IEC/PAS 62814

Edition 1.0 2012-12

PUBLICLY AVAILABLE SPECIFICATION

PRE-STANDARD

Dependability of software products containing reusable components – Guidance for functionality and tests

Withdrawn

IECNORM.COM : Click to view the full PDF of IEC/PAS 62814:2012

THIS PUBLICATION IS COPYRIGHT PROTECTED
Copyright © 2012 IEC, Geneva, Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either IEC or IEC's member National Committee in the country of the requester.
If you have any questions about IEC copyright or have an enquiry about obtaining additional rights to this publication, please contact the address below or your local IEC member National Committee for further information.

IEC Central Office	Tel.: +41 22 919 02 11
3, rue de Varembé	Fax: +41 22 919 03 00
CH-1211 Geneva 20	info@iec.ch
Switzerland	www.iec.ch

About the IEC
The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications
The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

Useful links:

IEC publications search - www.iec.ch/searchpub

The advanced search enables you to find IEC publications by a variety of criteria (reference number, text, technical committee,…).
It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished

Stay up to date on all new IEC publications. Just Published details all new publications released. Available on-line and also once a month by email.

Electropedia - www.electropedia.org

The world's leading online dictionary of electronic and electrical terms containing more than 30 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary (IEV) on-line.

Customer Service Centre - webstore.iec.ch/csc

If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: csc@iec.ch.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

IEC/PAS 62814

Edition 1.0 2012-12

PUBLICLY AVAILABLE SPECIFICATION

PRE-STANDARD

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Dependability of software products containing reusable components – Guidance for functionality and tests

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE XA

ICS 03.120.01

ISBN 978-2-83220-501-3

Warning! Make sure that you obtained this publication from an authorized distributor.

® Registered trademark of the International Electrotechnical Commission

CONTENTS

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
WITHDRAWN

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

INTERNATIONAL ELECTROTECHNICAL COMMISSION

____

DEPENDABILITY OF SOFTWARE PRODUCTS CONTAINING REUSABLE COMPONENTS – GUIDANCE FOR FUNCTIONALITY AND TESTS

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.

2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.

3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.

4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.

5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.

6) All users should ensure that they have the latest edition of this publication.

7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.

8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.

9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

A PAS is a technical specification not fulfilling the requirements for a standard, but made available to the public.

IEC/PAS 62814 has been processed by IEC technical committee 56: Dependability.

The text of this PAS is based on the following document:	This PAS was approved for publication by the P-members of the committee concerned as indicated in the following document
Draft PAS	**Report on voting**
56/1479/PAS	56/1490/RVD

Following publication of this PAS, which is a pre-standard publication, the technical committee or subcommittee concerned may transform it into an International Standard.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

This PAS shall remain valid for an initial maximum period of 3 years starting from the publication date. The validity may be extended for a single period up to a maximum of 3 years, at the end of which it shall be published as another type of normative document, or shall be withdrawn.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

INTRODUCTION

Technological growth is accelerating; development cycles for products are becoming shorter and shorter. At the same time software is taking an increasingly important part in the control and functionality of products and in integrating the functions of hardware components. The disciplined development of software has been going on for more than 40 years and software is now available in many forms and formats. Apparently, the cost of software development can easily be amortized if it is embedded as often, and in as many different products, as possible. This potential benefit of software reuse should at no time be at the expense of dependability. Dependability is the ability of a system to perform as and when required to meet specific objectives under given conditions of use.

Any innovative product that has matured enough to hit the shelves needs a new and progressive approach. Dependability of the products is an attribute that is mandatory for newly developed or reused software (and the complete product into which the software is embedded) to be accepted and sold. Therefore, the dependability of software and its components should be assured in just the same way that the dependability of hardware and its components have been assured for many decades. This requires the standardization of software and software components to keep up with the ever higher levels which hardware components continue to achieve.

The dependability of a system infers that the system is trustworthy and capable of performing the desired service upon demand to satisfy user needs. Whereas a software component may be perfectly suited to one application, it may prove to cause severe faults in other applications. To allow the innovators to concentrate on their main task – to create new and better products with an extended functionality – it is fundamental to provide the certainty that reused software is dependable in its new application and does not need to be re-designed from scratch. Safety and security aspects might be combined if required. Therefore an adequate test process considering the changed purpose and the different application configuration in combination with new, reused, or further used components is needed. Altogether, testing of software products containing reused components is an important target to be reached.

An additional, important aspect to be considered is the energy efficiency and eco-friendliness of hardware products controlled by software. Reuse of a component with a bad energy consumption behaviour will multiply this bad behaviour, and thus negatively impact the entire energy consumption of the new system that is composed of such components; the same way as an undependable component impacts the dependability of the system into which it will be built. A rule of thumb is that reused software should not result in a product consuming more energy than a comparable energy-efficient product on the market.

This publicly available specification (PAS) addresses the functionality, testing and dependability of software components to be reused and products that contain software to be used in more than one application; that is, reused by the same or by another development organization, regardless of whether it belongs to the same or another legal entity than the one that has developed this software.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

DEPENDABILITY OF SOFTWARE PRODUCTS CONTAINING REUSABLE COMPONENTS – GUIDANCE FOR FUNCTIONALITY AND TESTS

1 Scope

This publicly available specification introduces the concept of assuring reused components and their usage within new products. It provides information and criteria about the tests and analysis required for products containing such reused parts. The objective is to support the engineering requirements for functionality and tests of reusable software components and composite systems containing such components in evaluating and assuring reuse dependability.

Focus is on the dependability of software reuse and, thus, this PAS complements IEC 62309 which exclusively considers hardware reuse. In addition to this previous hardware-related IEC standard, the present PAS also crosses further, appropriate software-related standards to be applied in the development and qualification of software components that are intended to be reused and products that reuse existing components. In other words, this PAS encompasses the features of software components for reuse, their integration into the receiving system, and related tests. Their performance and qualification and the qualification of the receiving system is subject to existing standards, for example ISO/IEC 25000 [01][1], IEC 61508-3 [01] and IEC 61508-4 [03]. The process framework of ISO/IEC 12207 [04] on systems and software engineering and ISO/IEC 25000 [01] on system aspects of dependability on software engineering apply to this PAS.

The purpose of this PAS is to ensure through analysis and tests that the functionality, dependability and eco-friendliness of a new product containing reused software components is comparable to a product with only new components. This would justify the manufacturer providing the next customer with a warranty for the functionality and dependability of a product with reused components. As each set of hardware/software has a unique relationship and is governed by its operational scenario, the dependability determination has to consider the underlying operational background. Dependability also influences safety. Therefore, wherever it seems necessary, safety aspects have to be considered the way IEC 60300-1 addresses safety issues.

This PAS can also be applied in producing product-specific standards by technical committees responsible for an application sector.

This PAS is not intended for certification purposes.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 60300-1, *Dependability management – Part 1: Dependability management systems*

IEC 62628, *Guidance on software aspects of dependability*

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

[1] Numbers in square brackets refer to the Bibliography.

IEC 62309, *Dependability of products containing reused parts – Requirements for functionality and tests*

3 Terms, definitions and abbreviations

For the purposes of this document, the following terms and definitions apply.

3.1 Terms related to software engineering

3.1.1
software
programs, procedures, rules, documentation, and data of an information processing system

Note 1 to entry: Software is an intellectual creation that is independent of the medium upon which it is recorded.

Note 2 to entry: Software requires hardware devices to run, to store, and transmit data.

Note 3 to entry: Documentation includes: requirements specifications, design specifications, source code listings, comments in source code, "Help" text and messages for display at the computer/human interface, installation instructions, operating instructions, user manuals, and support guides used in software maintenance.

3.1.2
embedded software
software within a system whose primary purpose is realizing an application

EXAMPLE Software used in the brake control systems of motor vehicles, or to control an x-ray system in medical health-care.

3.1.3
software unit/software module
software element in programming codes that can be separately specified, compiled, documented and tested to perform a task or activity to achieve a desired outcome of a software function

Note 1 to entry: The terms "unit" and "module" are often used interchangeably or defined to be sub-elements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized.

3.1.4
software (configuration) item
software item that has been configured and treated as a single item in the configuration management process

Note 1 to entry: A software configuration item can consist of one or more software units to perform a software function.

3.1.5
software function/(software) function block
elementary operation performed by the software module or unit as specified or defined as per stated requirements to fulfil a well-defined user or system function or a part of it

EXAMPLE Calculation of sinus of a given angle is a function block of a unit to calculate trigonometric functions; giving the address to buy a ticket is a function block of a web portal.

Note 1 to entry: Software units consist of function blocks.

Note 2 to entry: A function block contains input variables, output variables, through variables, internal variables, and an internal behaviour description of the function block.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

3.1.6
software system
defined set of software items that, when integrated, behave collectively to satisfy a requirement

EXAMPLES Application software for accounting and information management, application-oriented system software for text processing / performance analysis / programming tools, system software for linking library functions.

3.1.7
(computer) program
set of coded instructions executed to perform specified logical and mathematical operations on data

Note 1 to entry: Programming is the general activity of software development in which the programmer or computer user states a specific set of instructions that the computer has to perform.

Note 2 to entry: A program consists of a combination of coded instructions and data definitions that enable computer hardware to perform computational or control functions.

3.1.8
program code
character or bit pattern that is assigned a particular meaning to express a computer program in a programming language

Note 1 to entry: "Source codes" are coded instructions and data definitions expressed in a form suitable for input to a transducer, that is, assembler, compiler, or other translator.

Note 2 to entry: "Object code" or "binary code" or "executable code" is the bit pattern obtained from a translator and is ready to run.

Note 3 to entry: "Coding" is the process of transforming of logic and data from design specifications or descriptions into a programming language.

Note 4 to entry: A programming language is a language used to express computer programs.

3.1.9
product line
collection of systems potentially derivable from a single architecture

3.2 Terms related to software dependability

3.2.1
software dependability
ability of the software to perform as and when required when integrated in system operation

3.2.2
reuse dependability
ability of a composite system containing reusable components to perform as and when required to meet users' service needs

3.2.3
software fault
bug
error or flaw in a software item that may prevent it from performing as required

Note 1 to entry: Software faults are either specification faults, or design faults, or programming faults, or compiler-inserted faults, or faults introduced during software maintenance.

Note 2 to entry: A software fault is dormant until activated by a specific trigger and usually reverts to being dormant when the trigger is removed.

Note 3 to entry: In the context of this standard, a bug is a special case of software fault, also known as latent software fault.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

3.2.4
software failure
failure that is a manifestation of a software fault

Note 1 to entry: A single software fault will continue to manifest itself as a failure until it is removed.

3.2.5
validation
confirmation by examination and through provision of objective evidence that the requirements for a specific intended use or application have been fulfilled

Note 1 to entry: Validation answers the question whether or not the right software has been developed.

3.2.6
verification
confirmation by examination and through the provision of objective evidence that specified requirements have been fulfilled

Note 1 to entry: Verification answers the question whether or not the developed software is correct.

3.2.7
qualification
process of validation and verification (V&V), used to demonstrate that the product is capable of meeting its specification for all the required conditions and environments

3.2.8
quality target
specified level of quality as a goal; wherever possible, quantified using a software metric

EXAMPLE The overall reliability of the composite system in terms of the required MTBF, or the requirement that cyclomatic complexity of a software unit be kept below 7.

3.3 Terms related to software reuse

3.3.1
software reuse
using a software asset, i.e., software or software knowledge, in the solution of a different problem in order to construct new software [05]

Note 1 to entry: This notion covers both “heritage” and “legacy” software, and it is refined into categories: “black-box”, “white-box”, “adaptive”, “systematic”, and “accidental” reuse (see 3.3.12 to 3.3.16).

Note 2 to entry: Opposite of “software reuse” is “software one-use” that requires being developed from scratch.

EXAMPLE Some dedicated software routines, such as security codes, are not designed for reuse; they are one-use components.

3.3.2
software (reuse) asset
software configuration item that has been designed for use in multiple contexts and domains

EXAMPLE 1 Design, specification, source code, documentation, test suites, manual procedures, etc.

EXAMPLE 2 Availability of the information that a specific navigation function uses an algorithm based on Kalman filter.

Note 1 to entry: Also a software-based or software-oriented knowledge is an asset.

3.3.3
context
software environment tied to mission and software requirements

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

3.3.4
domain
problem space or application area

3.3.5
software reusability
degree to which a (reuse) asset can be used in more than one software system or in building other assets

Note 1 to entry: In a (reuse) repository software reusability represents the characteristics of an asset that make it easy to reuse vertically or horizontally.

Note 2 to entry: Usability is a measure of software unit's or system's functionality, ease of use, and efficiency.

3.3.6
(software) component
constituent of a software system with specified interfaces and explicit context and domain dependencies

Note 1 to entry: A software component can consist of one or more software units to perform a software function.

EXAMPLE 1 An individual component is a software package, a web service, or a module that encapsulates a set of related functions (or data).

3.3.7
(software) component off the shelf
COTS
commercially available components

Note 1 to entry: A COTS software can consist of one or more software units to perform a software function.

Note 2 to entry: Components in governmental use are called "government off the shelf (GOTS)".

Note 3 to entry: COTS and GOTS software usually represents components; they can be also stand-alone applications.

Note 4 to entry: COTS and GOTS typically realize reuse incorporation or integration.

Note 5 to entry: COTS and GOTS are designed to be implemented easily into existing systems without the need for customization ("glue code", "wrappers", 3.3.8, 3.3.10).

EXAMPLE 1 Microsoft Office is a stand-alone COTS application that is a packaged software solution for businesses. An operating system, a word processor, a compiler, etc. are further examples of stand-alone COTS.

EXAMPLE 2 Also libraries that need linkage to an application code, e.g., graphic engines, Windows DLLs, etc., are COTS components.

EXAMPLE 3 Software that is used to create software, but is not part of a composite software system, is not COTS software; it is a development tool. However, development environments with runtime modules are COTS (e.g., Visual Basic™, Sysbase™), or information retrieval applications (e.g., hypertext and data mining tools), or operating system utilities (e.g., for file operations and memory management).

3.3.8
glue code
software that intermediates between the reusable component and the receiving system

3.3.9
connector
interface elements of composite system to receive reusable components

3.3.10
wrappers
additional software to complete the functional and interface requirements if they are not priorly fulfilled

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

3.3.11
(reuse) repository
storage of a collection of reusable components

Note 1 to entry: In a narrower sense, “software libraries” have the same function as software repositories, e.g., building sets of reusable software units such as trigonometric functions.

3.3.12
accidental reuse
reuse without strategy, typically reusing software components not designed for reuse

Note 1 to entry: Also known as “ad hoc” or “opportunistic” reuse.

3.3.13
systematic reuse
developing software components intended for reuse and/or building new applications from those reusable components, following a formal plan of product line, also known as “planned reuse”

3.3.14
adaptive reuse
using previously developed software that is modified only for portability, e.g., a new application on a different operating system

3.3.15
black-box reuse
reuse of unmodified software components, incorporating existing software components into a new application without modification

3.3.16
white-box reuse
modifying and integrating software (function) blocks into new applications

3.3.17
vertical reuse
reuse in the same domain

3.3.18
horizontal reuse
reuse in different domains

3.3.19
internal reuse
in-house reuse
reuse of a software unit developed within the company, or government unit

3.3.20
external reuse
reuse of a software unit of another company, or government unit

Note 1 to entry: A “third party software” is usually written by another company as a legal entity. It incorporates external reuse if it will be used in a context or domain other than that for which it has been designed and developed. It can be, however, also a dedicated, one-use software component.

EXAMPLE 1 Open-source software (OSS), mostly for external reuse.

EXAMPLE 2 A service-oriented architecture (SOA) can operate on components in internal or external reuse.

3.3.21
heritage software
inherited software reused from a previous mission that has currently been in usage

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

3.3.22
legacy software
software reused from a previous mission that has currently been out of usage, or in restricted usage

3.4 General terms

3.4.1
component to be reused
software component that is intended to be reused in a different context or domain other than its original development context or domain, by any kind of software reuse

3.4.2
reusable component
software component to be reused after being qualified for reuse

3.4.3
receiving system
software system in which reusable component(s) will be integrated

3.4.4
integration of reusable components
process of installation/assembly of reusable components into the receiving system, including integration validation to ensure the proper functionality of the final system

3.4.5
composite system
final system resulting from the integration of reusable components

3.4.6
qualified composite system
composite system after qualification

3.5 Abbreviations

COTS	Commercial-off-the-shelf
GOTS	Government-off-the-shelf
FB	Function block
FTA	Fault tree analysis
IP	Internet provider
IT	Information technology
KSLOC	Kilo-(thousand) source lines of code
OSS	Open-source software
PAS	Publicly available specification
RBD	Reliability block diagram
SOA	Service-oriented architecture

4 Dependability of software reuse methodology – Reusability-driven software development

4.1 General

Software reuse has many facets; the practical and relevant kinds have been defined in 3.3. A general taxonomy of software reuse is included in Table 1, which uses the following seven aspects for a thorough, exemplary classification [06].

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

– reuse *assets* and *entities* can be product-oriented and, thus, concrete, such as components; they can also be ideal, such as concepts, ideas, algorithms, etc.;
– *domain scope* refers to application area (3.3.17 and 3.3.18);
– *development scope* refers to origin of the component (3.3.19 and 3.3.20);
– additional work required prior to reuse is referred to by *modification (*3.3.14, 3.3.15, 3.3.16*)*;
– whether and which kind of work is to be done in performing reuse is a *managerial* aspect (3.3.12, 3.3.13);
– reuse *approach* is *compositional* if existing components are reused (such as the Unix shell); *generative* reuse requires application or code generators (such as refine and meta tool);
– *direct* reuse approach requires no "glue code" that intermediates between the reusable component and the receiving system, *indirect* reuse necessitates an intermediate entity (4.4).

Table 1 – Summary of reuse classification

Reuse asset	Reuse entity	Domain scope	Development Scope	Modification	Management	Approach
Ideas, concepts	Architectures	Vertical	Internal	Adaptive	Accidental	Compositional
Artefacts, components	Requirements	Horizontal	External	Black Box	Systematic	Generative
Procedures, skills	Designs			White Box		Indirect
	Specifications					Direct
	Source code					
	Object code					
	Test cases					

Clause 4 describes which dependability methods are available, which objectives should be achieved and which preconditions are assumed.

Software reuse is the process of creating software systems from existing software rather than building software systems from scratch. The vision of software reuse is as old as software itself – it was introduced already in 1968, in the year as the term “Software Engineering” was coined during the constitutional NATO conference in Germany [07].

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

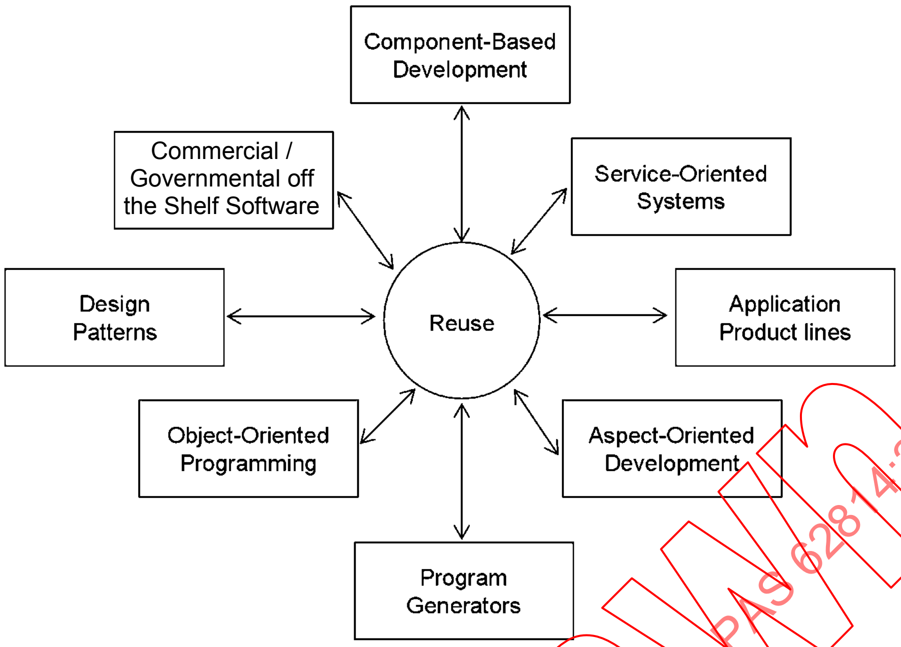

Component-Based Development
Commercial / Governmental off the Shelf Software
Service-Oriented Systems
Design Patterns
Reuse
Application Product lines
Object-Oriented Programming
Aspect-Oriented Development
Program Generators

Figure 1 – Approaches to software reuse and its elements

Software reuse is not limited to the source or object code; it has, moreover, to consider all of the information that is related to the product generating processes, including also requirements, analysis, design, documents, and test cases apart from the code (Figure 1). Examples of well-known, widely accepted practices of software reuse are [08]:

– Component-based development (CBD): Building systems by integrating components that conform to system's specification.
– COTS integration: CBD using commercial components.
– Service-oriented systems: Building systems by linking shared services.
– Program generators: Embedding knowledge of a particular type of application to produce component(s) in that domain.
– Application product lines: Generalization of an application around a common architecture so that it can be used to produce different applications in different domains for different customers.
– Object-oriented programming: Implementing applications using "objects" that consist of data structures, methods (algorithms) and their interactions and computer programs.
– Aspect-oriented software development: Weaving shared components into an application at different places when the program is compiled, if separation of concerns is feasible.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

4.2 Dependability methods for reuse

As hardware reuse (IEC 62309), also software reuse, whether involving home-grown or COTS components, certainly promises lower cost, better dependability, thus providing a decrease in risk, increase of productivity, and, consequently, considerable potential for a less stressful development process.

During the last decades much research progress has been achieved to technically master software reuse by industrial best practices. However, software reuse has proved to be complex and steadily evolving with the progress of software engineering so that it needs appropriate methods and techniques for dependability assessment and assurance.

The applicability of the conventional dependability techniques to the analysis and evaluation of software reuse is limited. A common recommendation for software reuse is to constraint the reusable software components to perform only one function completely. This restriction is supposed to ease the implementation, deployment, and maintenance of reusable components and composite systems that contain such components. Furthermore, deviation from such restriction could have adverse effect on dependability due to the possibility of errors introduced into the software during implementation or maintenance. This requirement "one component – one function" is, however, a severe constraint that limits the scope of software reuse, and is thus difficult to be accepted by the industrial and commercial software development and marketing, where rather universally deployable components are more attractive.

During the last years, specific methods provide effective and broadly well-understood and, thus, accepted solutions for dependability assurance, especially concerning functionality and testing of software. A selective amount of them are appropriate for effective reuse dependability analysis and evaluation, applicable to both reusable components and composite systems. Indeed, many factors influence the dependability performance in the life cycle, including the early stages and implementation and integration phases.

Figure 1 includes constructive methods and approaches that help avoid faults while producing reusable software, which is the best way of fault handling. Nevertheless, also analytic methods are necessary to detect and eliminate faults that could not be avoided. For testing and test case generation, these analytical approaches use either source code (if available) leading to white-box testing, or software specification and user profile, leading to black-box testing. Grey-box testing combines both approaches. Measures of the effectiveness of a test set in fault revealing, coverage-oriented adequacy criteria use the ratio of the portion of the specification or code that is covered by the given test set while testing to the uncovered portion. Examples are: C0 test, C1 test, dd-test, du-test (white-box testing), or cause-effect analysis and operational profile analysis (black-box testing). Belief is, the higher the degree of test coverage, the lower is the risk of having critical software artefacts that have not been sifted through.

To avoid singularity of test case-oriented testing, formal (e.g. sound mathematical methods) are recommended, such as model checking, model-based testing, and formal proofs of programs. Finally, reliability growth models statistically evaluate software based on test data recorded during testing, more precisely the number and time intervals of failures triggered by test cases. Thus, with some effort, it is also possible to determine the reliability of software reuse. Annex B, Annex C, and Annex D explain and exemplify these methods.

When considering the dependability of reuse, it is essential to include the operation and maintenance stages in the analysis of composite systems, such as legacy or heritage software. Dependability performance and dependability of service of reuse should be continuously monitored, analysed and evaluated.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

4.3 Dependability-related objectives of software reuse

Reuse dependability is the ability of a composite system containing reusable components to perform as and when required to meet users' service needs. Thus, for fulfilling the requirements of assessment of reuse dependability, the identification and deployment of the relevant methods and techniques of reuse dependability are indispensable.

Several activities influence the dependability of reuse, including the following ones:

– development, operation, and maintenance of reusable components as reuse assets and composite systems with reuse assets;

– management of practice and assets of reuse.

The dependability of the composite systems with reuse assets is influenced by following factors:

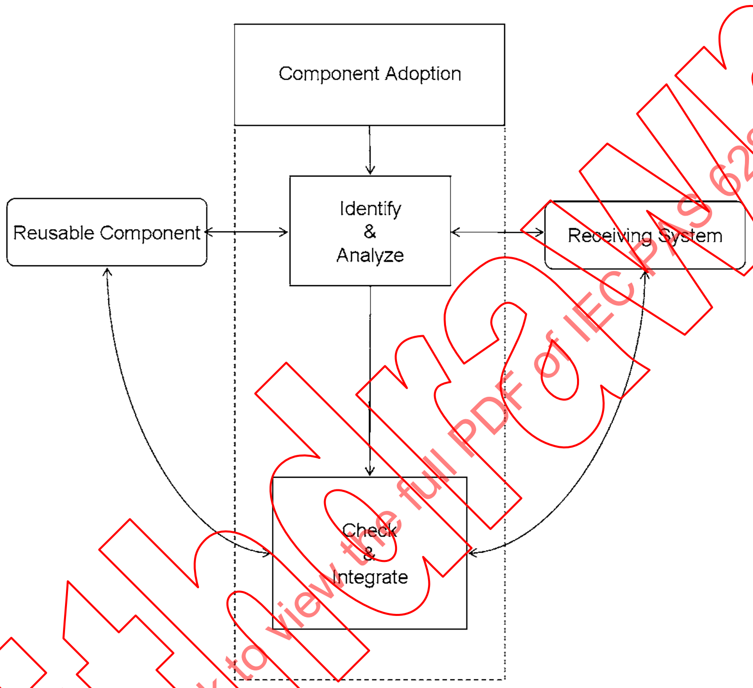
– software functions to serve and satisfy user needs;

– dependability of these services.

The present document focuses on requirements on functionality and tests of software products containing reusable components.

4.4 Ingredients of software reuse and hypotheses for reuse dependability

Software reuse, that is, the use of existing software components or knowledge to build a new software system, is supposed to realize benefits such as improved productivity, but also not negatively impact the dependability. Figure 2 summarizes the reuse process that consists of identifying an appropriate reusable component, its analysis, and integration into the receiving system.



withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Figure 2 – Elements of the reuse process

Figure 3 depicts the integration of a reusable component into a receiving system to produce a composite system. This process requires that both parts be “prepared,” e.g. by connectors and glue code (3.3.9, 3.3.10), which means additional work and, thus, additional dependability risks.

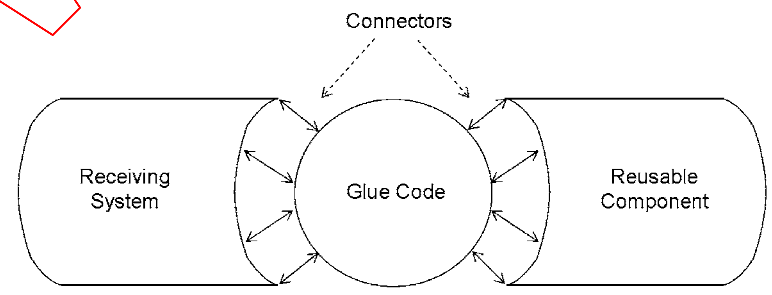


Figure 3 – Integration of the reusable component

Approaches to measure the software reliability are available. The relationship between hardware reuse and dependability has been pursued, for example in IEC 62309. The desirable, however, naïve demand about reuse can be stated as follows [06]:

“Increased software reuse can significantly improve the dependability of a software system.”

This desire necessitates fulfilling following preconditions:

– D1: Reusable components have to be more dependable than their one-use equivalent.
– D2: Composite systems built by reusable components have to be more dependable than its equivalent built by one-use components.
– D3: Generating a system from a high-level, user-oriented, behaviour-based specification realized by reusable components has to be more dependable than one built by hand.

To make these requirements operable, D1 and D2 can be studied by means of reliability measures that can be determined using software reliability models that are available for industrial practice (AIAA R-013-1992 [10], IEEE 1633-2008 [08]). It is evident that, from the view point of reliability determination, the glue software, and its relation with connectors of the receiving system, are to be considered as a part of the reuse process. Thus, D2 is influenced by the reliability of the glue software to an extent that is comparable to the reliability of the reusable component.

D3 concerns the domain which can be judged best by the end user of the system, considering also the domain of the composite system. See Annex D for practical examples.

5 Software reuse dependability methodology applications

5.1 Application aspects and organization of dependable software reuse

5.1.1 General

Dependability methodologies include application aspects and the organization of the reuse. Pre-store and pre-use characteristics should be met and the cases build-for reuse or build-by-reuse should be distinguished.

Another point covers validation and reliability aspects of the software. Also the assumptions and rules to improve software dependability are described and the hardware/software interaction is taken into account.

Architecture is the key to software reuse. The architecture of a system commits its structure to combine the elements it is comprised of and their features, and relations among those elements. Typical structures are hierarchical, centralized (star-form), or decentralized (network-form); relations are defined as consists-of or neighboured. Architectural elements can be event, state, or service-oriented. It is important for reuse that the software architecture should allow a precise design and specification of interfaces and their dependability-critical features so that it enables evaluation, selection, acquisition, and integration of reusable components into the receiving system.

While planning substantial reuse of their software components, software engineers are often overly optimistic concerning how much reusable functionality can be achieved. Reuse is not an ultimate saver of costs, schedule, or dependability. Even COTS deployment often satisfies only less than 40 % of the functionality of an industrial application.

Also important is the addressing of the critical non-functional requirements, that is, dependability and quality, which certainly result in schedule and cost impacts, and, caused by poor dependability and reliability, maybe invoke severe safety and security risks. Note that if the functional and interface requirements are not fulfilled, glue code (3.3.8) and wrappers (3.3.10) are to be planned, specified, designed, implemented, and carefully tested.

"Software-by-reuse" is the use of existing applications or their components to build new applications. It is widely accepted and convenient to consider software reusability from the following viewpoints.

– Build-***for***-reuse enables planned production of reusable components.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

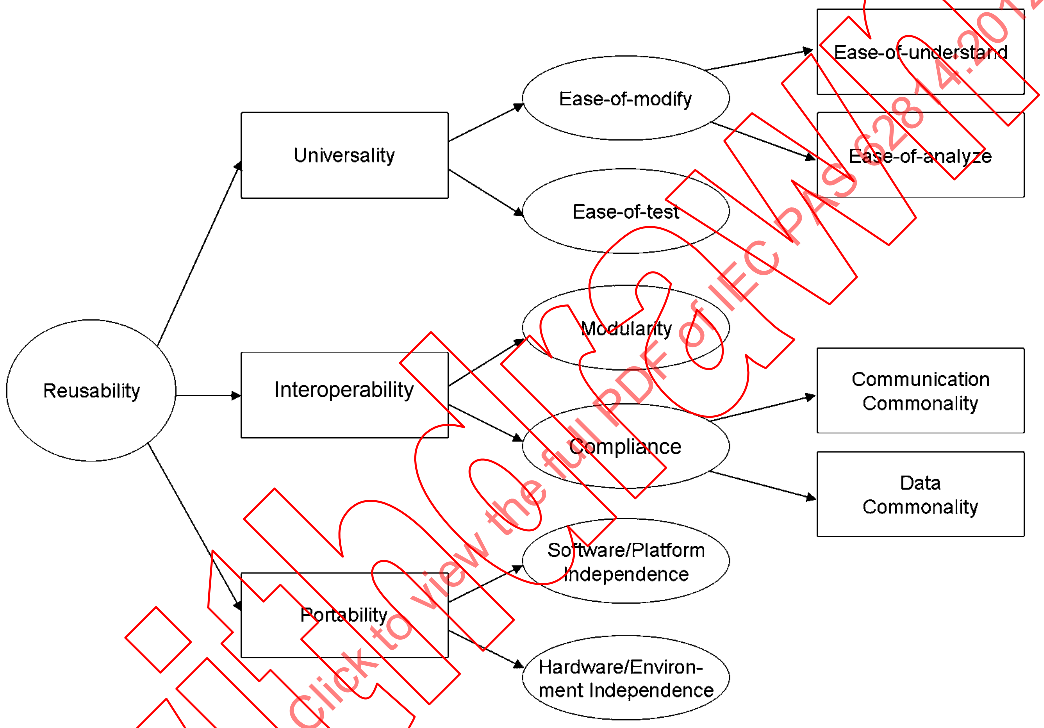
– Build-***by***-reuse attempts planned production of systems using reusable components.

Both of these viewpoints focus on characteristics of reusability that are to be checked before storing the component and before reusing it in a new product.

Following recommendations do not address only internal reuse; they can easily be adopted also for external reuse.

5.1.2 Pre-*store* characteristics of reusability

Before storing a component for reusability, following characteristics are to be considered as to whether and how to use it in other systems [12] (Figure 4).



IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Figure 4 – Characteristics of reusability

- *Universality* is defined over the range of the functionality and thus enables reusability in a large class of domains and contexts. Remember that the universality feature is likely to be in conflict with dependability (4.2). Universality requires the following sub-characteristics:
  - ease-of-modify requires the availability of the source code and an appropriate documentation of the component. The most important factors are ease-of-understand and ease-of-analyse;
  - ease-of-test requires availability of appropriate test criteria to generate test cases, and test oracles to justify the decision whether or not the modification achieves its goals, required by both reusable components and new system's specifications (B.1).
- *Interoperability* is defined over the ability to communicate with other systems to adapt and communicate, and requires following features.
  - *Modularity* is an architectural property of software being composed of units that are functionally independent from each other in the sense that a change in one component has minimal impact on other components.
  - *Compliance* requires the component follow existing standards and de-facto standards, i.e., state-of-the-art rules methodology or best practice techniques. It concerns primarily interfaces, protocols, bandwidth, and data structures used, among other things.

- *Portability* is defined by the ability of software to operate on different platforms and requires the independence from software and hardware resources, e.g., from programming languages, operating system characteristics, etc., and operating hardware and application periphery, e.g., processor's word length and speed, environmental influences (e.g., electro-magnetic emissions).

5.1.3 Pre-*use* characteristics of reusability

Prior to reusing a stored component, it will be retrieved from a library or repository, and, likely, modified to satisfy the receiving system's requirements. Accordingly, functionality characteristic is the major pre-use factor to be checked. In addition to the pre-store characteristics (Figure 4), pre-use process has to check the following sub-characteristics to validate functionality.

– *Suitability* is the ability of the component to achieve the receiving system's requirements and expectedly produce the results (outputs and behaviour) of the new system.

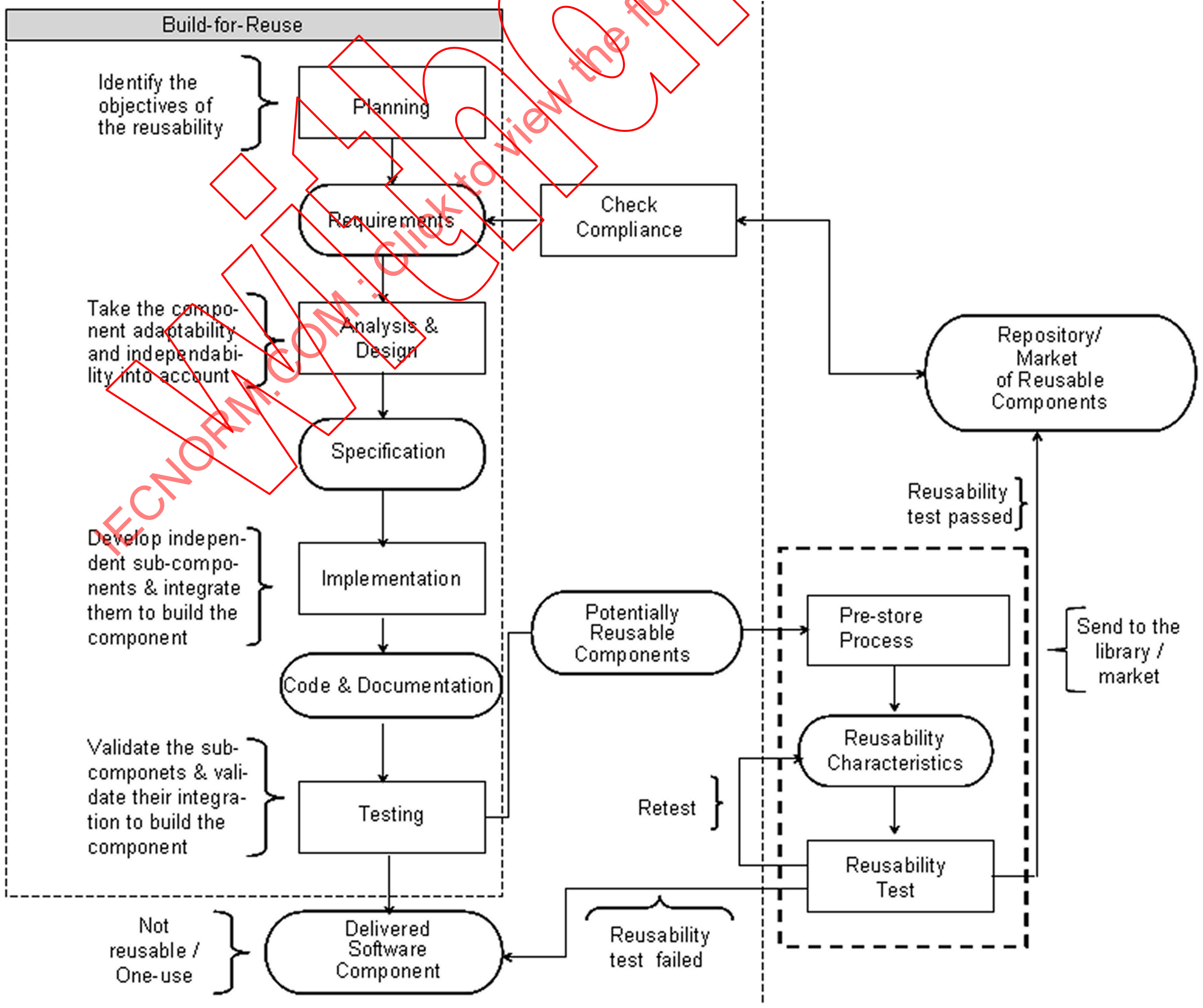
– *Accuracy* is the precision of the results expected from the reused component.

– *Compliance* is the feature of the reused component to comply with certain standards the composite system has to follow, perhaps in addition to the standards followed by pre-store process.

5.1.4 Build-*for*-reuse

To avoid dependability and productivity risks, reusable components have to be of high quality. Poor quality, and the risks associated, would be reproduced each time a low-quality component is reused. To avoid those risks, the development of reusable components requires some extra effort.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn


Build-for-Reuse
Identify the objectives of the reusability
Planning
Requirements
Check Compliance
Take the component adaptability and independability into account
Analysis & Design
Repository/ Market of Reusable Components
Specification
Reusability test passed
Develop independent sub-components & integrate them to build the component
Implementation
Potentially Reusable Components
Pre-store Process
Send to the library / market
Code & Documentation
Reusability Characteristics
Validate the sub-componets & validate their integration to build the component
Testing
Retest
Reusability Test
Not reusable / One-use
Delivered Software Component
Reusability test failed

Legend ▭ Process steps, activities ⬭ Results / triggers, information

Figure 5 – Build-for-reuse framework

Figure 5 summarizes the stages of the development of reusable components having some differences from the common software development [12].

– In the *planning phase,* the objectives of the reusability to enable the component extraction are defined. The component to be reused will be identified. Domain analysis provides precise explanation when/where/how it will be reused, and, if possible, the domains where it should not be reused will be listed.

– In the requirement phase, decision will be made whether to build the component for reuse or for only one-use. The reuse library and the market are checked as to whether such a component already exists.

– During analysis, design, implementation, and test phases, the pre-use characteristics are considered and assured (Figure 4, 5.1.2).

– *In the pre-store* process, the characteristics (Figure 4) of a reusable system are extracted to assure its qualification for use in different system(s).

– *In reusability test,* pre-store characteristics of the reusable system are validated before it is to be transferred to a library or a repository for storing, enabling consideration in a market.

As a result of the build-for-reuse process, two software products are delivered at the point in time “Delivered Software Product” (Figure 5) instead of one – even though those two are literally identical. The first one is the required system for the market (the component in the centre of Figure 5). The second one is the reusable component deployed in the specific composite system.

5.1.5 Build-*by*-reuse

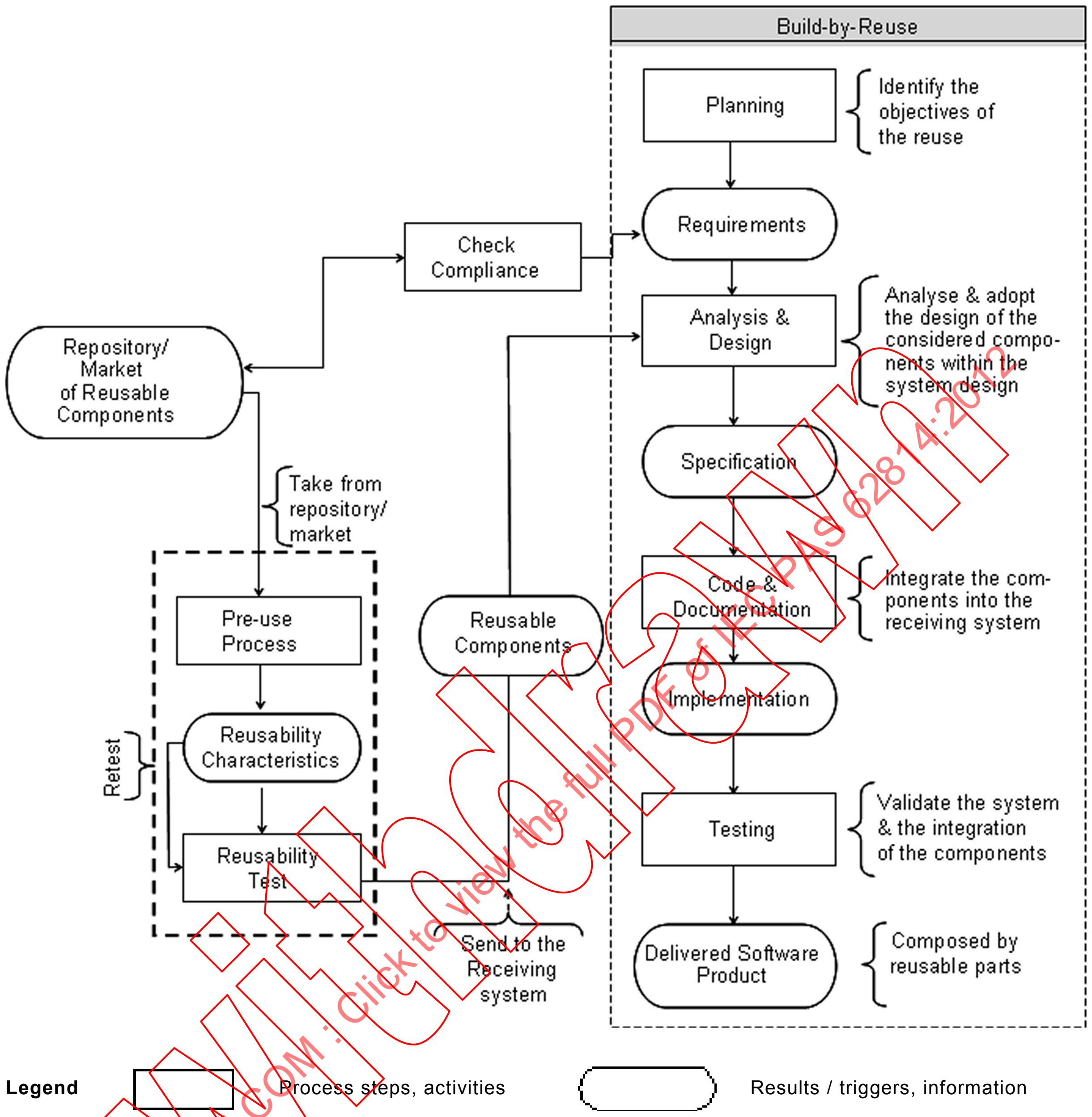
Reusability needs a proper management concept to select, modify (as little as possible), re-test, and deploy a reusable component. Figure 6 summarizes the stages of the development of reusable components that has the following differences from the common software development [12].

– In the requirement phase, the availability of reusable components should be checked and, if available, evaluated.

– During the system analysis, design and implementation phases, modifications should be considered and carried-out; in addition, likely side effects should be checked.

– Whereas the requirements defined by the library should be validated by test in build-for-reuse, or the requirements of the new, composite system should be validated on the market by the test in build-by-reuse (5.1.2 and 5.1.3).

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012


Build-by-Reuse
Planning
Identify the objectives of the reuse
Requirements
Check Compliance
Analysis & Design
Analyse & adopt the design of the considered components within the system design
Repository/ Market of Reusable Components
Specification
Take from repository/ market
Code & Documentation
Integrate the components into the receiving system
Pre-use Process
Reusable Components
Implementation
Reusability Characteristics
Retest
Testing
Validate the system & the integration of the components
Reusability Test
Send to the Receiving system
Delivered Software Product
Composed by reusable parts
Legend
Process steps, activities
Results / triggers, information

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

Figure 6 – Build-by-reuse framework

5.1.6 Coupling "build-for-reuse" and "build-by-reuse"

The transfer from "for-reuse" to "by-reuse" realizes the link between the development "for-use" and "by-use" and is of great importance for the success of reuse. This link couples Figure 5 and Figure 6 producing Figure 7, which summarizes the entire reuse process. The coupling link is located in the centre of Figure 7 and consists of three elements:

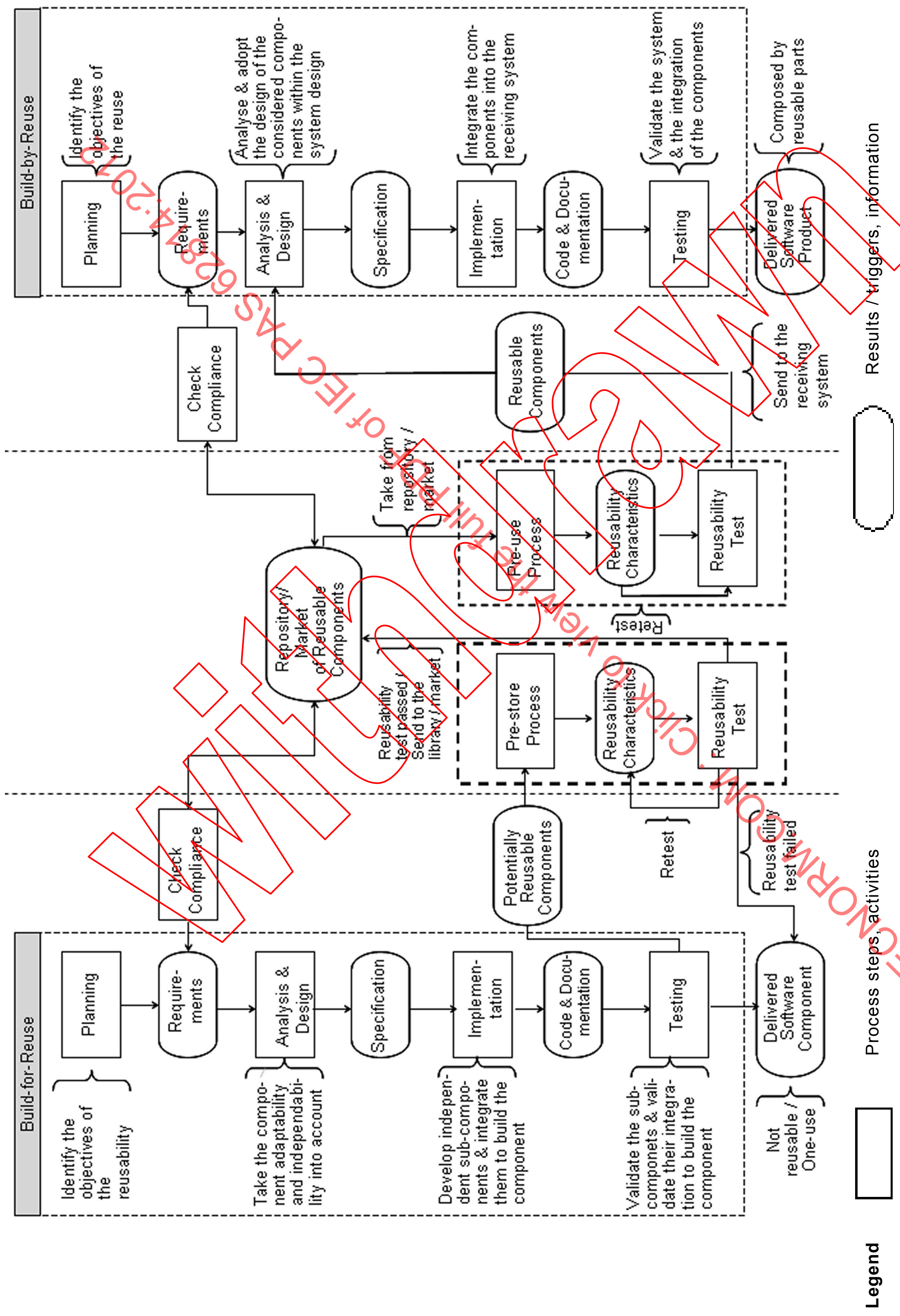
– Pre-store process,
– pre-use process, and
– storage of reusable components.

Figure 7 reveals also the main steps of the reuse process and also its requirements:

understand – analyse – modify – test

As explained in 5.1.2, ease-of-understanding, ease-of-analysis, ease-of-modification, and ease-of-testing are the key factors to *ease*-of-reuse of a component. They enable the selection and analysis of components to be reused in a new system (B.1).

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012



Build-for-Reuse
Identify the objectives of the reusability
Planning
Require-ments
Take the component adaptability and independability into account
Analysis & Design
Specification
Develop independent sub-components & integrate them to build the component
Implemen-tation
Code & Docu-mentation
Validate the sub-componets & validate their integration to build the component
Testing
Not reusable / One-use
Delivered Software Component
Check Compliance
Potentially Reusable Components
Retest
Reusability test failed
Pre-store Process
Reusability Characteristics
Reusability Test
Reusability test passed / Send to the library / market
Repository/ Market of Reusable Components
Retest
Take from repository / market
Pre-use Process
Reusability Characteristics
Reusability Test
Check Compliance
Reusable Components
Send to the receiving system
Build-by-Reuse
Identify the objectives of the reuse
Planning
Require-ments
Analyse & adopt the design of the considered components within the system design
Analysis & Design
Specification
Integrate the components into the receiving system
Implemen-tation
Code & Docu-mentation
Validate the system & the integration of the components
Testing
Composed by reusable parts
Delivered Software Product
Legend
Process steps, activities
Results / triggers, information

Figure 7 – Combining “build-for-reuse” and “build-by-reuse”

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Note that during its life cycle a reusable component encounters two types of modifications and thus two types of tests. The first to satisfy and assure the characteristics required by the library and/or the market and a second type of modification and test are then necessary prior to its deployment after it has been removed from the storage. Note also that the first modification and test aims at generalization of the component, whereas the second one aims at specialization of this component to satisfy the requirements of a new system in realizing another application in another domain.

5.2 Validation, re-validation and reliability of software reuse

The most common form of reuse is using software developed for one-use in a new application, which is, accidental reuse. One of the major objectives of the present PAS is to warn the managers that this kind of unplanned reuse can be a potential minefield because it can cause the inheritance of all the problems of the pre-existing software in the reaping of only a few of its benefits. Many managers, while planning for software reuse, forget that both the reused component and the composite system are to be tested in the new domain. Experience reports say that reusable software can cost 60 % more than one-use software, whereby a good portion of additional costs goes to testing.

Software reuse involves redesign, reimplementation, and re-testing. Redesign arises if the existing functionality does not fulfil the requirements of the new task because it requires reworking to realize the new function, and, prior to this, necessitates reverse engineering to understand its current functionality. The design change leads to reimplementation. Exhaustive re-testing (as a kind of regression testing) is necessary to validate the functionality of the reused software in the new domain to determine whether or not redesign and reimplementation are needed.

Following undesirable events/situations, mostly caused by managerial misjudgement, negatively influence the dependability of software reuse:

– failing to select the right component, or to favour the wrong selection criteria;
– failing to justify and adjust the need for and/or extent of the modification of the selected component to fulfil operational or application requirements;
– failing to justify and adjust the need for and/or extent of the maintenance of the selected component during operational stage.

To avoid such undesirability, redesign, re-implementation, and re-testing activities can be clustered in following groups:

- Redesign
  – architectural design modification: detection of architectural design part(s) to be modified, realization of the modification, re-validation of the entire architectural design;
  – detailed redesign: detection of design part(s) to be modified, realization of the modification, re-validation of the entire design;
  – reverse engineering: detection of the part(s) to be modified, which are not familiar to developers; understanding, modification, re-validation of the entire component;
  – re-documentation: detection of the part(s) to be modified, modification, re-validation of the entire document;
- Re-implementation requires re-coding, code review, and unit testing (IEC 62628).
- Re-testing activities can be clustered in following groups (B.1):
  – test re-planning
  – test procedures to be altered:
  – re-integration testing
  – re-release and re-acceptance testing
  – test drivers/simulators to be altered:

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

– test reports to be rewritten.

Fundamental facts influence dependability, especially reliability when using commercially available components, e.g., COTS components for software development (B.2, Annex D).

– Very often no source code is available, thus there is no way to correct a detected fault. This is a great restriction that prohibits application of the most widely used reliability models (“reliability growth models” (AIAA R-013-1992 [10], IEEE 1633-2008 [08]) that require perfect correction of detected faults.
– If source code is available: Note that COTS software is no longer COTS after its source code is modified to correct a fault detected because the COTS supplier no longer maintains the documentation and source code (just as electronics equipment warranties are no longer valid after a seal is broken). Furthermore, the modifications can violate the original software design. From then on, modified COTS software is to be handled as an accidental reuse.

5.3 Naïve assumptions and rules for improving software reuse dependability

Commercially available software components for reuse, e.g., COTS software, address common needs, and the arguments for them often induce to following assumptions that should be, however, seriously questioned [13][14].

– “COTS software contains fewer faults than the one-use ones.” Reality is that also this kind of software is made by regular developers and, thus, is also likely subject to having bugs.
– “System integrators know exactly the functionality and interfaces of COTS.” Reality is also here that we have ordinary software and software documentation, and, thus, this kind of software also needs a learning curve.
– “Glue code” and “wrappers” are easy to write.” Reality is that they interface other people’s software, and thus to get them properly function can be a very tedious and costly process.
– “Composite system will meet user requirement”. Reality is that almost always additional effort is needed.

The following rules help to improve the dependability of reuse.

- Minimize the use of reusable components which
  – use “combis,” that is, combine and perform many non-trivial functions (all of which need to be learned, trained, and maintained);
  – do not have clearly-defined interfaces.
- Maximize the use of reusable components which
  – perform clearly-defined functions;
  – have clearly-defined interfaces, all with easy-to-understand, predictable inputs and outputs;
  – have a visible architecture that can be identified and easily understood;
  – have been on the market for some time and equivalent alternates are available from competitive sources (then look for likely industrial standards).

5.4 Dependability and reuse aspects of software/hardware interaction

5.4.1 General

It is described which limitations for reuse could occur with upgraded software and remanufactured hardware. There might be limitations due to incompatibilities between hardware and software, dependability or the ecological properties like energy consumption.

5.4.2 Reuse of software with an upgrade / remanufactured hardware

Updated, refurbished, or remanufactured hardware means in most cases the reuse of software that might not be compatible with the new hardware. An upgrade of the software will

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

be required due to changes and new features or to avoid other side effects, for example different word lengths of the present and the new processor, or electro-magnetic compatibility, or too high energy consumption. Software could also be combined with the reuse of some refurbished hardware components, which might not be compatible with a new product.

5.4.3 Limitations of hardware

In those cases where the hardware of a product or a system remains almost the same and the system is only checked for quality purposes, two documents are important: The device master record (DMR) and the device history record (DHR). DMR describes the total documentation of the manufacturer, including the required updates; the DHR describes which updates have already been made. After comparison of the requirements and the real state, the necessary installations and updates are managed to guarantee the state of the art of the system to be resold. The state to be achieved has to be the same as if the product were being put on the market the first time.

5.4.4 Limitations due to incompatibilities

Also incompatibility of the software to networks or external systems might cause limitations. These cases can violate dependability requirements, or it might not be economical to enforce compatibility.

5.4.5 Dependability, energy consumption and ecology

Embedded software, e.g., wireless sensor networks, in which most applications do not have the sensors plugged in, sensors get power from the batteries they carry. To keep the network alive as long as possible, it is very important to conserve energy while the network is functioning. For this purpose, energy-efficient algorithms are available. The situation is similar for software embedded in equipment like mobile phones. Another example is software that controls technical processes invoked by green-house gases emissions and waste and pollutants production. Tradeoff analysis may be necessary between energy efficiency, sustainability, and dependability requirements, e.g., concerning workload balancing and lifecycle extension. It is also then very important for the dependability of the entire system to have the software that conserves energy to exploit all possible power-savings features of the hardware and controlled devices and processes.

There are many other ways to reach a better energy consumption goal; for example, by sparingly using battery charging commands, avoiding excessive transport of large amounts of data, or banning obsolete software routines that do not do this. All these operations, executed by reusable software, can cause more energy consumption than necessary. Therefore, software reuse should be critically checked and made lean wherever possible. Knowledge of the operational conditions of the hardware components is necessary.

6 Software reuse assurance

6.1 General

In Clause 6 the validation and the qualification of components to be reused for build-for-reuse and build-by-reuse are explained.

Software components that are to be reused should be qualified and the qualification process should be documented in accordance with ISO/IEC 12207 [04]. The context and domain of reuse and operational and/or embedding hardware are to be identified.

Dependability and safety requirements and potential exceptions should be specified; likely conflicts between those requirements are to be identified and solved.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

6.2 Build for reuse – Validation and qualification of components to be reused

6.2.1 General

The manufacturer of reusable components and, wherever possible, the reusing party should together specify the functional properties of these components and should identify the quality targets in accordance with ISO/IEC 12207 [04]. Design, redesign, test, and re-test issues are to be defined (4.2 and 5.2).

Any modification and additional software, including wrapper and glue code necessitated by reuse, might influence dependability, safety, and/or energy efficiency. They are to be taken into account and checked. Conformance to and compatibility with hardware and overall system requirements are to be validated.

6.2.2 Validation and qualification

The manufacturer of reusable components and, wherever possible, the reusing party should together specify and perform the validation and qualification process, and should document the test and re-test results in accordance with ISO/IEC 12207 [04]. These documents should be included in the validation and qualification documentation.

For external reuse, e.g., COTS software, where the reusing party is not always known, the validation and qualification process should be carried out by the manufacturer of the reusable component. Qualification of any other kind of reuse should be carried out by the reusing party.

Recommendations concerning organization, characteristics of reusability and validation process, test and re-test criteria, as explained in Section 4 and Section 5, should be taken into account.

6.2.3 Assessment of quantifiable quality targets

The manufacturer of reusable components or the reusing party should ensure that quantifiable quality targets are met, e.g., reliability measures, complexity metrics, test coverage measures (Annex B).

6.3 Build by reuse – Validation and qualification of the receiving system

6.3.1 General

Before integration of the reusable component, the receiving system is supposed to have been qualified and the qualification process is documented in accordance with ISO/IEC 12207 [04], IEC 61508-3 and IEC 61508-4 [01][03].

The manufacturer of the receiving system and, wherever possible, the reusable component should together specify the functional properties of these components and should identify the quality targets in accordance with ISO/IEC 12207 [04]. Design issues should be defined. Architectural issues, e.g., redundancy for realizing fault tolerance, to increase the availability, and/or other aspects to meet the quality targets, should be considered during design and implementation.

6.3.2 Validation and qualification

The manufacturer of the receiving system should specify and perform the validation process and should document the validation results in accordance with ISO/IEC 12207 [04] IEC 61508-3 and IEC 61508-4 [01][03]. Qualification documents of reused components should be included in the qualification documentation of the composite system. Quality targets should be validated.

The composite system should be qualified by the reusing party considering the documentation of the components manufacturer in the light of the context and domain foreseen for the reuse.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Recommendations concerning organization, characteristics of reusability and validation process, test and re-test criteria, as explained in Clause 4 and Clause 5, should be taken into account.

6.3.3 Assessment of quantifiable quality targets

The manufacturer of the composite system should ensure that quantifiable quality targets are met, e.g., reliability measures, complexity metrics, and test coverage measures.

7 Warranty and documentation

7.1 General

For warranty life cycle, contextual criticality and the warranty period are important. The product documentation is a basis for these aspects. Product safety and control have to be met. Legal aspects cover the contract and product liability.

7.2 Life cycle, contextual criticality, warranty period

The life cycle period expected by the relevant market participants and end users at the time of release into the market of the product should be defined. The warranty period and the warranty should be granted at least as a non-strict liability for new products in accordance with the jurisdiction applicable to the product and its release into the market.

7.3 Product documentation

The composite system, its purpose and functionality, its components and their interaction should be documented. The documentation includes also requirements pertaining to the operation description, trade dress, and marketing of the product; also the content of the product manual should be documented.

7.4 Product safety and control

Safeguarding basic safety requirements to protect human health and life and valuable goods, relative to the risks involved and the degree of safety expected from the product in question, should be considered. Control should be executed by after sales control of the product's behaviour in the field.

7.5 Legal aspects

7.5.1 General

The reuse of software affects also legal issues regarding the deployment and marketing of the composite systems. Some basic principles are outlined in the following.

7.5.2 Contractual issues

A composite system containing a reusable component should comply with the required standard of quality.

Reused components may be subject to license restrictions. Even slight modifications of the software component that will be reused in a different context may infringe third party rights to this component.

NOTE The context of reuse is problematic in case of open source software that is not necessarily free for any third party use, especially in commercial applications.

The reuse of a software component differs from a simple transfer of COTS software that has been purchased and will be forwarded to a third party. Modifications needed to adjust the

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

software component for reuse for the deployment in a different context may infringe the rights granted.

Any reuse of software in a different context may also result in a loss of warranty rights or guarantees granted by the original manufacturer of the software component to be reused.

7.5.3 Product liability

Any software component, which is not producing data for interpretation of a person, but is generating direct physical effects, may be subject to product liability rules.As a result also the safety and integrity of persons and property within reach of the composite system should be observed.

The composite system shall meet the required safety of the new context in which the reused software component is deployed. Software components which have been tested and qualified for a context in which a lower level of safety was required than the level required for the new context, should be re-qualified for the requirements of the new context.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Annex A
(informative)

General remarks on software reuse

Many efforts to reuse software have succeeded; there is an increasingly overwhelming number of success stories available in literature. Almost all major companies and institutions that deal with information & communication technology practice software reuse and report about their success, e.g., Nippon Electronic Company, GTE Corporation, Raytheon, DEC, HP, NASA, and many more [15], [16] and [17].

Nevertheless, the promises of decreased cost and increased dependability, and thus decreased risks, are not always realized. The frightening news about recent disasters definitely caused by careless software reuse are still being warningly associated with and attributed to all software reuse. The failure of Therac-25 system, in which a software component was carried over from a previous version of an X-ray system, caused the machine to malfunction, resulting in the loss of several lives in a terrible way; patients were actually burned [18].

In the Ariane project, failure of a reused software component caused the loss of a rocket costing around half a billion dollars [19].

These recent disasters as a consequence of bad reuse on the one side and success stories as a consequence of good reuse on the other side are the key factors in deciding whether or not to enhance and sustain continued provision of reuse from a lucrative business perspective.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

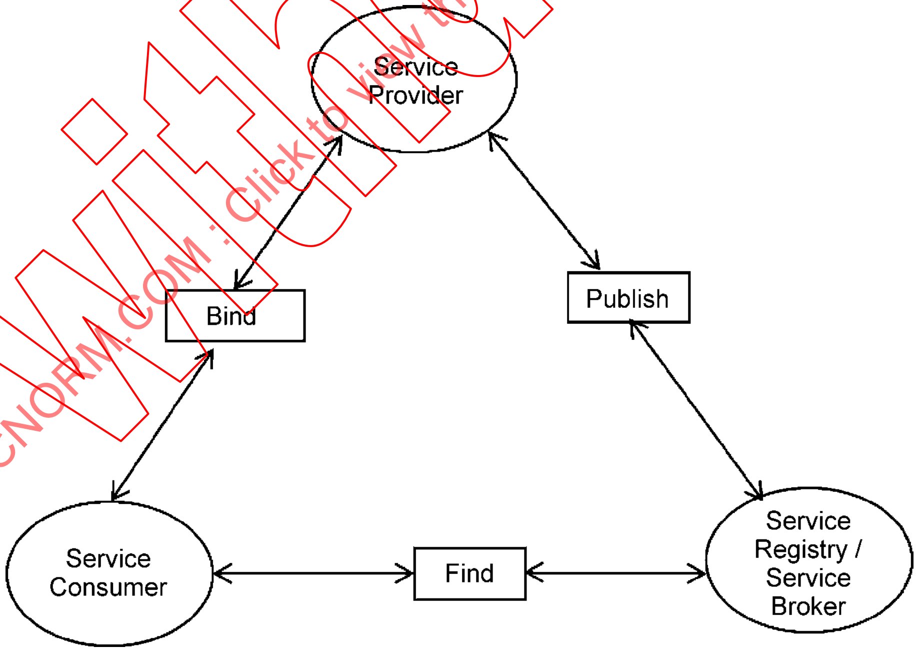


Figure A.1 – Service-oriented architecture

Not each “copy and paste” action, which programmers do daily when they construct their programs, forms a software reuse this PAS has in mind. Also calling an internal or external function and even a remote-procedure call is not necessarily a reuse this PAS would regulate. All these examples suggest that the context and domain of the called software does not change. Therefore, there is no need for them to perform pre-store and pre-use activities described in Clause 5.

Using a service in a service-oriented (SO) landscape or in “Common Object Request Broker Architecture (CORBA)” is of more interest to this PAS because the context and domain of the software that delivers a service might change. Indeed, Figure A.1 and Figure 7 both have similar constellations concerning constructing, offering, selecting, and validating services. A service has to be registered and “published” before it will be offered. Infrastructural services are offered to realize a broker, etc. [20].

The safeguarding of basic dependability requirements to protect human health and life and valuable goods, relative to the risks involved and the degree of safety expected from the product in question, should be carefully taken into account when considering reuse of existing software. Control should be executed by monitoring the product’s behaviour in the field after sales.

To sum up, before reusing a software component, the context and domain it was built for should be carefully compared with the context and domain it is intended to be built in, including the hardware and physical and organizational aspects [21].

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Annex B
(informative)

Qualification and integration of reusable software components

B.1 Testing issues

B.1.1 General

Systematic approaches to supporting compositionality during integration testing enable to establish reproducible connections between test cases for component testing and integration testing. In particular, adequate coverage criteria defining objective integration testing stopping rules are required.

A number of approaches suggest alternative possibilities of performing integration testing. It is widely recommended to proceed incrementally, that is, to integrate software units step-by-step. The reasons for avoiding the so-called “Big-bang” (non-incremental integration testing) concern economic aspects as well as effectiveness. In fact, integrating all software units in one single step assumes each of them having already been extensively tested on its own, thus requiring the cost-intensive provision of a high amount of non-reusable stubs and drivers, to be discharged later on without further benefit. In addition, non-incremental testing is likely to lead to a globally incorrect behaviour (the already mentioned “Big Bang”), extremely difficult to be diagnosed in terms of localizing its cause(s), i.e., the fault(s) to be removed.

Alternative to the “Big Bang”, incremental techniques for integration of components exist, which may be organized

– in a **top-down** fashion, i.e., integrating new modules only after all modules invoking them were already integrated and tested, or

– in a **bottom-up** fashion, i.e., integrating new modules only after all modules they invoke were already integrated and tested.

Both options complement each other with respect to their pros and cons. Consider that the hierarchical invocation tree usually consists of

– **logical modules** at the higher hierarchical levels, followed by

– **operational modules** (typically library packages) at the lower hierarchical levels.

With this in mind, it is easy to observe that

– integration testing approaches based on **top-down increments** will tend to test very thoroughly the upper logical modules, at the cost of neglecting the lower operational modules;

– the opposite is true for integration testing strategies based on **bottom-up increments**, which will tend to favour a more thorough verification of operational modules when compared with the logical ones.

Both types of modules, however, may be significantly relevant for different reasons:

– by encoding the control logic of the design**, logical modules** are likely to contain the most critical and complex faults and to require for such reasons to be thoroughly tested;

– on the other hand, even when conceived for re-use, **operational modules** may have been experienced in the past only under very particular robustness measures or circumstances preventing them from failing; identical defensive design conditions may not apply, however, to future applications.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

In order to support extensive testing of both logical and operational modules, a compromise is offered by the so-called "sandwich integration", which combines **top-down** incremental testing for logical modules with **bottom-up** incremental testing for operational modules before joining and testing both "**sandwich parts**".

B.1.2 Criteria for integration testing based on coverage measures

To measure the effectiveness of a test suite in revealing faults, mostly a coverage-oriented adequacy criterion is used; this criterion uses the ratio of the portion of the specification or code that is covered by the given test suite to the uncovered portion: the higher the degree of test coverage the lower is the risk of having critical software artefacts that have not been sifted through.

Concerning the definition of suitable coverage measures for integration testing, several approaches were proposed in the past. They mainly differ in the granularity of the underlying interaction concept. Assuming

– **components** to denote units of composition with contractually specified interfaces and

– **interfaces** to denote access points of components, through which clients can request services, then

– **integration testing** denotes the amount of testing activities concerned with exposing "defects in the interfaces and in the interactions between integrated components" [22].

The above mentioned definition of interface allows for a simple coverage concept based on the criterion of exercising at least once all access points to a component (also called *operation coverage*). Such a simplistic view of integration, however, neglects a major and frequent source of failures, caused by sporadically inappropriate interactions of correctly implemented components [23]. For this reason, following coverage concepts address the multiplicity of potential interaction behaviours, distinguishing among the following levels [24]:

– **interface coverage**, requiring to execute each interface at least once;

– **event coverage**, requiring to test each interface against all its possible invocations; in other words, event coverage does not only require the invocation of each interface once (i.e. interface coverage), but also demands "that each interface be invoked against all possible events of invocations in the application environment";

– **context-dependent coverage**, requiring to test at least once all possible operational sequences;

– **content-dependent coverage**, requiring to cover all pairs of interfaces, where one interface modifies the value of a variable used in the other, thus extending classical data flow testing concepts to interfaces (s. also [25]).

On the whole, a suitable level of integration testing granularity should lie between the two extremes of

– **black-box integration testing**, limited at the coverage of interfaces without taking into account the component-specific behaviour triggering the access to such interfaces. As commented above, this procedure is likely to result in a rather cursory test coverage;

– **white-box integration testing**, taking into account the complete control and data flow within each component. This procedure, on the other hand, is likely to result in an extremely laborious and inefficient phase, involving a considerable overlap with unit testing and requiring source code information which in case of re-usable off-the-shelf components may not be available.

For these reasons, the optimal level for the integrated system under test lies usually between both extremes just illustrated and is carried out by

– **grey-box integration testing** based on an abstract representation of component behaviour, typically a UML diagram, providing relevant information on components interaction(s) without requiring full implementation details.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

Depending on the underlying abstraction selected, different model-based integration testing strategies may be applied, like

– **dynamic model-based integration testing**, aimed at covering dynamic UML models like sequence diagrams or collaboration diagrams (e.g. by requiring the coverage of operational call sequences).
– **static model-based integration testing**, aimed at covering static UML models like state diagrams or collaboration diagrams (i.e. by requiring the coverage of state-based information).

The main limitation of dynamic model-based integration testing strategies is likely to lie in the depth of operational call sequences possibly preventing from an accurate data flow analysis. The latter is rather captured by state-based models reflecting to some abstract degree the internal behaviour of each component via state transitions. Different approaches were developed by elaborating on this concept:

– **including state-based information into dynamic models**, allowing to visualize state-dependent behavioural variants by means of graphical paths (e.g., so-called “state collaboration test models).
– **encoding component interactions** by matching state transitions (so-called “transition mappings” . Potential interactions between components (represented by state machines) are captured by pairs of state transitions, where the traversal of one transition (in the invoking component) has the effect of triggering the traversal of the other transition (in the invoked component).

Finally, a few words are devoted to the cost aspects of the testing techniques surveyed above. Except when explicitly required by licensing regulators white-box testing has been and still is usually avoided by industrial developers in view of the effort once required. This effort concerns the identification of suitable test data achieving the target coverage as well as the verification of the behaviour observable upon executing them. While this justification may have applied in the past, it is meanwhile obsolete. Test case generation techniques and tools based on evolutionary strategies are felt to have achieved the degree of maturity required for being applied to an industrial context. While they are not (and will never be) able to guarantee complete (i.e. 100 %) coverage (being based on random algorithms), they can nonetheless provide valuable support to any systematic testing phase (including integration testing) by generating a number as low as possible of test cases (i.e., test sequences with corresponding test data) capable of achieving coverage measures as high as possible.

B.2 Reliability issues

Intensive research effort was devoted in the past to the compositionality of reliability estimation in component-based systems, resulting in a number of different approaches summarized in the following.

Classical reliability approaches addressing component-based systems in general and without focusing on specific software engineering aspects, are often based on a Markov model of the underlying component-based structure. An analysis of the sensitivity of system reliability to changes in specific component reliabilities was carried out in [26].

Littlewood [27] considers the particular case of software systems and distinguishes between inter-modular failure rates and intra-modular failure rates. It is questionable whether the memory-less property (on which Markov transitions rely) always applies to systems. Depending on the granularity of components, this property may be violated.

Successive research work considered also the issue of reusability of components as well as of testing with respect to operating experience gained with them. [28] compares testing and operational profiles observed in the past with usage profiles expected in future and suggests evaluating the maximal relative deviation between past and future frequency of occurrence

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

per demand. This measure allows deriving a conservative estimation of the overall system reliability.

Saglietti [29] considers a similar problem in the light of individual software component reliability estimations derived by analysing correct component-specific operating experience by means of statistical sampling theory. This theory allows for an evaluation of testing with respect to operational experience of reusable components in case the runs observed fulfil the following conditions:

– the selection of a run should not influence the selection of further runs;
– the execution of a run should not influence the outcome of further runs;
– runs should be selected in accordance with their expected frequency of occurrence during operation;
– no failure (or at worst very few failures) is (are) observed during testing.

The approach taken combines the component-specific reliability estimations into conservative system reliability estimation. Further research was carried out to sharpen this estimation by providing reliability lower bounds at predefined confidence levels.

It has to be remarked that these approaches refer to a long-term observation of correct and representative executions such as is only possible after extensive component and integration testing phases (B.1); the latter support interaction fault detection by focusing on the validation of the underlying component-based architecture.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Annex C
(informative)

Testing and integration of reusable software components – Issues for industrial best practice

C.1 Basic concepts about software component testing

C.1.1 General

Today component engineering is gaining substantial interest in the software engineering community. In the real world, many software systems are developed based on reusable components. This annex reviews and identifies the testing and integration issues and challenges of component-based reusable systems. It also discusses testing processes and maturity model for controlling the quality of reusable components. Finally, it shares some insights on the needs of test standardization for reusable components.

As more commercial and open-source third-party components are available in the market, more software companies begin to build software systems using the component-based software engineering approach. As the advances of the software component technology continue, people begin to realize that the quality of component-based software products depends on the quality of software components and the effectiveness of software testing processes [30]. While they applied the conventional testing methods to deal with reusable components and their integration, they have encountered some new issues and opening challenges in testing and integration of reusable components [31], [30], [32]. Today, as more software systems are developed based on reusable components, engineers and managers are looking for the answers to the problems and challenges.

Clause C.2 reviews first the existing challenges and solutions in component testing and component-based software are reviewed. Then, the opening challenges in validating reusable components are discussed. Clause C.3 discusses testing issues for software components and reusable components before Clause 0 introduces component testing maturity levels in quality control of reusable components. Clause C.5 discusses the problems and solutions in integration of reusable components in component-based software.

C.1.2 Types of component testing

In traditional testing concepts, software component testing refers to testing activities that uncover software faults and validate and confirm the quality of a software component at the unit level. For checking the functions, structures and behaviours based on the given specifications in a given operational environment, white-box testing and/or black-box tests are performed to detect structure-related program faults, and b) specification-based faults [30].

In component-based software engineering, components should be validated from two different perspectives: a) vendor-oriented testing, and b) user-oriented testing. These two types of component testing have different focuses, tasks, and objectives.

- ***Vendor-oriented component testing***, which occurs as one step of a component development process. It refers to a component test process and testing activities performed by a component vendor to validate a software component based on its specifications. In vendor-oriented component testing, the primary purpose is to answer the following questions for component developers:
  - Are we building a right component with high quality?
  - Are we building a component based on the specified standards and component model?

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

- ***User-oriented component testing***, which occurs as a part of a component based software development process for a specific application project. It refers to the component validation processes and testing activities in a specific context to make sure all involved software components deliver the specified functions, interfaces, and performance. Moreover, component reuse is validated in the given context and operational environment. User-oriented component testing is performed to find the following answers for component users:
  – Are we selecting and deploying a reusable component that is right for a system?
  – Are we reusing a component correctly in a system?
  – Are we adapting or updating a component correctly for a project?

C.2 Validation processes for reusable software components

C.2.1 Types of reusable components

Today, many software systems are made based on four types of reusable components. They are: a) third-party commercial components from other vendors, or from open source, b) reusable components from other projects, c) altered reusable components from previous releases, and d) a set of newly constructed components. Therefore, the quality of software products depends on the quality of these components, and their integration as well as the effectiveness of involved validation processes and quality standards. This section covers the component testing processes for different types of components.

C.2.2 Vendor-oriented component testing

In vendor-oriented component testing, test engineers of a vendor implement a component test process based on well-defined component test models, methods, strategies and criteria to validate the developed software components. The vendor-oriented component testing has three major objectives:

– uncover as many component faults as possible;
– validate component interfaces, functions, behaviours and performance based on component specifications;
– check component reuse, packaging and deployment in the specified platforms and operation environments.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

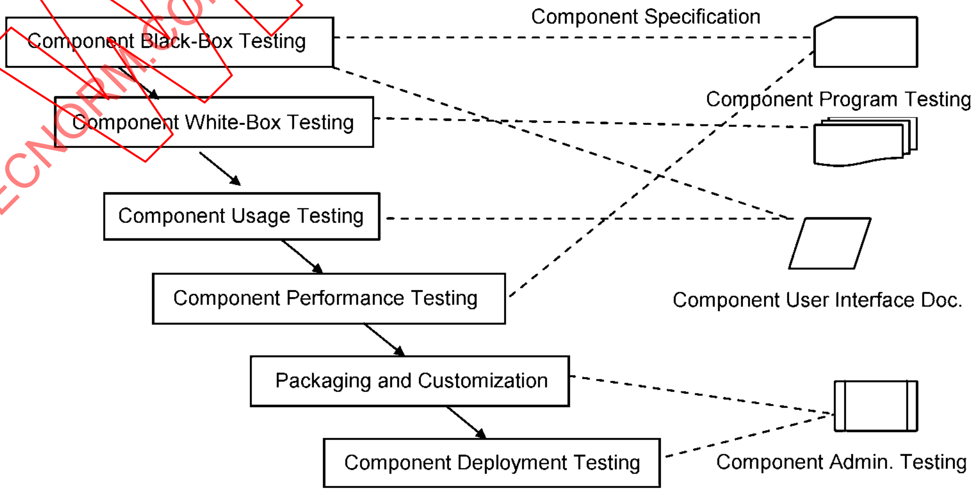

Component Black-Box Testing
Component White-Box Testing
Component Usage Testing
Component Performance Testing
Packaging and Customization
Component Deployment Testing
Component Specification
Component Program Testing
Component User Interface Doc.
Component Admin. Testing

Figure C.1 – A test process in vendor-oriented component testing [30]

As shown in Figure C.1, a component test process for a vendor consists of the following six steps:

– **Step 1: Component black-box testing –** Component developers and test engineers exercise black-box tests to check incorrect and incomplete component functions and behaviours based on the component specifications. Traditional black-box test methods can be used here.

– **Step 2: Component white-box testing –** In this step white-box tests are performed to uncover the internal faults in program logic, structure, data objects and structure. Existing white-box test methods can be used here [31], [30].

– **Step 3: Component usage testing –** Test engineers exercise various component usage patterns through contract-based component interfaces to confirm its correct functions and behaviours. Traditional usage test methods can be used here.

– **Step 4: Component performance testing –** Test engineers and quality assurance people validate and evaluate the component performance based on non-functional requirements. Traditional performance testing methods can be used here.

– **Step 5: Packaging and customization –** This step is only useful for components that provide built-in customization features and packaging facility. Its testing focus is component built-in customization features and packaging functions.

– **Step 6: Component deployment testing –** As the last step of a component test process, it validates the component deployment mechanism to make sure it is correctly designed and implemented according to a given component model.

C.2.3 User-oriented component testing and its process

User-oriented component testing checks component reuse and refers to a component validation process and its testing activities that confirm the quality of the involved software components for a specific system. Engineers involving user-oriented component testing are application component engineers, test engineers, and quality assurance groups. They perform component testing to achieve the following objectives:

– validate the functions and performance of a reusable component to make sure that they meet the specified requirements for a project and system;

– confirm the proper usage and deployment of a reusable component in a specific platform and operation environment;

– check the quality of customized components developed using reused components;

– test the quality of new components created for a specific project.

Testing reused components (such as COTS components) in a new context and domain is necessary and critical to a component user even though component vendors have already tested them in other reuse contexts. As already mentioned, the Ariane 5 disaster has showed us that using a component in a new reuse context without validating may cause serious consequences and failures.

In user-oriented component testing engineers should devote their testing efforts to the validation of component reuse by answering the following questions:

– Is a reused component packaged and deployed properly in a targeted operational environment?

– Does a reused component provide the specified user interfaces accessible to a user?

– Does a reused component provide the correct functional features, proper behaviours, and acceptable performance when it is reused in a new context and environment?

Validating newly developed components is similar to vendor-based component testing. The component test process described before can be used here. However, validating reused and updated components has different focuses and limitations. For example, a component user usually has no access to the source code and artefacts of a completely reused component from a third party. The user has to validate the reused components using black box testing without access to the personnel and expertise used to create it. This occurs as a part of a component evaluation process at the earlier phase of a component-based software development process.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

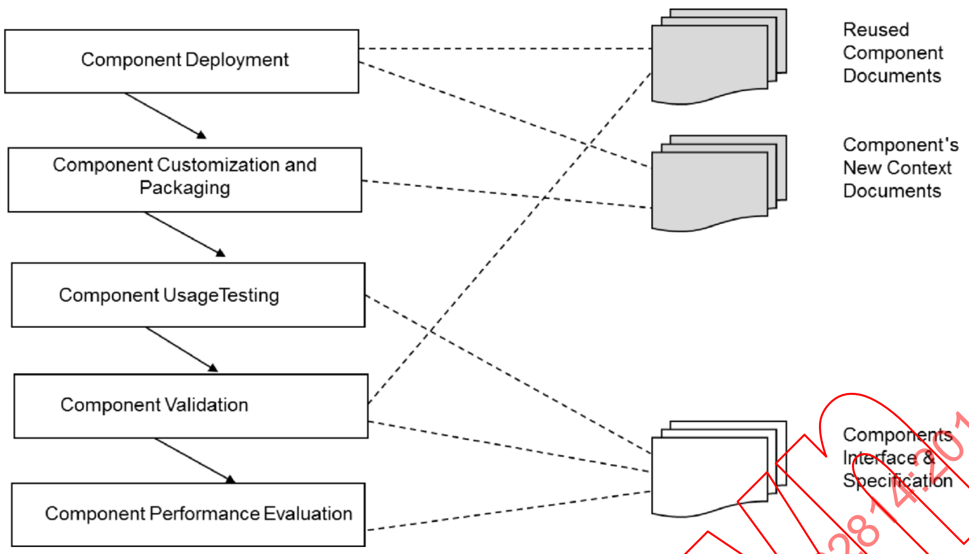

Component Deployment
Component Customization and Packaging
Component UsageTesting
Component Validation
Component Performance Evaluation
Reused Component Documents
Component's New Context Documents
Components Interface & Specification

Figure C.2 – A validation process for completely reused components [30]

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Figure C.2 shows a validation process for completely reused components from a third party. The process includes the following five steps:

– **Step 1: Component deployment** – In this step a component is validated to see if it can be successfully deployed in its new reuse context and operational environment for a project. It focuses on the built-in component deployment mechanism and supporting facility.

– **Step 2: Component customization and packaging** – In this step it is checked whether a component can be successfully packaged and properly customized using its built-in packaging and customization features in its new context environment. The major focus is on the built-in component packaging and customization mechanism, and supporting facility.

– **Step 3: Component usage testing** – In this step, a component user designs test cases to exercise different usage patterns of a component using user interfaces. Its primary goal is to cover the important component usage patterns in their new context and environment. Two typical examples are checking frequent function invocation sequences and trying typical usage patterns on data parameters in component interfaces.

– **Step 4: Component validation** – In this step component black-box tests are performed to validate component functions and behaviours in the new reuse context and environment. Various existing black-box testing methods can be used here.

– **Step 5: Component performance evaluation** – In this step the component performance is validated and measured in a new context and operation environment to make sure that it satisfies non-function requirements. The typical validation focuses are component operation (or function) speed, reliability, availability, load boundary, resource usage, and throughput.

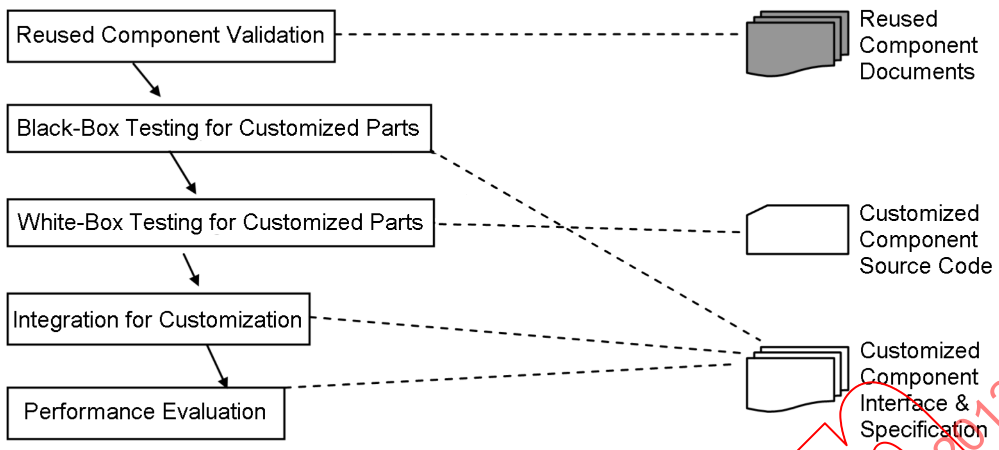

Reused Component Validation
Black-Box Testing for Customized Parts
White-Box Testing for Customized Parts
Integration for Customization
Performance Evaluation
Reused Component Documents
Customized Component Source Code
Customized Component Interface & Specification

Figure C.3 – A Validation process for adapted and customized components

Validating adapted and updated components is another major task for component users. These components are known as customized components. They are developed based on reusable components by customizing and altering them. Since they contain reused components and newly added parts for a specific project, their validation process differs from the reused components. Figure C.3 shows a validation process for customized components.

– **Step 1: Reused component validation –** In this step, component users validate completely reused components, such as COTS components, based on its previous validation process.

– **Step 2: Black-box testing for customized/updated parts –** In this step, black-box tests are exercised to check those customized parts. The major objective here is to uncover the function and behaviour faults of the newly altered parts based on the given specifications.

– **Step 3: White-box testing for customized/updated parts –** In this step, white-box tests are performed based on the given source code to uncover program logic and structure errors in the customized parts, such as added functions, adapters, and tailored parts.

– **Step 4: Integration for customization –** Reused components are integrated with customized parts (such as an adapter, new function feature) to form a specified customized component in a new reuse context and environment.

– **Step 5: Performance evaluation –** Concerning component performance, component users have to evaluate component performance and its non-function requirements for a customized component in a new context and environment.

C.3 Testing issues for software components and reusable components

Since 1990, there are a number of testing issues addressed in various technical publications [30], [32]. These issues are identified and summarized below.

– **Adequate component test criteria**

Adequate testing of modern software components is different than adequate testing of traditional modules because of their unique properties in reusability, interoperability, composition, packaging, and deployment. These new component properties extend the semantics and scope of adequate testing. For example, testing a traditional module always concerns one specific usage context and operation environment. However, a reusable software component should have diverse usage contexts, and it may support more than one operating environment. This suggests that a vendor should try to test it for diverse reuse contexts, and validate it under all specified operating environments to achieve adequate testing. On the other hand, component users also have the difficulty to understand and define an adequate test model and criteria for reusable components in a reuse context. In the past years, some published papers have proposed different test coverage criteria to address the needs in testing software components, such as API-based test execution sequences [33] and API-based test pattern coverage [34].

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
WITHDRAWN

– **Testability of reusable components**

Testability of software components usually includes a) observability, and b) controllability [24]. Controllability of a program (or component) is an important property which indicates how to control a program (or component) on its inputs, operations, behaviours and outputs. Observability of a program (or component) is another critical property which indicates how easy to observe a program in terms of operational behaviours and outputs relating to inputs. In [30], three other factors are added to observability and controllability: Understandability, traceability, and test support.

Based on the feedback from engineers, it is not easy to test third-party software components. From a customer's point of view, the testability of current software components depends on the following features:

- external tracking mechanisms and tracking interfaces in software components for a client to monitor or observe external behaviours;
- built-in controllable interfaces in software components to support the execution of component tests and check the test results;
- built-in tests and standard test interfaces to support component test automation at the unit level.

Although most in-house components contain some built-in tracking code for fault trace and exception reporting, there is not yet any consistent tracking mechanism, trace format, and tracking interface for all components. In recent years, many technical papers discuss this issue by providing different solutions to increase component testability. Systematic methods and standards help create testable components and improve their testability.

– **Component testing processes and certification criteria**

Today, many software workshops have established in-house quality control processes for software products. During the course of paradigm shift from traditional software construction to component-based software construction, they frequently run into a question about the difference between a component quality control process and a software program quality control process. They are not sure whether or not the existing quality control process and standards can be applied onto software components. Due the lack of standard component quality control processes and standards, they frequently end up in an ad-hoc component testing process without well-defined quality certification standards. As discussed in [35], [36], well-defined component testing and certification criteria and standards are needed in component testing for third-party component certification.

– **Component test drivers and stubs**

In the component engineering paradigm, software components are reusable parts for component-based software products. To support component testing, engineers have to construct test drivers and test stubs. In the past, engineers used to construct product-specific test drivers and stubs (or simulators) for a specific project based on given requirements. This approach becomes very ineffective and costly for component-based software projects because of the evolution of reusable components and their diverse functional customizations. For in-house reusable components, engineers usually use an ad hoc approach to develop simple test drivers and/or test stubs for unit tests. However, they are not easy to be reused and updated, and integrated to support component integration and system tests due to the following factors:

- they are developed using ad-hoc methods, technologies, and computer platforms;
- they are project-specific or product-specific;
- they have no well-defined standard interfaces between components and test suits, and components with a test bed.

The existing software testing methods and tools can help testers generate black-box and white-box tests for components in a systematic manner [31], taking the issues listed in Table C.1 into account.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012 Withdrawn

Table C.1 – Testing issues of reusable software components

Component test adequacy	What are adequate test criteria for reusable components in a reuse context? What are adequate test criteria for customizable components?
Component testability	What is the testability of software components? How to construct testable components? How to measure the testability of software components?
Component test suite	How to construct well-formatted test suites for software components? How to manage and maintain test suites for software components?
Component test platform, test stubs & drivers	How to construct component drivers and stubs in a systematic approach? How to maintain and manage test drivers and test stubs in a cost-effective way?
Certification and quality control	What is a well-defined certification standard for software components? How does a quality control process of software components differ from a regular quality control process for a software product?
Component test automation	How to achieve test automation for reusable software components? How to generate a reusable component test platform for software components?

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
WITHDRAWN

C.4 Testing maturity for reusable software components

Building cost-effective products needs a productive product line. As component engineering gains a wide acceptance in today's software industry, many software companies have begun to set up component-based software product lines. Delivering high quality component-based software needs also a well-defined component testing process. To understand the status of a component test process, it is important for a manager to have a tool to measure the effectiveness of the process. A maturity model is a tool that is useful to measure the maturity level of a process in an organization. In this section, we define a maturity model to help managers measure the status of a component test process. It focuses on component test standards, test criteria, management procedures, measurement activities. Figure C.4 shows the five process maturity levels which are defined as follows.

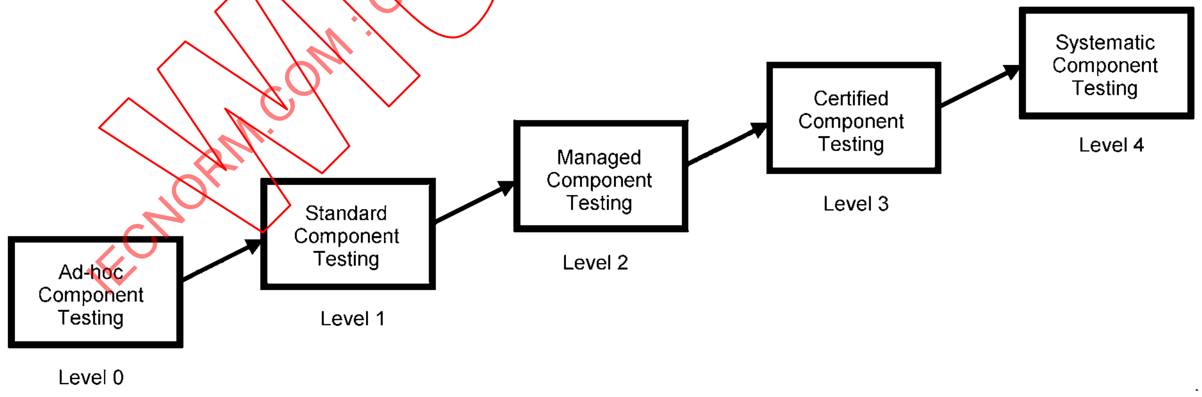

Ad-hoc Component Testing
Level 0
Standard Component Testing
Level 1
Managed Component Testing
Level 2
Certified Component Testing
Level 3
Systematic Component Testing
Level 4

Figure C.4 – Maturity levels for a component testing process

– **Level 0: Ad-hoc component testing**

A component testing process is considered as an ad hoc process if it has the following characteristics: a) ad hoc test information formats, including test cases, test procedures, test data and scripts, b) ad hoc test generation and test criteria for both white-box and black-box tests, c) ad hoc quality assurance standards and quality control systems, d) ad hoc construction of component testing environment, such as test drivers and stubs, and e) inconsistent requirements and mechanisms on tracking component behaviours. An ad hoc component test process is inefficient and costly due its poor reusability of test information and test drivers/stubs. Besides, it is very hard for managers to control and manage it. The most critical issue of the process is the quality of delivered components. Without well-defined test criteria, managers cannot control the quality of tested components.

– **Level 1: Standard component testing**

  A component testing process is characterized as a standard process if managerial operations and engineering activities are performed based on a set of test information standards, pre-defined management procedures, well-defined criteria, and well-designed mechanisms. The standards define requirements and formats on test plan, test case and data, trace messages, test reports and problem reports. Management procedures refer to a component quality control process, a component testing workflow, a configuration management procedure, and a problem tracking workflow. Test criteria include white-box and black-box test criteria, acceptance test criteria, and quality control criteria. The defined mechanisms support component tracking, problem tracking, and configuration management.

– **Level 2: Managed component testing**

  A component testing process is categorized as a managed process if it collects the detailed measures of the process and component quality, including the measures of component test cost, component test metrics, component quality metrics, and process measurements.

– **Level 3: Certified component testing**

  A component testing process is considered as a certified process if it has defined and implemented a certification standard and procedure for software components. The certification standard includes certification procedure, test plan, test tools, test platform and environment, criteria, test metrics, and test report. The test plan includes certification tests, which focus on testing of user accessible component features, installation, and customization or configuration functions. Moreover, this process has a well-defined certification procedure and workflow. A designated engineer or group, known as a component certifier, implements this procedure based on the given standard. After completing the certificate tests for a component, the certifier will issue a certificate for a component product according to the test report.

– **Level 4: Systematic component testing**

  A component testing process is characterized as a systematic process if it has defined and implemented systematic methods and mechanisms to automate this process. To achieve this goal, engineers need well-defined systematic methods to support their operations and activities. The essential systematic methods are classified in four areas: a) test suite design and construction, b) component design for testing, c) component test environment, and d) configuration management.

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn

C.5 Emerging techniques for component integration

Since 2000, a great number of papers address the problems and solutions in component-based software integration and system regression testing. They can be classified into three approaches:

– **Component-based interaction approach**

  A component interaction model, known as CIG, is used to present different types of interactions between components in component-based software. The model is used to represent three types of component interactions. They are: a) API-based interactions, b) event interactions between components, and c) message interaction between components.

– **UML-based approach**

  UML-based models (such as component collaboration and sequence diagrams) are used as system-level test model to support component integration and re-integration, as well as system regression testing.

– **Hybrid approach**

A graphic presentation model is used to present the combining white-box and black-box information from specification and implementation, respectively. This graphical representation can be used for test case identification and reuse, based on well-known structural techniques.

The major applications of these system-level test models for component-based software include: a) integration strategy identification, b) system-level change and impact identification, and c) system regression test selection and reuse.

In a component-based software product line, programs are built based on a set of software components. It includes third-party components, in-house components, and newly constructed application components. In fact, a product is an integration of customized components to meet the specific requirement set. There are two factors, which affect the complexity of component integration. The first is the number of involved components. The other is the customization capability of components.

Withdrawn
IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012

Annex D
(informative)

Example of software pre-use

D.1 System under consideration and operation experience

A software component realizes a supervision system as described by Figure D.1. It controls an analogue variable of an industrial plant based on two thresholds: one for warning and another one for plant shut-down. In both cases a digital signal plays a role, indicating whether the warning should be only optical or optical and acoustical, and whether the shutdown should be slow or fast. If the analogue signal is not valid, e.g., if it has noise-level only, another digital signal is checked, whether or not a warning should be issued, and a further one whether or not an alarm shall be given. As indicated in Figure D.1, a total of eight paths exist.

Experience is available by observing the software in operating with

– one supervision channel, and
– several parallel supervision channels.

Following, basic reliability theoretical background is summarized for calculating the reliability of the above described component for a reuse.

D.2 Reliability estimation based on sampling theory

Upper limit $\tilde{p}$ of unknown probability p of observing failures during operation can be estimated by using statistical sampling theory.

If we assume that successive operation runs are statistically independent Bernoulli trials, the number of failures in n operation runs distributes Binomial:

$$P(R=r)=C_{\mathrm{r}}^{\mathrm{n}}(p)^{\mathrm{r}}(1-p)^{\mathrm{n-r}} \tag{D.1}$$

where $P(R=r)$ is the probability of r observing failures in n operation runs, and $C_{\mathrm{r}}^{\mathrm{n}}$ denotes the number of combinations of r objects from n objects. Hence, the probability of 0 observing failures is:

$$P(R=0)=(1-p)^{\mathrm{n}} \tag{D.2}$$

For given any confidence level (β) and n correct operation runs, $\tilde{p}$ can be determined as:

$$(1-\tilde{p})^{\mathrm{n}}=1-\beta=\alpha \tag{D.3}$$

$$\tilde{p}=-\ln(\alpha)/n \tag{D.4}$$

An upper limit of its failure probability for each demand class C_{j} is:

$$p_{\mathrm{i}}=-\ln(\alpha)/n_{\mathrm{i}} \text{ with } i = 1,2, \ldots, k \tag{D.5}$$

IECNORM.COM : Click to view the full PDF of IEC PAS 62814:2012
Withdrawn