they are the equivalent of testing in which no control has been exercised over the choice of test runs. For example, they can be helpful in narrowing the focus and scope of fresh assessment activities. Their precise value is hard to determine, because it is dependent on the quality of the data recording process, and varies from project to project.

Note that PSH should be available at early phase of the project ("a priori" information) and it should contain information collected from previous systems where the existing software was used, therefore building the PSH "a posterior' should not be considered as it does not pre-qualify its level of confidence for its use in the project.

More guidelines and information about PSH can be found in [FAA-DOT-report] and [FAA-DOT-Handbook].

7.7 Development process examination

Numerous attempts have been made to equate specific objectives for which Product Service History (PSH) can be traded with. PSH does not provide any direct objective evidence of the process used in creating the software that it describes. Suppliers wishing to make use of product service history should determine a way of demonstrating compliance with the objectives of the project requirements. With this in mind, it is correct to focus specific attention on things that can be done to gain confidence in these other areas. This often involves complementing product service history with additional alternate methods. Process examination can be used to support the assessment of the reuse of existing software as an additional alternate method to complement product service history.

Process examination is the acceptance of the evidence that a recognized development process was applied to the existing software. Process examination can help to establish the validity and value of the product development records. Product information is not very useful without confidence that the processes that generated the information are comparable to commonly accepted practices for developing high critical software. Various methods have been developed to assess these processes, including ESA's method based on ISO15504:2004, defined in ECSS-Q-HB-80-02. It is understood that these processes do not meet the intent of project specific requirements, yet portions of these assessments can be used to support some of them.

If it can be demonstrated that the existing software was developed under a defined process that had an independent accreditation throughout the software life cycle, then it can be possible to show that the defined process complies with the project requirements objectives. While all of the objectives of the project requirements should be met (for the identified software criticality level), satisfactory evidence of the software development processes can allow credit to be taken for satisfying objectives achieved in the process used by the original supplier of the existing software.

The data from this process should be analyzed against the objectives of project requirements to determine which objectives were satisfied by the process.

Where existing software is incorporated into the high critical related software, the analysis of the current software development process should include analysis of the original development process and any additional activities undertaken to meet the project requirements and the criticality level of the software product where it should be integrated into.

Annex A
Content of Software Reuse File (SRF)

Content of a software reuse file

This Annex provides guidelines for the completion of ECSS-E-ST-40 Software Reuse File DRD. It is in line with clause 5 of this handbook that presents the different activities to be performed for the reuse of existing software, mentioning when the SRF is filled in. In addition to clause 5, this Annex lists the different sections of the SRF and provides guidelines about what their content could be and when they should be filled in.

<1> Introduction

The SRF should contain a description of the purpose, objective, content and the reason prompting its preparation.

<2> Applicable and reference documents

The SRF should list the applicable and reference documents to support the generation of the document.

This list will include references of all incoming inspection reports produced when acquiring the selected existing software to be reused.

<3> Terms, definitions and abbreviated terms

The SRF should include any additional terms, definition or abbreviated terms used.

<4> Presentation of the existing software intended to be reused

<4.1> Requirements definition

This section is expected to describe the requirements of the reused software, including resources and performances of the existing software in the operational environment. Description is also provided of the reused existing software operational environment. Section 5.2.2 of this handbook provides detailed guidelines for the requirements definition.

To help achieve the later acceptance and qualification activities, all requirements for the reused existing software identified in this section should be clearly and uniquely identified or numbered in order to be traceable for qualification purposes

This SRF section is expected for all software criticality categories and appropriate effort should be dedicated to completing this section properly.

It should be first provided at the SRR review based on the system-software requirements. Then it should be completed for the PDR review when both the software requirements and architecture are available.

<4.2> Existing software candidates

The SRF should provide information about each existing software candidate and its supplier(s). It consists in the collection of both: the information for the existing software candidate and existing software supplier information.

The SRF should describe the technical and management information available on the software intended for reuse.

The SSS of the project may already list the requirements on the existing software to be used by or incorporated into the system (or constituent software product) (e.g. a specific real time operating system) documented as computer resource requirements or design requirements and constraints [ECSS-E-ST-40 Annex B SSS DRD].

In addition, for each candidate, the SRF should provide (or state the absence of) the following information about both the candidate existing software and its supplier:

- software item name and main features;
- considered version and list of components;
- licensing price, agreement and conditions (e.g. installation, operation, training and usage conditions; copyright and intellectual property rights e.g. for modifications; exportability constraints; license transfer);
- industrial property and exportability constraints, if any;
- implementation language;
- development and execution environment (e.g. platform, and operating system) and the durability and validity of methods and tools used in the initial development, that are envisaged to be used again;
- applicable dispositions, conditions and price for acceptance, warranty, maintenance, installation, use and training (e.g. responsibility and commitment on duration of both the warranty and maintenance service, possibility of changes);
- available support information;
- any other (commercial) existing software necessary for software execution, if any;
- size of the software (e.g. number of source code lines, and size of the executable code).
- existing software supplier name and contact details;
- existing software supplier industrial standards (including quality)
- existing software supplier experience and references
- existing software supplier support
- existing software supplier configuration management
- existing software supplier policy for obsolescence and life-span forecast
- Product Service History: this is provided when existing. This can alternatively contain references to approved usage in projects with similar criticality category level as the existing software where it is going to be used

- Cost and conditions for supplying certification material (if needed)

This SRF section is expected for all software criticality categories and at SRR and PDR reviews. See section 5.2.2 for the activities that will produce this information.

<5> Compatibility of existing software with project requirements

<5.1> Gap analysis

The SRF should describe which part of the project requirements (RB) are intended to be implemented through software reuse.

For each software item, the SRF reports about the availability and quality status (e.g. completeness and correctness.) of the following information:

a. software requirements documentation;

b. software architectural and detailed design documentation;

c. forward and backward traceability between system requirements, software requirements, design and code;

d. unit tests documentation and coverage;

e. integration tests documentation and coverage;

f. validation documentation and coverage;

g. verification reports;

h. performance (e.g. memory occupation, CPU load);

i. operational performances;

j. residual non conformance and waivers;

k. user operational documentation (e.g. user manual);

l. code quality (adherence to coding standards, metrics).

For each of the points listed above, the SRF should document the quality level of the existing software with respect to the applicable project requirements, according to the criticality of the system function implemented. This means that for criticality category D bullets c, d, e and g are not applicable and bullet b is limited to the architectural design. See clause 5.2.3 of ECSS-Q-HB-80-01 for the activities that will produce this information.

<5.2> Derived requirements

Derived requirements about needs for the system to prevent adverse effects of any needed and unneeded functions of any of the existing software candidates should be described in this section.

The requirements about the handling of any potential adverse effect of any unneeded functions refer to the identification of potential design techniques around the reused existing software.

The requirements about the handling of any potential adverse effect of any needed functions refer to the identification of potential design techniques for the software system where the existing software will be integrated (see clause 7.3 different SW design techniques).

All these derived requirements should be clearly and uniquely identified or numbered in order to be traceable for qualification purposes. See section 5.2.4 for the activities that will produce this information.

This SRF section is expected for all software criticality categories and at SRR and PDR reviews.

<6> Software reuse analysis conclusion

The SRF should document the results of the software reuse assessment.

For each existing software candidate, the SRF should provide the following information:

- decision to reuse or not reuse, based on the information provided in previous chapters and in case of adaptation needs, a description of the adaptation need;
- estimated level of reuse;
- assumptions and methods applied when estimating the level of reuse.

In this section an evaluation table should be established summarizing the collected supplier information and showing the final position of each solution.

This section should identify potential final candidates that could possibly satisfy the requirements for the reused software identified in section <4.1> of the SRF (see section 5.2.2). A final selection would in principle be made based on the assessment evaluation results detailed in section 5.3.3.

NOTE Even if there is only one potential candidate, proper justification is needed for why the software should be reused and not developed, and should focus on demonstrating how the functionality of the existing software candidate meets those requirements identified in section <4.1> of the SRF.

The selection and the justification of the choice should be provided for all criticality categories and at the SRF file at SRR and PDR reviews.

<7> Detailed results of evaluation

The SRF should include the detailed results of the evaluation. See section 5.3.2 for the activities that will produce this information.

NOTE It is recommended to provide the detailed results of the evaluation in an appendix.

<8> Corrective actions

The SRF should document any corrective actions identified to ensure that the software intended for reuse meets the applicable project requirements and should document the detailed results of the implementation of the identified corrective actions. This information is proposed to be provided in the following two sections:

<8.1> Reuse qualification plan

Definition of the activities to be done to the reused of existing software to meet the verification requested in the current project for the SW criticality level of the software where the reused software is going to be used. See 5.5 for the activities that produce this information for the reuse of existing software and see 6.3 for the tool qualification activities.

<8.2> Reuse qualification results

Qualification plan implementation results. See sections 5.5 and 7 for the activities that will produce this information for the reuse of existing software and see section 6.3 for the tool qualification activities.

<9> Configuration status

The SRF should include the detailed configuration status of the reused software baseline. Product identification, with product configuration management information as product version and available documentation. See section 5.4.3 for the activities that produce this information.

<10> Certification aspects (when required)

A summary and a reference to the information about the existing software certification data package

NOTE The term certification is related to commercially available software products that are supported by "certification packages" to satisfy DO-178 objectives.

<11> Product Service History (when needed)

Annex B
Content of the Product Service History file

This Annex describes the content of the Product Service History file.

The PSH supplier should provide, as a minimum, the following information:

<1> General information

- Software release number
- Number of licenses installed
- Observation period
- Hardware serial number
- Operation time
- Operation environment
- Number of hours in service per mode
- Configuration file and parameters used
- Number of anomalies and alarms on the period
- Number of automatic reloads or reboots
- Number of version upgrade (plus justification of the upgrade)
- Number of “power-up” with date
- Number of “reset” with date
- Number of “shut-down” with date
- Number of return for maintenance

<2> Software analysis

The supplier should provide an exhaustive list of all reuse existing software (including their principal functionality and their dependencies), identifying:

- the ones which are reused without modification
- the ones reused with modification
- the ones which are new developed software

For the two firsts, the PSH file should be formalized

<3> Comparison with current project software standard

A comparison between the standards applied during the development of the software proposed for reuse and the standards specified in the current project for the applicable SW criticality category level.

<4> Hardware environment analysis

The supplier should analyse and explain the differences between operational hardware environment used to provide PSH data and the operational hardware environment of the current project.

NOTE The hardware environment of PSH should be the most similar possible to the target hardware environment.

<5> Operating environment analyse

Analyze the differences between operating environments (e.g. functionalities, performances, constraints) of the existing application in the PSH and the one used in the current project context (e.g. % memory usage, CPU load)

If these operating environments differ, additional verifications in the current project context should be defined in the software development plan.

<6> Length of service period

The supplier should define the length of service period covered by PSH and specify the context of number of hours in service, including factors such as operational modes. Specifically, the supplier should provide the definition of “normal operation” and “normal operation time”.

The supplier should provide the list with all the components which have never been used in normal operation and should plan additional verifications for these modules for safety purpose.

<7> Anomaly definition and anomaly rate

The supplier should define and justify the criteria to identify an anomaly (i.e.what is the definition of an anomaly).

The supplier should define the global anomaly rate threshold (the acceptable global anomaly rate), which should be closed to the associated subsystem MTBF. If subsystem MTBF is not considered, a justification of the defined threshold should be provided.

The supplier should provide an estimation of anomaly rate versus time (refer to Table B-2).

<8> PSH data feeding

The supplier should identify the periodicity used to feed the contents of the PSH file.

<9> Configuration management process

The configuration management process of reuse existing software and associated data should be described, including not only internal management anomalies and version management (e.g. in-house subsystem supplier) but also external anomalies and version management (third party supplier, for instance, in case of a new version of a COTS product).

The supplier should also provide evidences of the application of the configuration management process.

<10> PSH raw data collection

The supplier should identify the data to be provided in the PSH file and the means to collect them.

<11> Anomalies report

Three categories of anomalies should be distinguished and reported in the PSH:

- Anomalies leading to a specification/requirements evolution
- Anomalies related to the development process, design, coding and verification
- Anomalies related to hardware problems

The anomalies report for the two first categories of anomalies should be provided (hardware anomalies are not considered) including:

- Anomalies list and identification
- Origin and cause of each anomaly (and the link to the concerned components or requirements for traceability)
- Corrections of each anomaly (and the link to the concerned components for traceability), which can be either product or process correction
- The decision taken, such as to use the previous version, to perform additional test/verifications.
- List of non-regressions tests performed.

<12> Anomalies Estimation

Definition of anomalies and definition of the global acceptable anomaly rate estimation of the anomalies versus time, providing the following information:

Table B-1: Anomaly rate estimation

	During Overall validation	Expected value on a 3 months duration	Expected value on a 6 months duration
Total number of modifications			
Number of problem reports			
Major			
Minor			
Number of evolutions			
Global anomaly rate			
Anomaly rate of problem reports (Major)			
Anomaly rate of problem reports (minor)			
Anomaly rate of evolutions			
Ratio of COTS software modules impacted by modification			
Ratio of COTS line of code impacted by modifications			
Number of COTS upgrade performed			

<13> Stability and maturity of the product

The supplier should provide the history of modifications at each PSH report delivery.

Table B-2: Anomaly rate versus time

	From… to…	From…to…	From….to…
Total number of modifications			
Number of problem reports			
Major			
Minor			
Number of evolutions			
Global anomaly rate			
Anomaly rate of problem reports (Major)			
Anomaly rate of problem reports (minor)			
Anomaly rate of evolutions			
Ratio of COTS SW modules impacted by modification			
Ratio of COTS line of code impacted by modifications			
Number of COTS upgrade performed			

The supplier should compare the observed global anomaly to the threshold defined for the project as shown in Table B-2. If the global anomaly rate observed is higher than the threshold defined, then the supplier should conduct an analysis with the customer for the PSH validation.

The supplier should compare the observed anomalies rate to the estimated ones versus time. If the observed anomaly does not evolve as estimated, then the supplier should conduct an analysis with the customer for the PSH validation.

<14> Version configuration report

Configuration changes during the product service history should be identified and the effect analysed with the customer, to confirm the stability and maturity of the software. Any change should be performed according to the configuration management process. Any uncontrolled change during the product service history can invalidate the use of product service history.

Annex C
Risk management considerations

C.1 Introduction

The guidelines provided in this Annex are complementary information to facilitate the identification of the particular risks associated with reuse of existing software. Therefore, in the performance of risk management for a project including the integration of existing software, other candidate risks from those presented here and which are related to reuse of existing software should be also considered.

C.2 Risk scenarios and mitigation actions

- The risk scenarios related to reuse are almost all induced by the same basic underlying phenomena:
  - deficient control of the supplier of the existing software
  - deficient knowledge of product characteristics
  - existing software designed for a different context.

Therefore, it is particularly important the identification and analysis of risk due to the combination of commercial, legal and technical considerations.

- In addition, it should be noticed that a number of the risks classes not directly related to reuse can be exacerbated in a situation of reuse. Risk categories such as safety, cost, reliability, performance and schedule are naturally as relevant in a context of reuse as they are in any other project; however, each of these can have added weight in a situation of reuse.
- Examples of project objectives which can be placed at risk by a given reuse strategy are:
  - cost (for example if additional evaluation, design, integration or replacement activity is required)
  - schedule (as above, as well as due to uncertainty in existing software supplier's delivery or release dates):
    - Estimation of the effort to integrating the existing software. The mitigation action will depend on among other whether:
      - The historical exists and corresponds to similar level of reuse
      - The components integrated fit the project requirements
      - Availability of the component on time
      - The supplier can provide on-site support
    - Poor component procurement validation. The mitigation action can be the pre-integration for requirements assessment

- Documentation availability
- Additional integration effort due to unexpected behaviour. Mitigate action can be to analyze the level of definition of the external interfaces

— reliability of result (especially due to lack of visibility of internal structure of component):

- Reliability of the component. Mitigation action can be:
  - Perform black box testing
  - Perform reverse engineering of the module to glean information about its internal working to allow white box testing
  - Ask whether the unit test used are suitable and whether they have shown their accuracy in past projects
  - Ask for PSH justification

— usability of result (due to different assumptions of operational context):

- Documentation availability or the lack of visibility of the design and internal structure of components where source code is not available (e.g. COTS). The mitigation action can be the use of prototypes to assess the existing software. If the source code is not available, pre-integration for requirements assessment can be envisaged.
- Effective requirement coverage or components which were not designed for environments with the same requirements on safety and reliability. Mitigation actions can be the same as above.
- Lack of validation. Mitigation actions can be the same as above.

— maintainability of result (due to additional or poor change):

- Poor maintainability. If the source code is not available, actions to mitigate this risk can be:
  - Evaluate maintainability towards the ability to evolve of the system
  - Evaluate maintainability towards PSH and problem report management
- Discontinuance of maintenance and/or obsolescence of component. If the source code is not available, actions to mitigate this risk can be:
  - Evaluate maintainability towards life length planned for the system
  - Try to ensure the non-discontinuance of the maintenance
  - Envisage contractual section to let the code and documentation available in case of discontinuance of maintenance
- Product control or lack of information on the release plans of existing software suppliers. Mitigation actions can be:
  - Early implementation of mechanisms for controlling changes
  - Request PSH information
  - Take into account reliability and completeness of track record of the supplier, configuration management suppliers process

— quality, safety or security of result

- Feasibility of safety or security requirements and/or lack of visibility on product assurance process. Mitigation actions can be:
  - Get evidences in product assurance files
  - Get evidences from PSH file
  - Take into account whether the developer has quality system certification
  - Perform independent or self evaluation of process assessment
  - Perform independent or self evaluation of product quality assessment

— functional compromises

- Degree of independence from product or supplier for the functional implementation. The mitigation action can consist in:
  - Analyze if modifying existing software is the ultimate solution
  - Implement wrappers to isolate the system from certain aspects of the components behaviour
  - Increase the level of unit/integration/validation test of the reused component.
  - If the source code is not available, then analyze design documentation, if possible

— operational costs

— performance limitations

- Exist stringent response time or throughput requirements. Mitigation action can consist in performing scheduling analysis to reduce or eliminate the uncertainties

- In estimating the likelihood of each scenario, the following should be taken into account:
  — the depth and extent of evaluation performed to date on the existing software
  — the reliability/maturity of the component as evidenced by established track record (product service history – see Annex B of this handbook, maintenance records)
  — the level and quality of information available about the design and internals of the existing software
  — the organisation's level of expertise with the chosen existing software candidates
  — the lifetime of the system into which the existing software should be integrated
  — the expected amount of change in the system
  — the expected amount of change in the existing software, and the likelihood that the current version continues to be supported by the existing software supplier
  — the probability of change in the existing software having consequences on the system
  — the availability of the existing software, as well as competing solutions
  — the similarity between the current context and that for which the existing software was developed
  — technology forecasts and estimations of market volatility
  — the existing software supplier’s ability to provide support, and the cost of this support

— the degree to which the existing software supplier's actions can be influenced

— the economic soundness of the existing software supplier

— past experiences with this existing software supplier.