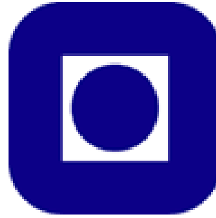
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK, OG ELEKTROTEKNIKK



HOVEDOPPGAVE

Kandidatens navn: Linda Kristiansen

Fag: Datateknikk

Oppgavens tittel (norsk): COTS komponenter i sikkerhetskritiske system

Oppgavens tittel (engelsk): COTS components in safety-critical systems

Oppgavens tekst:

The trend towards shorter time to marked for software application and the increased demand for more functionality makes the use of COTS components inevitable in most software systems. Safety-critical systems require the use of standards, such as IEC 61508, during development to ensure software quality and system safety. COTS components are usually not developed according to such standards.

Is it still possible and beneficial to use COTS components in safety-critical systems? If so, where does the trust in the COTS components come from, and how can the COTS component be certified for use in safety-critical systems?

Oppgaven gitt:	20. januar 2002
Besvarelsen leveres innen:	17. juni 2002
Besvarelsen levert:	14. juni 2002
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veileder:	Professor Tor Stålhane

Trondheim, 14. juni 2002

Professor Tor Stålhane
Faglærer

COTS components in safety-critical systems

Preface

This diploma is the final assignment in the educational degree of ”sivilingeniør” from NTNU. The problem was presented to me by my supervisor, Tor Stålhane, and the work has been sponsored by Lånekassen and my kind grandma.

This diploma work has been interesting, and challenging. I have learned a lot about my ability for individual work. Almost all prior large project experiences has been with project teams, and it has sometimes been difficult to feel the progress in my work, because of the lack of close project partners to discuss my problems with. I would therefore like to thank my supervisor, Professor Tor Stålhane, for arranging discussion meetings with many interesting and competent people, for giving me full feedback on my written work along the way, and for providing me with several interesting articles.

I would also like to thank the ”lunch-and-ice-cream-gang” for providing intellectual conversation during my many diploma breaks.

Trondheim, June 14th 2002

....
Linda Kristiansen

Contents

Abstract **vii**

1 Introduction **1**
1.1 Context . . . 2
1.2 Description of the problem . . . 2
1.3 Intentions . . . 3
1.4 Method . . . 3
1.5 Structure of the report . . . 4

2 The problem with using COTS in safety-critical systems **5**
2.1 Definitions . . . 6
2.2 Safety critical systems . . . 7
2.3 Commersial Off-The-Shelf, COTS . . . 10
2.4 Why is it important to use COTS? . . . 12
2.5 State of the art . . . 13
2.6 What could go wrong? . . . 14
2.7 Should COTS components perform critical operations? . . . 16
2.8 COTS based system topology . . . 18
2.9 So what are we up against? . . . 20

3 Certifying COTS **21**
3.1 Definitions . . . 22
3.2 The necessity of certifying components . . . 22
3.3 Different ways to certify a component . . . 23
3.4 Safety Case . . . 33
3.5 HAZOP . . . 40
3.6 What is minimum certification? . . . 43

4 Finding suitable COTS products **47**
4.1 Definitions . . . 48
4.2 What decides which product to choose? . . . 48
4.3 COTS and functionality . . . 54
4.4 Cost and benefit tradeoffs? . . . 55
4.5 What should the choice ultimately be based on? . . . 56

5 Restricting components through system design **59**
5.1 Definitions . . . 60
5.2 Wrapping . . . 60
5.3 Physical separation of components . . . 66
5.4 Redundancy . . . 66
5.5 Information polling . . . 69

6 Discussion **71**
6.1 Certifying COTS . . . 72
6.2 Suitable COTS components . . . 74
6.3 Restricting COTS components . . . 76
6.4 Cost/benefit . . . 78
6.5 Futher work . . . 79

7 Conclusion **81**

References **83**

List of Figures

2.1 Event chain, with the last event being critical 16
2.2 Event chain in a system containing a COTS component 17
2.3 Example of a simple COTS based software system 18
2.4 Example of a glued COTS based software system 19
2.5 Example of a layered COTS based software system 19

3.1 Overall safety lifecycle 24
3.2 Fault injection 29
3.3 The use of elements in a safety case 36
3.4 Hierarchic Argument Structure 37

4.1 The principle of decision making [46] 57

5.1 The basic principle of wrapping 61
5.2 The set of values to and from a COTS component in an ideal world 63
5.3 The set of known and unknown values to and from a COTS component 64
5.4 Wrapping the system 65
5.5 Using more than one COTS component for the same function 68
5.6 Information on request, quality ensured by a checkpoint 69

Abstract

The intention of this report is to discuss the possible advantages of using COTS components in safety-critical systems. The difficulties the combination of COTS with safety-critical systems may produce will be discussed, and the report will also give a discussion of the possibility of integrating and using COTS components in safety-critical systems. Certification of components and possible ways to restrict component through system design will also be discussed. The COTS marked is large, and cover a variety of components. This report will give a short explanation to some selection criteria you may use to locate the most suitable COTS component for a predefined requirement.

Safety-critical system needs to prove their safety to an assessor. These proofs are based on some evidence of how the system has been developed and tested. The proofs are not definite, and it will always be up to the assessor if he chooses to believe that the given evidence proves that the system is safe enough. This is also true if the system contains COTS components.

In a safety-critical system there are some functions labelled as safety-critical. The failure of these functions may endanger people or equipment. These functions should not be performed by COTS components, unless it can be proven that the COTS components has been developed under the same kind of control as a bespoke software component would have been.

COTS components are not built for certification. They are not developed according to a safety standard, and they do not follow any standard's requirement for documentation. The COTS code will in most cases be hidden from the COTS user, and the documentation of the component will not have the same degree of quality as documentation especially intended for safety-critical applications. COTS components are hence black boxes to their users, and the greatest concern about the component is not what you know it can do. The things you do not know that the component is capable of are far more difficult to handle. By testing the component thoroughly and protecting the rest of the system from the component through system design, it should still be possible to use COTS components for less critical operations.

The problem of lack of insight into the programming code present in most COTS components could be avoided by using Open Source Software. Open Source Software should be easier to certify than regular COTS components, due to the access to see and alter the code of the components.

COTS components should not be used in safety-critical systems unless some benefit of doing it could be proven. The benefits of using COTS could be saving money on development, saving time coding the component, or taking advantage of the fact that other people's expertise sometimes exceeds your own. These benefits may in some cases be difficult to prove. COTS component are more difficult to test and certify, and hence more expensive to certify than bespoke software. COTS components also needs to be restricted in the safety-critical system. This increase integration costs in the application.

Chapter 1

Introduction

Contents

1.1 Context 2
1.2 Description of the problem 2
1.3 Intentions 3
1.4 Method 3
1.5 Structure of the report 4

1.1 Context

This report is the result of the diploma work of Linda Kristiansen, done at the department for Computer and Information Science (IDI) at The Norwegian University for Science and Technology (NTNU).

The diploma covers 10 credits (Vekttall), and has a time frame of 20 weeks. The diploma was started January 20th 2002, and delivered June 17th 2002.

Professor Tor Stålhane has supervised the diploma work.

1.2 Description of the problem

Developing software is not an exact science. There are many ways to reach the goal of implementing a given set of requirements, and there are different demands for different systems. Some systems need a high degree of security, such as medical journals or bank transaction systems. Some need high usability, such as check-in machines at airports or automatic ticket sales machines at train stations, and some need a high degree of safety, such as airspace control systems. The latter type of systems are called safety-critical system, since a failure in these systems may cause harm to persons or equipment.

Software reuse is easy to justify. Even the best programmers only produce 10 lines of documented and tested code per day, which for large systems (typically more than 100 000 lines of code) have made custom software development expensive [30]. The demand for shorter time to market and high functionality for software systems has made reuse of components common. It has become a large industry to develop and sell components, and such components are called Commercial Off-The-Shelf (COTS) components. It is common to use such components as part of many software systems, so that you do not have to invent the wheel all over again with every system you implement. Someone has implemented the requirements for you, and you can purchase it as a component package.

A system crash due to a COTS component is usually not considered critical for the average COTS user. How many times have you had to restart your computer due to a crash somewhere in, e.g. the Windows 2000 operating system? Usually it is not a big crisis. You lost the last letter to aunt Marge, but other than that, everything is intact, and back to normal functioning mode after a "quick" reboot.

This is, however, not the case for a safety-critical system. A system failure is a big deal, since it might endanger the health, or even the life of someone. Therefore, system failures have to be avoided in safety-critical systems.

COTS components are valuable elements in many software systems, but is it possible to use these components in a safety-critical system? What are the limitations with COTS components in regard to safety-criticality, and will the use of such components contribute positively in a safety-critical environment? What do the developer of the safety-critical system need to keep in mind, if he chooses to use COTS components in his system? From where will he find the trust he needs in the COTS component?

These are the main problems that will be discussed in this report. The focus of the report content will be on how to find suitable COTS component, how to certify them for use in safety-critical systems, and how to restrict them through system design, so that they will not cause a system failure if they fail.

1.3 Intentions

The intention with this report is to discuss how developers of safety-critical systems can trust COTS components enough to use them in their systems. The report will also discuss how developers can take precaution when integrating COTS component into their systems.

The report will also suggest a method for certifying COTS components for use in safety-critical systems.

Presumed advantages with the use of COTS component will not necessarily be present when the components are used in safety-critical systems. This report will discuss some of the tradeoffs between cost and benefit the developer needs to consider before using COTS components in a safety-critical system.

The report will also try to give an answer to the question: is it advisable to use COTS components in safety-critical systems?

1.4 Method

The discussions and results in this report are mainly based on the study of relevant literature. Several articles has been located and read.

The supervisor for the diploma has also contacted people with relevant experience. It has been arranged meetings with these people, and the problems have been discussed. The problems have also been discussed regularly with the supervisor during the whole period of work.

1.5 Structure of the report

Chapter 2 contains the background knowledge needed to understand the problem area. The chapter discusses what a safety-critical system is, and what COTS components are. The chapter tries to justify the use of COTS components, and it gives a short survey of the state of the art within use of COTS components. The last sections of the chapter discusses how COTS components can be integrated into a system, and the problems such integration into safety-critical systems may cause.

Chapter 3 discusses the certification process for a COTS component. It gives reasons why a component should be certified and how this may be done. The chapter then suggests the minimum certification a component needs before it can be integrated into a safety-critical system.

Chapter 4 describes the criteria you may use, when you want to find a suitable COTS component for a specific requirement. The chapter also discusses how a possible mismatch between your need and the functionality of the component may compromise the safety of your system. A discussion of how the costs of using the COTS component should be compensated by the benefits of using it can also be found. The chapter ends with a discussion of what the choice of COTS component should be based on.

Chapter 5 discusses the methods to improve safety of the system that could be used in the design of the system. It describes several methods to restrict the component within the safety-critical system.

A discussion summarising the most important arguments is given in chapter 6, and chapter 7 contains the most important findings.

Chapter 2

The problem with using COTS in safety-critical systems

Contents

2.1 Definitions 6
2.2 Safety critical systems 7
2.2.1 Threats to safety 8
2.2.2 How to achieve safety 8
2.2.3 Process definition 9
2.2.4 The development process 10
2.3 Commersial Off-The-Shelf, COTS 10
2.4 Why is it important to use COTS? 12
2.5 State of the art 13
2.6 What could go wrong? 14
2.7 Should COTS components perform critical operations? 16
2.8 COTS based system topology 18
2.9 So what are we up against? 20

This chapter will give a short introduction to what a safety-critical system is, what the major threats to such systems are, and how safety can be achieved in a safety-critical application. The chapter will also give a definition to the term COTS, and discuss why, if at all, such components should be used in a safety-critical system. A short survey of the state of the art within use of COTS will be given in section 2.5. Problems that could arise from trying to integrate COTS components into safety-critical systems will also be discussed.

Using COTS in a safety-critical system does not have to be a question of use of not to use. Section 2.7 discusses how critical an operation in a system can be, if a COTS component is to perform it. How one could define a system by the way their components interact with each other and the environment, will be discussed in 2.8. The last section in this chapter gives a short summary of what must be resolved before COTS components can be used in a safety-critical application.

2.1 Definitions

COTS -	Commercial Off-The-Shelf, see section 2.3
Harm -	irremediable or irrecoverable damage [1].
Error -	An error is that part of a system which is likely to lead to a failure [7].
Fault -	The hypothesized or attributed cause of an error is a fault [7].
Failure -	A failure of a system occurs when the delivered service deviates from implementing the system function, that is, what it is intended to do [7].
Safety -	freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment [2].
Safety case -	The documentation of the reasons why the system is believed to be safe enough to be deployed. It reflects the design and assessment work carried out in the development process [1].
Safety-critical service -	A Service is judged to be safety-critical in a given context if its behaviour could be sufficient to cause the controlled equipment to inflict, or prevent the equipment from inflicting, harm on resources for which the organisation operating the service has responsibility [1].
Safety-critical System -	A safety-critical system in one that has at least one safety-critical service [1].
SOUP -	Software Of Uncertain Pedigree.

A safety-critical service can be both systems intended for doing harm, such as weapons systems, and systems intended to protect something from harm, such as a nuclear reactor protection system. For the rest of this report the definition of safety-critical service will only mean systems intended to protect something from harm. This is done in order to keep the argumentation clear and easy to follow.

The definition for a safety-critical systems is only applicable to computer systems, due to the definition of a safety-critical service.

There are two sorts of safety:

- *Intrinsic* safety - a system is intrinsically safe when there is no possibility of it causing absolute harm;
- *Engineered* safety - this term applies when a system has been designed to minimize risk, or reduce it to an acceptable level.

In practice the systems of most concern are those that have to be engineered to achieve acceptable levels of safety. This report will focus on engineered safety.

2.2 Safety critical systems

Computer systems, and hence software, can only influence safety if they are used to control some physical process which can lead to harm. A computer system offers services to a controlled equipment (also known as the Equipment Under Control, EUC). Computer systems operate in order to satisfy some goal in the management of the equipment that the computer system is designed to control or influence. All systems which can influence safety can be viewed as safety-critical [1].

A system can be both incorrect and unreliable without being unsafe, if a failure in the system will not be handled and hence be hazardous. A system can also be correct, but unsafe, due to errors in the specification.

A system has to be safe even in the presence of failure. Absolute safety is a unobtainable goal, and it is a matter of opinion if enough has been done to achieve acceptably safe operation. The problem in general is to be sure that all serious problems the system can handle.

During development of safety-critical systems, it is necessary to have [1]:

- understanding of the system, its environment, its working, its failure modes, etc.
- Independent assessment (confirmation) of the design, implementation, failure analysis, etc.

Safety is at least as much a management issue as it is a technical one. For project management issues that means to control the safety lies within:

- Control over the process, e.g. design visibility
- Documentation of the process, e.g. safety plans and hazard logs.

2.2.1 Threats to safety

There will always be some problems when developing safety-critical systems [1].

- specificability - the difficulty of producing adequate and effective specifications.
- complexity - the difficulty of assessing, measuring and controlling it.
- unanticipated combination of events - the difficulties of handling such events.
- human frailty - people make mistakes which can lead to accidents.

All these issues are sources of problems in practice, and cannot be entirely overcome. They will always be present to some degree, and the main aim when designing safety-critical systems is to minimize the risk that they will lead to an unacceptable safety level for the system produced.

The managerial enemies of safety are mainly concerned with aspects of human frailty. One of the problems is *unfamiliarity.* Project managers in industries developing computer-based safety-critical system can be unfamiliar with software. Project managers also have to follow a predefined budget for the development. Exceeding the budget may lead to reduced sales of the products being produced. The other prospective of unfamiliarity is that the software developers may have insufficient knowledge about the problems the system is supposed to control. They lack the knowledge of the specific industry area.

Managers of project also have to face the problems of *inappropriate standards.* Some of the standards to which the products and processes have to conform are inappropriate, or outdated. Developers have a responsibility to achieve safety, and conformance to standards. They may therefore have to carry out one set of activities to achieve certification, and another to achieve safety. This inevitably cost money and human resources.

2.2.2 How to achieve safety

In order to master the problems described in section 2.2.1 a number of techniques can be deployed. These techniques cover both specific design and analysis techniques. The also cover "safety principles", i.e. concepts and ideas which guide the way a system should be designed and assessed.

One of the most fundamental principles is *simplicity*. This is the only real way to ensure understandability of the system being produced. It is an easy principle to state, but a hard one to achieve. The best counter-measure to this problem is to ensure that there is a simple and precise statement of requirements for the system.

If a system cannot be made sufficiently simple, it can be due to the use of an inappropriate *structure*. Structure is crucial. From a computer systems point of view, the software and hardware system architecture is probably the most important aspect of structuring.

It is possible to use *mathematical rigour* to understand the system. With this principle one uses rigorous mathematical models (and automation) to clarify the meaning with the system. This reduces the opportunity for human failures. Mathematical techniques can be used to analyse the timing behaviour of programs, and to analyse scheduling properties.

Failure mode analysis is a vital aspect of the assessment of safety-critical systems. This method can be used to investigate the behaviour of the system once a failure has occurred.

It is also crucial to ensure that *comprehensive validation* is performed. This validation should be done both during development and in operation. The validation process involves review, testing, and operational usage.

The project managerial difficulties are primarily related to people and organisations. One fundamental issue in all development processes is to find *good people*. People with the right skills and attitudes are necessary to achieve safety. In a software engineering context, knowledge of the principles of specification, verification and testing is important. The experience of the staff is also a major concern because it is often the knowledge about the problem domain, not abstract knowledge of computer science that helps the staff detect flaws in the specifications.

2.2.3 Process definition

The definition of the development process is one of the most important aspects of achieving safety. *Definition of a process* is a document recording the key facets of the process, including stages and phases, particularly identifying and specifying safety-specific activities. This definition is important because it creates the basis for demonstrating that an adequate level of safety has been achieved.

The process will include hazard analysis, failure modes and effect analysis, establishment of integrity level for the system and software components. The definition should state which techniques to use and identify the requested skills and experience of the staff involved.

The process definition also defines the *choice of developmental technologies*. This is the use of particular techniques, e.g. formal methods, for particular activities and components, as demanded by the integrity level. Generic standards such as those developed by International Electronical Commission (IEC), are relevant sources of information.

A *safety case* is the documentation of the reasons why the system is believed to be safe enough to be deployed and it reflects the design and assessment work carried out in the development process. In many cases the safety case will be the major deliverable to the certification process or to an independent assessment process, see more about safety cases in section 3.4.1.

2.2.4 The development process

The development process for a safety-critical system is as follows:

1. Identify potential hazards.
2. Design and verify the system to show that hazards will not arise.
3. Analyse possible failure modes and show that the safety is maintained even in the presence of failure - this is the key aspect of validation.
4. Control the development process and produce documentary evidence so that it manifest that you have done steps 1, 2 and 3 properly, both in absolute terms and against relevant standards.

2.3 Commersial Off-The-Shelf, COTS

The concept of COTS covers a wide range of products, with different characteristics. It is a concept used by many, with different intended meanings. It can be a software component that forms part of a program (such as a library), it can denominate a standalone program or utility, or a high-level service that interact with multiple programs (e.g. operating systems kernels or web servers). COTS can also be used as a term for an entire system, where hardware and software are integrated [3].

Another concept similar to COTS is SOUP. SOUP stands for Software Of Uncertain Pedigree, and for the purpose of this report SOUP and COTS will be taken to mean the same thing. The reason for doing this is that in the context of a safety-critical system, the commercial products used, will in most cases also be of uncertain pedigree, and the minor differences there might be between the definitions of the two terms, will have little impact on the discussions in this report.

Many have tried to make a definition of COTS, but this has turned out to be difficult, since COTS is a term that is meant to cover many different products. According to the Federal Acquisition Regulation/ Oberndorf, the main characteristics of COTS is that it [4]:

- exists a priori.
- is available to the general public.
- can be bought, or leased or licensed.

Another definition states that COTS software has the following characteristics [4]:

- the buyer has no access to the source code.
- the vendor controls its development.
- it has a nontrivial installed base, i.e., more than one customer; more than a few copies.

In [3] it is stated that the main characteristics of SOUP are that it:

- already exists and cannot be re-engineered.
- is generic and hence may contain functions that are unnecessary for the system application.
- is subject to continuous change. A mass market SOUP will evolve to meet customer demands and to match the competition.

None of these definitions cover all possible types of software components. The notion of Open Source Software (OSS) is not considered in any of the definitions. In this report the definition used for COTS and SOUP will be:

- it exists a priori
- it is available to the general public
- it is generic and hence may contain functions that are unnecessary for the system application
- it is often subject to continuous change
- the vendor controls its development

The definition does not explicitly mention access to source code. This is because both component that do grant access to source code, and component that do not grant access to source code should be covered by the definition.

2.4 Why is it important to use COTS?

The trend in software development is to shorten the time-to-market. This demands reusability and component based system development. Systems are broken down into modules based on function. Such modules can be reused in other systems demanding the same function, even if the context is different. By reusing modules, developmental effort is saved, and the time used to develop the system is shortened.

COTS can be considered to be reusable modules, and the increasing demand on shorter time-to-market makes it difficult to develop a system without using some sort of COTS components.

When developing a safety-critical system, or any system for that matter, you may encounter problems or problem areas you have no previous experience with. In these situations it could be wise to seek the expertise of others, since it could take time to master the new problem area, and you wish to spend this time elsewhere. Using COTS components could then be an attractive alternative. The developers of COTS components know the product they are developing and they are familiar with the problem domain. Even if you would prefer more control over the product (for use in a safety-critical system) it is not certain you could do a better job than the COTS developers, even if you tried. It would therefore in some cases be advisable to use COTS components, and use your own resources on areas where you are the expert, and thus spending the efforts where it would increase your own competitiveness on the software marked.

The COTS market is enormous, and no matter what your needs for functionality are, you will more likely than not find that someone else has attempted to solve the same problem, or at least part of it. It is therefore likely that some existing product will provide parts of the functionality needed in any new system.

In many cases the use of COTS will reduce developmental costs. Saved lines of code to write, means saved cost on development. This is, however, not a simple equation and there might be hidden costs. For example, all parts of the system needs to function, even in the presence of errors and failure. When a COTS component is used in such a system, the component needs testing and a close examination. In many cases it is therefore important to consider the cost-benefit of using COTS. It could sometimes be cheaper to just write the component from scratch, instead of spending a lot of time examining COTS, and this should always be thought of before deciding to use a COTS component. On the other hand, if a COTS component has been certified for one safety-critical system, the effort needed to certifying the component for another safety-critical system should not be too extensive.

A COTS component often has a wide range of users. Many users from different environments has tested and used the component in/to many different tasks. This

provides a large test base. Errors have been reported to the developers and they have been corrected for future versions of the product. This can be both an advantage and a disadvantage. Since many have used the product, the likelihood of undetected errors is small. On the other hand, the frequent product updates can cause trouble. Vendors often support their products, and this is obviously an advantage with COTS products, but the vendors typically only provide support for newer versions of the product. Thus, the users are forced to keep up with the new releases of products [31]. With safety-critical system this is bad news, since it will take time to certify one product version, and the change to a new version is not necessarily trivial, because the new version most likely would have to be treated as a totally new product, and hence certified from scratch.

The development process for COTS components is often hasty. This may lead to errors, but many vendors still release product in order to be able to stay in the competition for customers. Such errors are correctable with new releases or downloadable fixes [5]. For safety-critical systems, this is not a good thing. The best thing would be to have correct and stable products from the start. It might therefore not be advisable to use "new" products in safety-critical systems, but rather wait for more stable versions to be released.

COTS components have often a large user base, and in order to keep it this way, a lot of effort has been put into the usability of the products. Again - this can be both an advantage and a disadvantage. In order to keep the products user friendly, much of the more sophisticated functionality may be hidden for the user. In safety-critical system this can cause trouble, because the hidden functions could lead to problems, since the user cannot inspect these functions thoroughly. On the other hand, the ease of use makes the products just that, easy to use, and it also becomes easier to find people with experience using the component.

2.5 State of the art

COTS can mean a vide variety of component types. Databases, operating systems, web browsers, word processing programs, spreadsheets, device drivers, libraries for programming languages, and much more. Most COTS used is at the large software component level, which includes GUIs, operating systems, and databases [35].

COTS as libraries for programming languages are also widely in use. The implementation of mathematical functions, such as sine and cosine already exist. With object oriented languages the most used objects, e.g. list and string, already exist and is accessible to users of the programming language.

The quality of COTS components is often only as good as the demand for the components. Even the best systems will contian 3-6 fault per KSLOC (1000 Source Lines

Of Code) [30], and the industry standard for good commercial software is around 6 defects per KSLOC in an overall range of 6-30 defects per KSLOC [36]. It would be difficult to sell components containig too many faults, since users of such component would loose patient if the component failed too many times. The average COTS user can, however, tolerate that faults in a COTS component may lead to system failure once in a while. For safety-critical systems system failure is not desiarable.

The most important criteria for components used in safety-critical systems is that they are "proven" to be safe enough, that is, no single component should cause the system to fail. The most common way to prove this is to follow some sort of standard during development. For safety critical systems development and certification thus rely heavily on standards [9], such as IEC 61508. This standard requires a certain process, such as a defined life cycle, design and code review, testing planned and performed, work performed under a QMS (quality management system) and more [8]. For more information on the IEC 61508 standard, see section 3.3.1. There are often appointed assessors for safety-critical systems. These assessors task is to make sure that it is proven that a component has been developed according to some standard, and hence is safe enough to use in a safety-critical system.

For this reason COTS components are not commonly used in safety critical systems. COTS components can in most cases only be assessed in retrospect, and this is not good enough according to the standards. There are, however, examples of COTS components used in the Human-Computer Interface (HCI) of safety-critical systems. The argument for this is that many interfaces are familiar to users. E.g., a person used to the Windows HCI, with its icons and window structure, will perform safer on a system with a similar HCI, with functions similar to the once he knows, than if he should get used to an entirely new "type" of interface. Note that this is just an example, *not* an argument for using Windows in safety-critical systems!

2.6 What could go wrong?

A software system is a human made construction and faults in the system could thus be a result of humans' inability to master all aspects governing the behaviour of the system. An important step towards mastering a system is to understand all possible ways in which the system can fail. A system does not always fail in the same manner, and it is therefore useful to be able to classify the different types of failure in a system. This has lead to the notion of failure mode which can be characterized in accordance with three points of view: failure domain, perception of failures by the system users and consequences of failure on the system environment [7].

The failure domain leads to the distinction between value failures and timing failures [7]. Value failure covers the possibilities where the value of the service delivered does not permits the accomplishment of the system functions. That is, the value is illegal

according to some conditions, and the function is unable to complete. Sometimes the timing conditions for the delivery of the service no longer permit the accomplishment of the system function. Something is taking too long, and the rest of the system cannot wait for it. This is timing failure [7].

Perception of failures by the system users leads to the distinction between consistent and inconsistent failures. When all system users share the same perception of failures, the failure is consistent. When the system users have differently opinions about failures, the failure is inconsistent [7]. This point of view is not important in the study of COTS components in safety-critical systems.

The classification of failure consequences on the system environment leads to severity or gravity, of failures [7]. This classification therefore permits us to list failure modes according to level of severity. Examples could be minor failures and catastrophic failures.

In addition to timing and value failure unexpected access to memory can be a serious problem in a safety-critical system. Such access could for example be a pointer to a memory location it was not intended for, or a function out of control overwriting important data. Data stored in the wrong place are also at danger of being overwritten. Unexpected tampering with the interrupt vector is another memory failure that can lead to serious problems.

From this it is possible to divide the ways a system could fail in the following categories:

1. timing failure
2. value failure
3. memory failure

Each of these categories can be subdivided by grade of severity og gravity.

The principal danger posed by the use of COTS components in safety critical applications lies in the discontinuity it creates in the understanding of the system as a whole [19]. COTS components often appear as black boxes the developers have to deal with. This black box association is what causes the problem. The developer does not know what is inside the black box, and he does not know the reasoning and tactics used during the development of the component. Thus, the developer of a safety-critical system can analyse his own work, but where COTS components are involved the reasoning becomes weaker, since he cannot argue for the internal functions of the COTS component.

2.7 Should COTS components perform critical operations?

Using COTS components in a safety-critical system does not necessarily have to be a problem of use or not to use. It could also be a problem of where to use. A system can be complex, and not all of the functions in the system are critical. It could therefore be an idea to let COTS components perform functions not listed as critical.

One question that rises then is; how critical can an operation be, before we cannot let a COTS component perform it? As mentioned in section 2.2.1, the unanticipated order of event can be a threat to safety. Introducing COTS components into the system can increase this danger, since the component can have more functionality and perhaps more complexity than necessary.

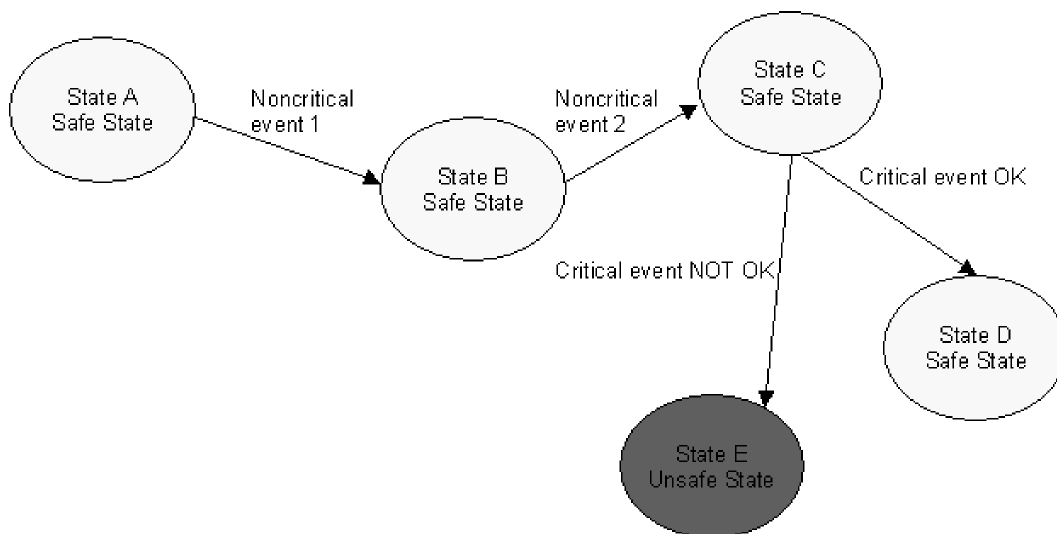


Figure 2.1: Event chain, with the last event being critical

It is difficult to draw a line, and say that functions less critical than this is okay for COTS components, and the rest of the functions should not be performed by COTS components. The criticality of functions and where and how they occur may vary from system to system, and are therefore difficult to generalise. In order to find out which functions COTS components could perform, and which not to perform, the system should be analysed. In this analysis there would appear several chains of events, some of which could lead to dangerous situations.

Figure 2.1 shows a chain of events, where the last event is listed as critical. In this chain a COTS component cannot be the last line of defence [5]. That is, the action

performed in state C should not be performed by a COTS component, since the next state for the system, depends on the outcome of that event, and an incorrect value may send the system into an unsafe state. In theory, event 1 and event 2 in figure 2.1 could be performed by COTS components, since the outcome of these events cannot send the system into an unsafe state. After the implementation of this chain in a safety-critical system, state E should be impossible to reach, even if the event in C where performed by a COTS component. The reason for advising against the use of COTS in this situation is that the COTS component is more difficult to control than custom written code would be in this situation.

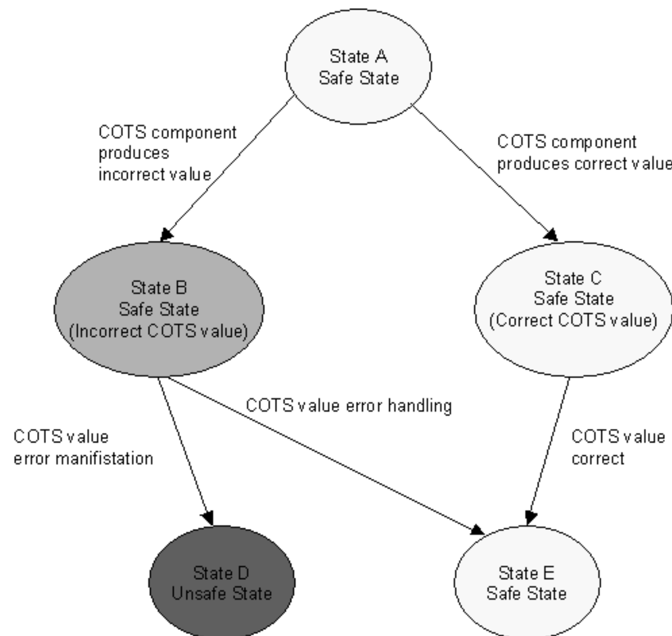


Figure 2.2: Event chain in a system containing a COTS component

Event chains are also the key to understanding the hazards in a system, since a fault not necessarily is produced by the last event, but are triggered by it. Figure 2.2 illustrates this point. An event sends the system away from state A. This event comes from a COTS component, and the value from the component can be either correct, or incorrect. Whether the value is correct or not has no visible effect on the next state in the system. State B and State C seem similar to a person observing the system from the outside, and both states are safe. The next event in the chain is, however, critical, even if it is not listed as critical from the analysis of the system, since the incorrect value from the COTS component is propagated further in the system. If the analysis did not anticipate the faulty value from the COTS component, the event would not be listed as critical, and error handling for the value would be missing, and hence, state D could be reached. If all possible faulty values from

the COTS component is handled in state B, it should be impossible to reach state D.

This illustrates the discontinuity COTS components create in the understanding of the system as a whole. By using a COTS component you introduce variables you have little or no control over. Imagine states similar to state B in figure 2.2, one for each possible faulty value from the COTS component. The possible value space would be large, and this increases the necessity for more error handling.

This means that even if an event is not listed as critical, COTS components could not be used without taking certain precautions, making sure that the component would not cascade errors further down the event chains, and hence create dangerous situation. Functions listed as critical should not be performed by COTS components.

2.8 COTS based system topology

As described in section 2.3, COTS is a wide term, covering a variety of components. It is therefore difficult to pre-specify system topologies. To bring some order into the chaos it is possible to order systems on how they interact with their surroundings.

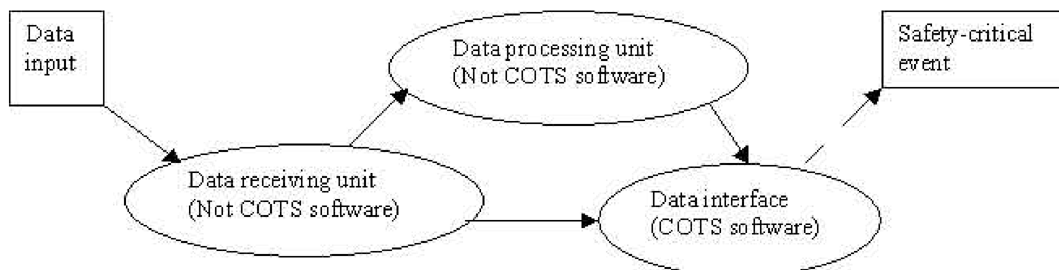


Figure 2.3: Example of a simple COTS based software system

[12] describes a simple COTS system, where some COTS components are combined with components which the user of the COTS have developed. It describes glued COTS systems, where some glue code has been used as an interface between two or more components. This glue code is provided by the development organisation, and most of the system functionality is provided by COTS components. [12] also describes layered COTS systems. In these systems the environment in which the system is embedded, consists of COTS software.

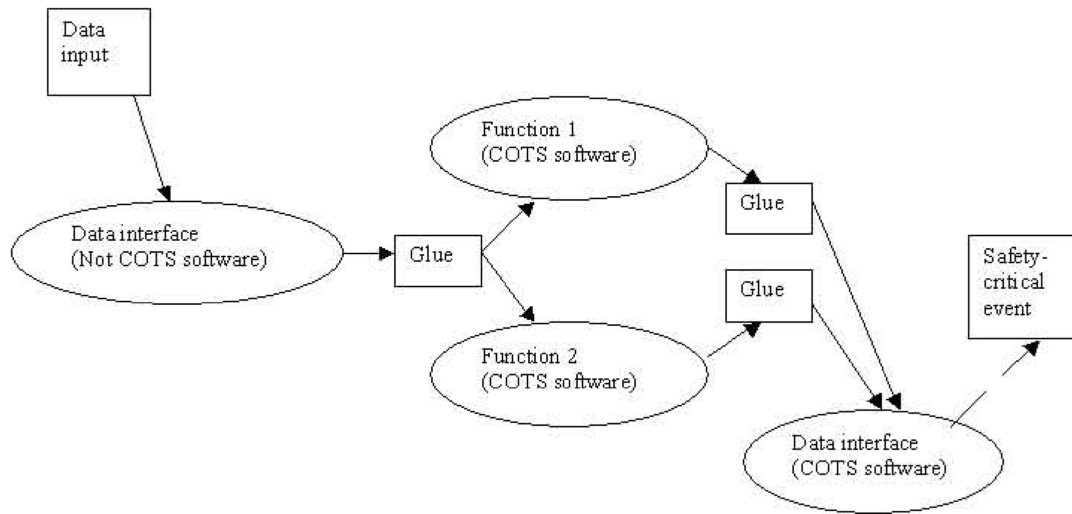


Figure 2.4: Example of a glued COTS based software system

Figure 2.3 on the facing page gives an example of a simple COTS based system. In this example the data interface is a COTS component, and it cooperates with other "in house" components.

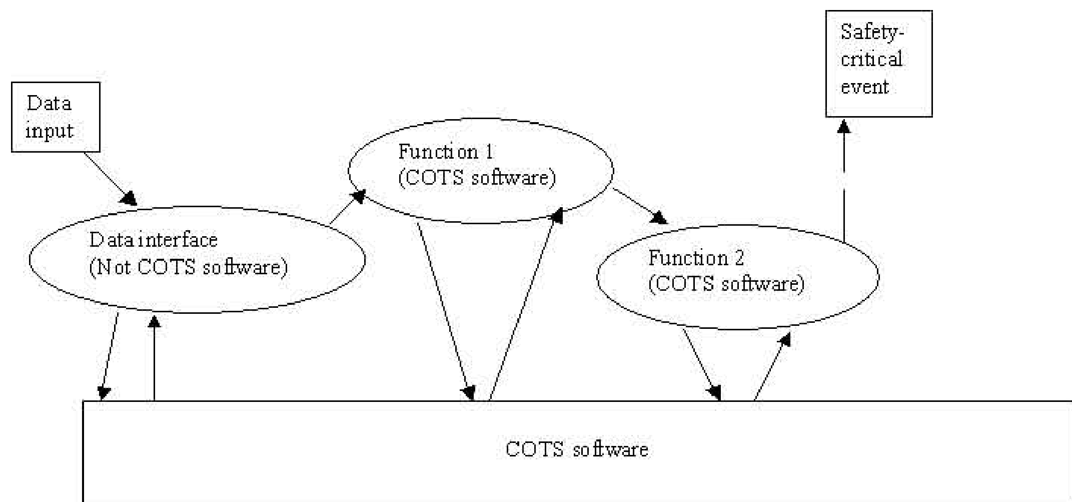


Figure 2.5: Example of a layered COTS based software system

Figure 2.4 gives an example of a system comprised of more than one COTS component. Each of the COTS components have been integrated into the system by developing glue code. The development organisation has supplied the glue code so that the components will cooperate.

Figure 2.5 on the preceding page gives an example where the environment in which the system is embedded, consists of COTS components. An example of such a COTS component could be an operating system, or a database.

2.9 So what are we up against?

Software in general is never perfect. From time to time errors are made. In safety-critical systems the main issue is to make sure that such errors do not cause the system to fail, and hence risk harm to people or equipment. In other words, safety-critical system needs to be able to tolerate faults, and return the system to a safe state if faults do occur. In order to guarantee such behaviour, the system developers needs to be extra aware of how the system is developed.

COTS components have become a necessary and invaluable part of many non-critical systems, but the expression "COTS" is itself not a trivial term. A definition such as the one in section 2.3 is needed to clarify.

Since COTS components bring indisputable contributions to non-critical systems, could they also be of some use in safety-critical systems? The problem of trying to combine safety-critical systems with software not initially intended for it is a divers problem with many degrees of freedom. Why would one want to use COTS components in safety-critical systems? What additional, or "new" problems could the use of COTS cause in a safety-critical system? What could a system comprised of both COTS components and in-house components look like? How much can we trust COTS components?

This chapter has discussed these problems, but in order to answer them thoroughly we need to take a closer look at the problem. Is it possible to certify COTS components? Is it difficult to find suitable components, and if you want to integrate one into a safety-critical system, how do you make sure it does not cause a serious failure in the system? The next chapters will try to elaborate on these questions.

Chapter 3

Certifying COTS

Contents

3.1 Definitions . . . 22
3.2 The necessity of certifying components . . . 22
3.3 Different ways to certify a component . . . 23
3.3.1 Process assessment, IEC 61508 . . . 23
3.3.2 Test methods . . . 25
3.3.2.1 Black box testing . . . 25
3.3.2.2 White box testing . . . 27
3.3.2.3 Fault injection . . . 27
3.3.3 Third party assessments . . . 30
3.3.4 Proven in use . . . 31
3.3.5 Custom built . . . 32
3.3.6 Certifying properties . . . 32
3.4 Safety Case . . . 33
3.4.1 What is a Safety Case? . . . 33
3.4.1.1 CELENEC EN 50129 safety case . . . 34
3.4.1.2 SHIP safety case . . . 36
3.4.2 COTS and Safety Case . . . 38
3.5 HAZOP . . . 40
3.5.1 What is HAZOP? . . . 40
3.5.2 COTS and HAZOP . . . 42
3.6 What is minimum certification? . . . 43

"It is very likely that there will be a large number of COTS in the Galileo system. However, studied standards do not provide definite guidance on how to justify the use of COTS in safety critical systems."

This statement is taken from [9], page 153. [9] describes several standards. These standards cover some of the most common application areas for safety-critical systems. In this chapter the necessity of certifying components before they are used in safety-critical systems will be elaborated. The chapter will also explain several ways to certify components and to establish if there is a demand for a minimum certification.

3.1 Definitions

assessor -	An independent person assigned to investigate the safety of a given system.
certification -	Documentation of a products suitability for a designated purpose. The assessor should be able to guarantee the product quality. The documentation should contain evidence of verification and validation of the software [47].
CBSE -	Component-Based Software Engineering.
HAZOP -	HAZard and OPerability study, see section 3.5.1.
Random failure -	Failure that can occur at any time and is due to a degeneration in the hardware [1].
Safety case -	The documentation of the reasons why the system is believed to be safe enough to be deployed. It reflects the design and assessment work carried out in the development process [1].
Systematic failure -	Failure that is the result of a fault in some stage of the life-cycle of a system. It can occur in either hardware or software [1]. Systematic failures will always occur if the same conditions are repeated [2]. All software failure are systematic failure [2].

3.2 The necessity of certifying components

More and more systems use the CBSE paradigm. That is, parts of the system consist of reused components. This is also true for safety-critical systems, but in safety-critical system it is important to be sure that the components act in a satisfactory manner. There should be some proof that the component do its designated tasks, and nothing else.

It is usually not the developer's intention that a system should fail, but it still happens sometimes. For obvious reasons, such failures are more serious in a safety-critical system, and failure must be avoided in such systems. To lower the likelihood of failure, the developer needs to be sure that all components in use are individually safe.

Unfortunately it is not possible to *prove*, in the mathematical sense of the word, that a component will never fail. The best one can do is to explain why it is not *likely* that a component will fail. The credibility of the explanation itself therefore becomes important. Developers need to be able to trust argumentation as to why the component most likely will not fail. Certification of components is a written argumentation for why the component is considered to be safe. Someone examines the component, and claims that it is safe enough to be used in a safety-critical system, but it is still up to the developer to decide if he wants to *trust* the certification or not.

Components with certification are inspected components. Someone has taken a long hard look at what they do, and how they do it, and has arrived at the conclusion that they do it well, and that the component is recommended for use under certain conditions. A component with a certification is more likely to be used, because the certification is a vote of confidence in the component.

3.3 Different ways to certify a component

There are several ways to certify components. Some methods are concentrated on the development process the component has been developed with, others focus on testing existing components. In order to certify, sometimes more than one of the methods described below have to be used. In some cases it is impossible to perform one or more of the methods. This section will focus on explaining the methods. The discussion on what is necessary in order to certify a component can be found in section 3.6 on page 43.

3.3.1 Process assessment, IEC 61508

The most common way to get components certified is to follow some predefined process. There are several standards describing such processes. It is common for each application sector (such as avionics, railways, nuclear, defence etc.) to have its own standard for certification of safety related systems. However, these standards share some common characteristics [9]. The IEC 61508 standard is a proposed generic evaluation approach for all safety lifecycle activities for systems composed of electrical/electronic/programmable electronic systems (E/E/PES) [9]. This report will focus mainly on the IEC 61508 standard.

IEC 61508 is built around the principle of a structured safety lifecycle. It identifies the safety lifecycle activities to be performed during the entire life of the target

system. An overall safety lifecycle is used as a technical framework for dealing systematically with the activities necessary to define the safety related requirements, to develop the safety related systems implementing these requirements, and to provide assurance of the satisfaction on the safety requirements at each phase on the lifecycle. Figure 3.1 describes the overall safety lifecycle as it is described in the standard [9].

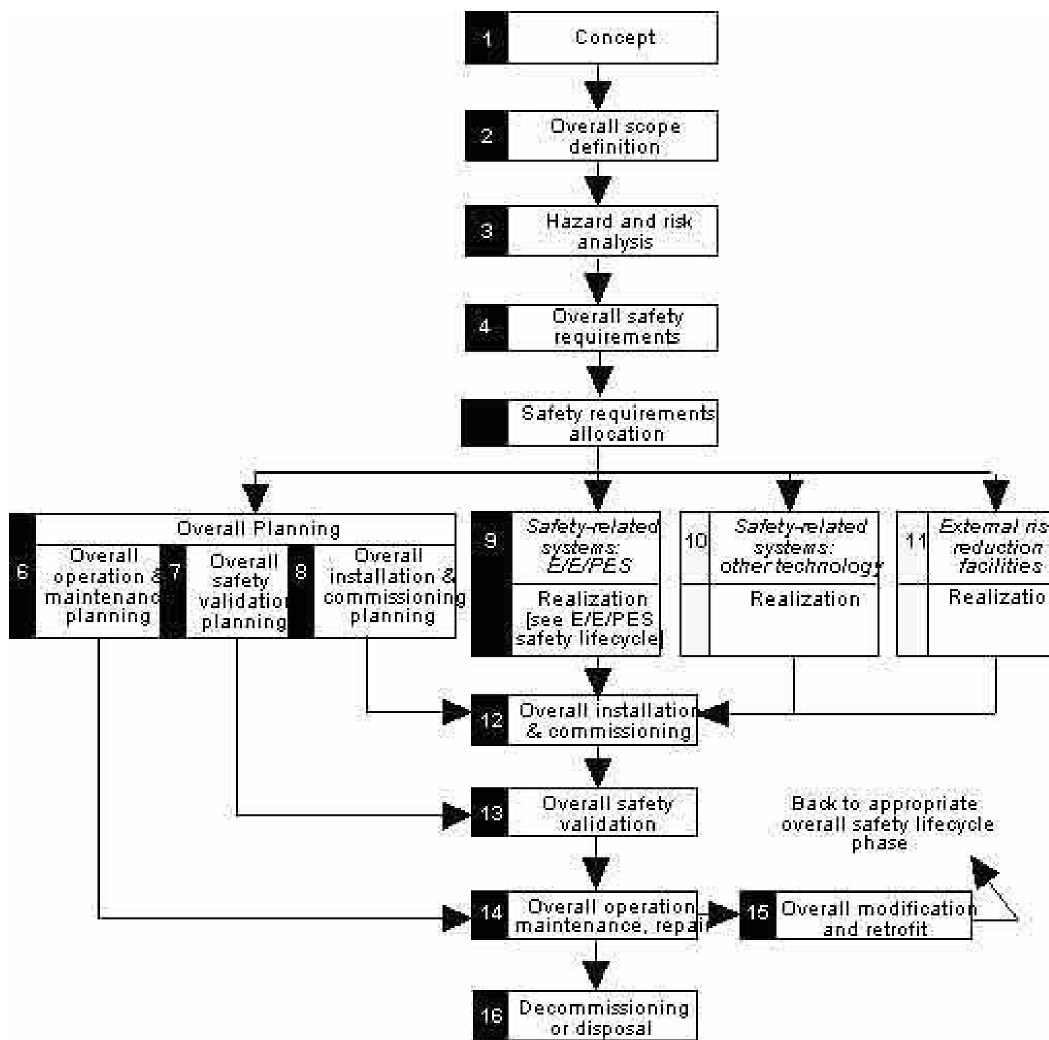


Figure 3.1: Overall safety lifecycle

For each safety lifecycle phase the IEC 61508 standard specifies [9]:

1. the objectives to be achieved;

2. the scope of the phase;
3. the requirements to be satisfied;
4. the required inputs to the phase;
5. the outputs required to comply with the requirements

The standard is not specially adapted to the use of COTS components. It does, however, not forbid the use of such components. The only requirement that is stated in the standard regarding previously developed software, is that such software should be clearly identified and its suitability in satisfying the software requirements be justified. COTS or other previously developed software will be suitable if there is enough evidence of satisfactory operation in a similar application (see section 3.3.4 for an explanation of "proven in use"), or the software has been subject to the same verification and validation procedures as would be expected for an newly developed software [9].

IEC 61508 is mainly concerned with the developmental *process*. By following the standard, a minimum level of quality is secured for the component, and hence the component can be certified. On the other hand, a product will only be as good as its developers, and following the IEC 61508 standard is no definitive *guarantee* for product quality of product safety.

Process assessment that leads to certification is a technique used during development of components. In some situations the standard dictates what to do also after integration.

3.3.2 Test methods

Methods for testing are also incorporated into some standards, and even if standards do not mention testing techniques, it would be a good idea to test a component extensively before it is used in a safety-critical systems.

With COTS components the version of the component tested should be frozen. Only the version tested may be used later in the system, since newer versions may include new, and possibly unwanted functionality, and old functions may have disappeared [35]. For the purpose of use in safety-critical systems, newer versions of a product will have to be treated as a totally new product, and therefore all tests have to be performed all over again.

3.3.2.1 Black box testing

This technique treats the testable object as a "black box". The technique is also known as functional testing [13]. In this technique the tester does not know the in-

ternal structure of the component tested, he only needs to know how the component is supposed to behave. Black box testing should be done according to the product specification or the specific project requirements; access to source code is unnecessary [16]. The tester imposes some input on the system, and checks the output. This means that the tester has to be able to tell if the output from the component deviates from the correct values.

The black box techniques gives some advantages [13]:

- The test is unbiased, since the designer of the component and the tester are independent of each other.
- The test cases can be describes as soon as the specification for the component is completed.
- The tester does not need knowledge of any specific programming language.

This means that persons with minimal technical skills can do the black box test. Expenses on highly educated personnel can therefore be better spent on other things than testing. It is in some cases possible to automize the tests. This would also free personnel from performing the tests.

There are, however, some disadvantages with using black box testing [13]:

- It is impossible to test every possible input.
- The test cases can be difficult to design, either because the component can take a wide variety of input, or because input can be combined in many ways.

With COTS components, black box testing is often the most appropriate testing technique, since the user in many cases will have limited access to the source code. Vendors of COTS components want to keep their trade secrets to themselves and force the user to focus on the functions of the component and not the implementation [16]. The black box technique can give an indication as to how well the component functions, but since it is impossible to test all possibilities, only a limited range of values can be tested. The technique can therefore not give indication of fault that the designers of the test cases not have anticipated.

Black box testing can be used to test both good behaviour of the component, as well as how the component reacts to illegal values.

Another problem with black box tests for COTS components is to create the test suit [15]. How do the tester know which values to test for? More often than not, the COTS vendors write a description of their component, which is too nice. They will avoid mentioning areas their component is weak, and hence leave the tester in the blind. In addition COTS developer will rarely give a written proof of black box

tests they have performed on the component.

For use in safety-critical system the black box test will be a valuable indication to the stability of COTS components. If a component cannot be tested with other techniques as well, the black box test alone will not be sufficient for certifying a COTS component.

3.3.2.2 White box testing

White box testing is also known as glass box, structural, clear box or open box testing [14].

With white box testing the tester uses explicit knowledge of the internal structure and the programming code of the tested component to examine output. The purpose of this sort of testing is to ensure that all of the program code has been executed at least once, and that it has not caused errors. The test is accurate only if the tester knows what the program is supposed to do [14]. White box testing does not account for errors caused by omission, and all visible code must also be understandable [14].

White box testing is not effective, and often not possible with most COTS systems, because necessary low-level insights are not possible [16]. The tester needs explicit knowledge as to what the code is supposed to do, and even if the COTS developer has performed white box tests for his component, the process and the result is almost never documented for the customer. According to [14], a complete software examination needs both white box and black box tests, and this is obviously a problem for COTS components. In order to use COTS in safety-critical systems, one needs some other ways to compensate for the lack of possibility to perform white box tests.

White box testing can, however, be used during development of wrappers around third party components [16]. For more information on wrappers, see chapter 5.2.

3.3.2.3 Fault injection

The purpose of fault injection is to insert faults into a program to test the effectiveness of the fault tolerance logic in the program, or its robustness properties. It may be achieved by modifying, recompileing and reruning the software (intrusive instrumentation). Fault injection can be more simply accomplished through data alterations at run-time by executing extra code outside the program being analysed (non-intrusive instrumentation)[9, 17].

An *anomalous event* (or anomaly) is a problematic event that occurs during software execution. During an anomalous event, the state of the software is altered. This alternation may or may not affect the output of the software. The number of different

anomalies that software can experience during its lifetime is large, and the size of the set is unknown. Anomalies of interest as candidates for fault injections are [17]:

- Problems that can arise from code defects (caused by programming errors or design defects).
- Problems related to human factor errors.
- Problems with corrupt data being read from stored files.
- Problems caused by external failures (e.g. hardware or other software upon which the program depends for inputs).

Software fault injection will answer the question on how likely it is that the software will not fail, even if it is defective. It also tells us how badly the software might behave if it fails [18].

If the fault injection process can be fine-tuned so that the results provided are clear and unambiguous, then it will be possible to predict *a priori* whether the future behaviour of a program will be acceptable [17].

The key concern for all software fault injection methods is that the information collected may not be relevant for the problems that the software will experience in the future, i.e., the results of fault injection will not hold for the real problems that will eventually manifest themselves [17].

Since the anomalies injected are *hypothesized* and their likelihood of future manifestation is unknown, the result from fault injection cannot be considered as an *absolute* measure of risk. Instead, the results are a *relative* measure of risk [17]. This means that you cannot be 100% sure that the faults you injected into the systems were the most relevant. It is also impossible to be sure that you have tested for all possible faults. Fault injection can, on the other hand, be assuring, because if the system could handle the fault you did inject, it is likely (not in the mathematical sense of the word, but as a belief) that the system can tolerate other faults as well.

Since it is not possible to inject every possible anomalous operational scenario and then observe the software's behaviour, it is necessary to somehow ensure that the anomalies injected are representative for the types of anomalies that the software will experience during its lifetime [17]. It might be a problem to find such a representative set, especially if the tester is not sure exactly what types of anomalies the system can experience during its lifetime.

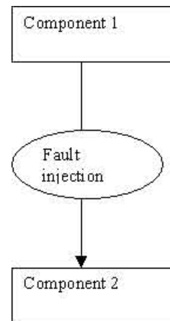


Figure 3.2: Fault injection

Fault injection has a limited applicability when it comes to COTS. More often than not, the code is not accessible, and this prevents intrusive instrumentation. It is however possible to interact with the interfaces to the COTS components, and in this manner control the values going in and coming out of the COTS. Hence, fault injection can be used to characterise the COTS dependability properties by analysing its behaviour in presence of faults in its environment or in presence of faults simulating or activating its internal faults [9].

Figure 3.2 shows the principle of fault injection used on a system containing COTS components. Say component 1 is a COTS component and component 2 is not. Fault injection becomes a layer between these two, corrupting the values passed through the interface between the components.

Fault injection has the ability to reveal secrets about the system. It is a method for testing the systems robustness, and these tests can sometimes provoke the system to display faults the system tester has not thought of. This is an advantage when testing COTS, because one of the problems with COTS is that when integrated into a system, they can cause the system to exhibit unanticipated behaviour. Fault injection can help reveal such behaviour.

One software fault injection method could be to arbitrarily flip bit values. This would corrupt the data in a way that could flush out software behaviour that in many cases is a total surprise for software developers and designers [27]. By matching such a arbitrary-randomness technique to the arbitrary randomness problem of undefined input spaces, the predictability about future software output behaviour could be increased [27].

Fault injection does not necessarily find errors in the code, but it can be used to test how well the component can deal with errors, i.e. the survivability of the component [28]. Fault injection provides assurance that your system can tolerate someone else's mistakes [29].

3.3.3 Third party assessments

When a component is to be certified *someone* has to do the inspection and testing to see if the component fulfils the strict requirements on safety. Who this someone is also has some importance.

When a developer chooses to use ready-made components, he can choose to check and certify the component himself. Done this way, the developer has full control over what has been tested, and how it has been done. The developer may, however, not be an expert on checking software for weaknesses, and by letting someone with more experience do the certification he may obtain a stronger certification, provided that he trusts the one who assess the component.

In such case, it would not be the developer of the component and not the user who ends up certifying it, but an independent third party. Like Voas does in [11], these third party agencies for certifying components will be called Software Certification Laboratories (SCL) in this report.

By getting an independent SCL to certify a component some advantages could be gained. The agency should be unbiased, and thus not try to hide weaknesses or emphasise strengths more than they deserve. An agency could certify components at request, or create a "library" of already certified components. More than one developer could be interested in such component, and thus creating a business for the SCL. With such libraries available, developers would save some time choosing and making sure the components were safe.

The world is, however, not perfect. Even if it could be possible to pick from a library of certified components, the variables concerning the *use* of such components are many. The certifier needs to establish the entire environment for the component in order to give the system developer the pieces of information he needs. The component, plus the certificate would therefore cover such a narrow area, that it would be difficult for others to use, and this would be bad business for SCL's.

Another cloud on the SCL sky is the liability that the SCL would have, if one of their certified components should fail. Any certifier carry some responsibility if the component they certified does not hold up to the expected standard, and in case of serious failure, there will always be a search for someone to blame for the tragedy that occurred. This might be the reason why software certification laboratories not have become a widespread business [11].

3.3.4 Proven in use

Many COTS products have achieved a high degree of quality. They are stable and reliable in their designated environments and they have few faults. Still, they cannot be used in safety-critical systems because it cannot be *proven* that they satisfy the strict requirements.

In [12] the concept of "proven in use" is described. It is a concept used to give confidence to a component by referring to the component's track record. IEC 61508 accepts the proven in use argumentation for using components in safety-critical systems. The standard has the following requirements for proven in use:

- unchanged specification;
- 10 systems in different applications;
- 10^5 operating hours and at least 1 year of service history.

Proven in use relies on the term "number of operating hours". The number of operating hours is defined as the sum of the operating hours of each installation that runs the software under consideration.

The proof to support proven in use is given through documentation. This documentation should at least contain:

- exact designation (*identification*) of the system and its components, including version control for hardware;
- users and time of application;
- operating hours;
- procedures for the selection of the system and application procured to (*used in*) the proof;
- procedures for fault detection and fault registration as well as fault removal.

It is possible to build the "proven in use" argument on data from the use of a component in a non-safety-critical application if the operating time or the number of demands according to safety integrity level is known. In addition a non safety-related failure rate less than:

- 10^{-2} per year with a confidence of 95 % based on 300 years of operating experience,
- 10^{-5} per year with a confidence of 99.9 % based on 690 000 years of operating experience;

must be demonstrated.

Proven in use can most likely be built on experience from non-critical systems, but it is not recommended that all prior experience stems from non-critical applications [12]. This is a dilemma, since the component should be proven in use in a safety-critical environment before it is used in a safety-critical system. But how do you get that usage, if you cannot use it without proving it in use first?

There are probably many COTS components that could easily satisfy the requirements for proven in use. The problem is, however, that nobody has bothered to *document* the usage. It is not profitable to document a component to this degree. Therefore, it is almost impossible to find a COTS component that will satisfy the proven in use requirements for documentation, and hence they cannot be used.

3.3.5 Custom built

Another possible way to get components certified is if they are custom made. In this case the component can fulfil some predefined requirement specification, and hence not include extra functionality. The buyer will know exactly what they get, and they will have someone to complain to, if the component produces faulty values, because they will have a contractor working for them, developing the component.

In the case of custom built components the most likely certification is to follow some standard, such as IEC 61508 (see section 3.3.1). This, however, is not likely to occur for COTS components. It violates many of the principles with COTS. The components will be highly specialised, and the user base will be limited, hence the components will become expensive. Having a component custom made, is in principle the same as using in-house components, and the certification methods used is not comparable to the once needed for certifying "regular" COTS components.

3.3.6 Certifying properties

To certify software components is not a trivial problem. In order for a COTS component to work properly in a system, there is more than one factor that should act correctly. The environment of a component has a great impact, and a certification of the component will mean nothing if the component does not fit in the intended environment. This component environment mismatch can become the cause of errors.

For this reason, the certification of a component will make little sense if not every important quality of the component has been defined. Identifying all factors that will have an impact on a component performance can in some cases be a huge task. Therefore, it may be possible, for each component, to identify a set of behavioural properties the component should satisfy, and then to certify that these properties

are satisfied by the component [34, 35]. I.e., an operating system supplier might certify that a lower-priority task does not interrupt a higher priority task as long as the higher priority task holds the resources required to continue processing.

If there for COTS components, could be documented such a set of properties the component satisfied, it would make the properties of the component more obvious to the user. It would probably make it easier to certify the component as well. On the other hand, it would demand more from the person selecting components for use. If the user is to trust such certificates without any other knowledge about the component, he needs to be clear in what he wants the component to do and not to do. He needs to identify a list of properties he does not want in his component, and then match this list against available certificates. Problematic properties he forgets to include in his list, may end up being part of the component he buys, and hence cause problems after integration into a larger system.

Finding a list of properties that will cover all the relevant properties in a COTS component seems like an impossible task. First, relevant properties may be different from component to component. Second, even if the relevant properties could be defined, it would be difficult to find generic categories that would cover all possibilities. Such generic categories would be needed if difference components should be compared. This leaves us with the same problem as testing the components would. It is difficult to handle the things you did not think of.

Another difficulty with using certification based on properties is that the certification would be difficult for someone other than the developer of the component to perform. There are few developers that issue such guarantees with their products.

3.4 Safety Case

3.4.1 What is a Safety Case?

A safety-case is a documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment [23].

The safety case is a line of argumentation, not just a collection of facts. It aims at convincing a licensing authority, who does not necessarily have a deep knowledge of the specific technological questions involved, that a given product or system is safe enough to be put into use. Or was safe enough - the safety case could be used as evidence in genuine legal proceedings years after the system was authorised and commissioned [26].

How to develop a safety case is not a precise science. Some standards, such as CELENEC EN 50129 (see section 3.4.1.1) gives a precise definition of what should

be in a safety case, but even if this procedure for creating a safety case is not followed, it is possible to call a document a safety case. IEC 61508 does not demand a safety case, as it is defined in this section. IEC does, however, require a "functional safety assessment" [23]. Other methodologies, such as the one described in 3.4.1.2, are also possible to follow in order to write a safety case for a component.

3.4.1.1 CELENEC EN 50129 safety case

CELENEC EN 50129 is a European standard for railway applications. It defines the activities for developers and manufacturers but also describes what a third party assessor shall verify [26]. This standard has a section describing safety cases and what a safety case should contain.

According to EN 50129, a safety case must contain enough information to give a clear impression of the system's safety properties and indicate where more details about the system can be found if this is needed. The safety case must contain descriptive text rather than just a more or less complete list of references [26].

EN 50129 also describes how a safety case should be structured [25]:

1. Definition of system
2. Quality Management Report
3. Safety Management Report
4. Technical Safety Report
5. Related Safety Case
6. Conclusion

From this list one can extract three main areas of approval, namely evidence of quality management, evidence of safety management and evidence of functional and technical evidence [9]. Each of the safety case sections needs a further description.

1. Section for Definition of system
The first section of a safety case shall precisely define the system to which the safety case refers. This means a description of what the system is, its functions and purpose, with reference to the requirement specification. The first section should also define the product structure, i.e. a document that identifies the components of which the system is made up, and the way they are related to each other and the overall system. A description of all interfaces is also a part of the definition of the system. Both external and internal interfaces, with references to corresponding documentation should be included. Issue, revision and date of all applicable documentation and/or versions of components or modules should also be included in the definition

of the system [26].

This means that any modification of the component in question, will necessitate a new or updated safety case [26].

2. Section for Quality Management Report
The second section of a safety case is a report describing what has been done to ensure that the system has the required quality throughout the entire life cycle. This part of a safety case should contain a description of the quality requirements, a description of the quality management system and a description of what has actually been done. The description of quality requirements must have references to the corresponding "source" documents. These source documents can be generic, internal company documents or laws or regulations [26].

The description of the quality management system needs a reference to the corresponding plans and procedures. This section must also contain a description of the project's organisation and identify by name the people in each position, their responsibilities and qualifications [26].

The description of what has actually been done should include references to e.g audit reports, minutes of meetings and any other documents of performed activities. Deviations from plans and procedures shall also be described and justified in this section [26].

The quality management report should describe the measures taken during development to ensure that the quality of the system is high enough.

3. Safety Management Report
The safety management report in a safety case should contain a brief summary of the safety requirements of the system, with reference to the safety requirements' specification. It should contain a description of the safety management system with reference to the corresponding plans and procedures and a description of what actually was done, with reference to e.g. hazard logs, safety audit reports or test reports [26]. The safety management report should describe the measurements taken during development to ensure that the required system safety has been achieved.

4. Technical Safety Report
In this section, the technical safety characteristics of the system are described. This section should identify which safety standards have been used, and any describe any underlying philosophy for achieving safety, e.g. the system architecture. The safety relevant properties of the system shall be presented and references made to the corresponding evidence, e.g. test and analysis results, verification and validation reports, certificates and so on [26].

5. Related Safety Cases
If the system's safety relies on the use of safe parts or components, the corresponding safety cases shall be identified in this section of the system safety case. Restrictions or application conditions mentioned in other relevant safety cases shall be recapitulated in this section. "Related safety cases" can also refer to certificates for parts or components, since such will themselves be based on documentary evidence of the relevant safety properties [26].

6. Conclusion
In conclusion of a EN 50129 safety case the evidences presented in the earlier sections of the safety case are summarised. The section contains the arguments for why the relevant system/subsystem/equipment is adequately safe [26].

3.4.1.2 SHIP safety case

[23] and [24] presents another safety case methodology. This idea for safety case was initially developed in the EU-sponsored project SHIP. The concepts were further developed in the UK Nuclear Safety Research Programme (the QUARC Project). Some of the concepts have subsequently been incorporated into safety standards such as MOD Def Stan 00-55, and have also been used to establish specific safety cases for clients [23]. To implement a safety case [23] finds it is necessary to:

- Make an explicit set of claims about the system.
- Produce the supporting evidence.
- Provide a set of safety arguments that link the claims to the evidence.
- Make clear the assumptions and judgements underlying the arguments.
- Allow different viewpoints and levels of detail.

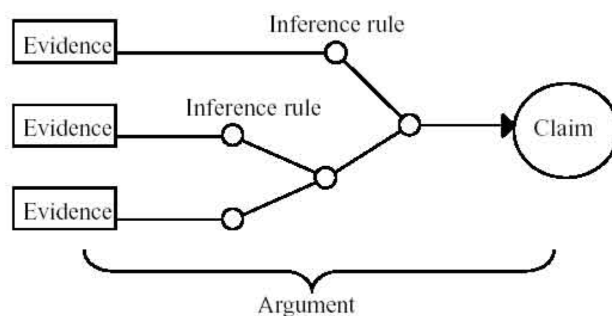


Figure 3.3: The use of elements in a safety case

The main elements of a safety case are [23] are shown in figure 3.3:

- *Claim* about a property of the system or some subsystem.
- *Evidence* which is used as the basis of the safety argument. This can be either facts, (e.g. based on established scientific principles and prior research), assumptions, or subclaims, derived from a lower-level sub-argument.
- *Argument* linking the evidence to the claim. The argument can be deterministic, probabilistic or qualitative.
- *Inference* is the mechanism that provides the transformational rules for the argument.

In practice it is unlikely that any safety case will be entirely deterministic or probabilistic. It may consist of a number of claims about specific properties of the system. In addition it needs to be viewed at various levels of detail. It is proposed that a safety case can be structured as a hierarchy of claims as shown in figure 3.4 [24].

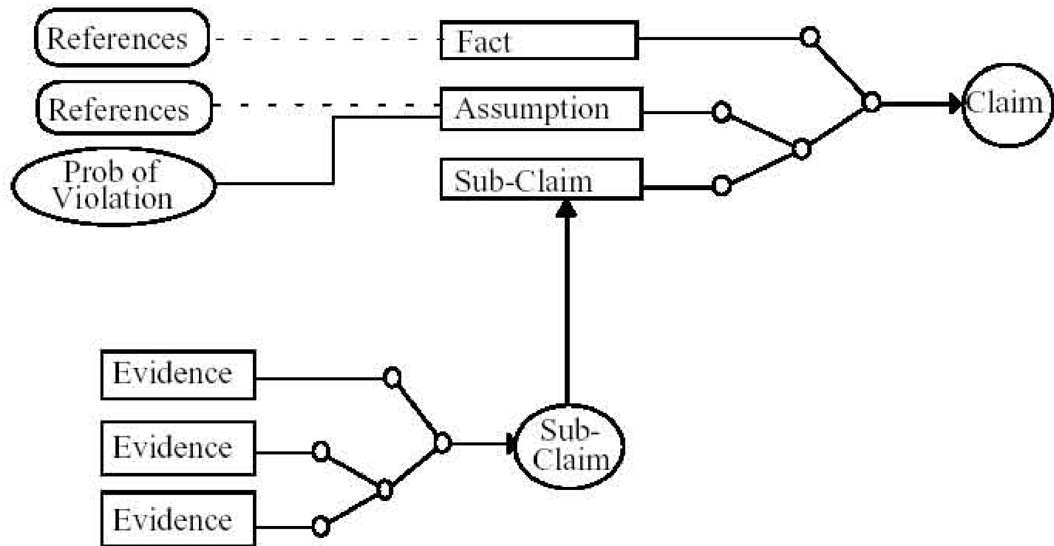


Figure 3.4: Hierarchic Argument Structure

The SHIP‘s arguments can utilise evidence from the design, the development processes, simulated experience (e.g. reliability testing) or prior field experience. This separates the SHIP method from the EN 50129. EN 50129 has a strict set of rules for what should be in the safety case, and hence lay some limits on *how* the safety evidence is given. The SHIP methodology gives four possible ways to obtain evidence to support the safety claim about the system (design, process, testing and prior experience). It does not dictate the use of one or the other source of evidence. It would

make a stronger argument if more than one source of evidence were to be used to support the same claim, but it is not a *requirement*. Using the SHIP methodology, the choice of argument will depend in part on the availability of the different "types" of evidence [23].

3.4.2 COTS and Safety Case

How important a safety case is for certifying a component highly depends on the quality of the safety case. Ultimately it will be a matter of opinion whether the component is safe enough to be used in a safety critical system or not. If the assessor of a component finds the safety case believable, the safety case is a useful tool in the certification process of COTS components.

It is assumed for the rest of this section that the safety case should be developet by the user of a COTS component. The developers and the users of the component are not the same. It is also assumed that both of the two safety case methods described in 3.4.1 is possible to write, during the development of a safety critical system not containing any previously made components.

Building a safety case for a COTS component could turn out to be difficult, especially if the EN 50129 standard is to be followed. The first section of the EN 50129 safety case should be a substantial part in any safety case. EN 50129 states that the components function and purpose should be stated, *with reference to requirement specification*. In any attempt to certify a component, the components functions and purpose should be explained, but for a COTS component the *original* requirement specification is rarely available. A requirement specification written by the COTS user will not be an adequate substitution, since the user of the component will not have total control over the components functionality. The first section of a EN 50129 safety case would therefore in many cases be difficult to fulfil for a COTS component. The intent of EN 50129 is to ensure that the final implementation covers the intent. The requirement specification created by a user, after the functionality of the COTS component has been implemented, could never take the place of the original requirement specification, because it does not express the initial intent of the component. It could, however, be possible to write a less detailed version of the section for definition of the system.

The EN 50129 section for quality management would be difficult to write for a user of a COTS component. This section requires the knowledge of the entire lifecycle of the component. The quality management report should also make references to source documents where procedures and plans for quality management of the component throughout the components lifecycle are described. Even if such documents exist for COTS components, it is more likely than not that the documents not will be distributed with the component. The COTS developers will probably try to keep their internal procedures secret, in order to keep others from "stealing" their success.

In order to write the quality management report we need more information than is usually available for COTS components.

In the safety requirement report of EN 50129 there should be a reference to the safety requirement specification for the system under investigation. Again, this could be difficult for COTS components, since the developmental procedures and documents are not available for the persons writing the safety case. It could, however, be possible to write a less detailed version of the safety requirement report, by documenting the hazards that the component gives based on for example a HAZOP (see section 3.5), or documented tests performed on the component.

EN 50129‘s technical safety report requires a more specified documentation of the system than would be possible for most COTS. The architecture of the system, the safety standards used (if one is used at all), and underlying safety philosophy would in the COTS case be difficult to obtain, and hence be difficult to use as evidence for the component's safety. It could be possible to write a "light" version of the technical report for COTS components, but it would then be more difficult to trust the safety case, since the evidence would be weaker. The technical report could refer to test reports an/or certificates for a component. This would probably be possible to achieve, even for COTS components, but then you would be dependant on a strong certificate for the component. See section 3.6 for a more detailed description of certification.

The "related safety cases" section of a EN 50129 safety case, could be important for a safety case on a system containing a COTS component. During the development of a safety critical system it would be possible to follow the EN 50129 description of how to build a safety case, given that the system does not contain any previously used components. If the safety critical system were to be made up from already made components, it is still possible to write a safety case for such a system, given that each of the components come with its own safety case. The section of related safety cases ensures this. The same section also states that if a component comes with a certificate, this could be used as a related safety case. This means that if a COTS component can be adequately certified, the certificate can become evidence for the components safety, and this certificate can be used in the system safety case as evidence for the entire systems safety.

After this examination of an EN 50129 is would seem difficult to build such a safety case for a COTS component due to the lack of information about developmental processes and internal architecture of the component. The validity of a safety case is, however, dependant on the belief of the assessor, and if the safety of a component could be proven just by examining the product, not the developmental process, and documented in a safety case, this could be valuable for certification of COTS components.

The conclusion of a EN 50129 safety case could possibly be built using the SHIP methodology. By making high level claims and using the first 5 sections of the safety case as evidence, it could be inferred that the system is safe. The arguments lie within the fulfilment of the definition of the system, the quality management report, the safety management report, the technical safety report and the related safety cases. This is provided that the five first sections give enough information.

The SHIP model opens up for using simulated experience (testing result etc.) as evidence in a safety case. For a COTS component this might be the only possible way to prove the safety. Design documents, documentation of the developmental process, quality management and access to requirement specification are rare when you buy a COTS component, and this makes it difficult to follow the EN 50129 standard for safety case. The SHIP model is hence more suited for COTS components, and it could be possible to write a safety case for a COTS component using this model, providing that the component can be properly tested.

It is important to have safety cases for each of the components in a safety critical system. This due to the argumentation that if component A is proven to be safe and component B is proven to be safe, and the connection between the two components is proven to be safe, then the entire system (made up from component A and B and a connection between them) is safe. If you want to compose a system from component A and component B, but only one of them is proven to be safe, it creates a hole in the reasoning about the safety of the system. You cannot know that "2 operator B" = 5, if you are not sure that the operator is + and that B equals 3!

3.5 HAZOP

3.5.1 What is HAZOP?

HAZOP stands for HAZard and OPerability study, and it is a technique for identifying and analysing the hazard of a system. The purpose of HAZOP is to identify the hazards posed by a system so that, following the analysis, appropriate countermeasures may be taken to improve the safety of the system and its environment [2].

HAZOP is a process, and a team performs it. A study initiator, preferably someone other than the persons responsible for the day-to-day development of the safety-critical system, should take the initiative to perform a HAZOP. This person will have the responsibility for monitoring the study and for providing support whenever it is needed for the study to meet its objectives.

A study leader will also have to be appointed for the HAZOP. The study leader will have the organizational responsibility for the study team. The study leader will also be the study initiators contact with the team. The team itself needs to be nonhomogeneous in order to get the widest possible experience base. The team also need

to be well functioning end free from destructive conflicts for obvious reasons.

HAZOP examines the component and the interconnections between components so as to explore whether deviations from design intent are possible, and if so, what their causes and consequences might be [2].

A HAZOP is not carried out on a physical system, but on a representation of a system [2]. This representation could be called a design representation. Design representation could come in a variety of forms: block diagrams, UML diagrams, mathematical notation, etc. It is, however, important that there exists *some* sort of representation and that all the HAZOP team members understand this representation.

The representation of the system is a major part of the HAZOP. A representation portrays the intention of the designer through the features of design. The representation can take many forms and may be more or less detailed. It is often pictorial, but it does not have to be, and other forms have been used successfully [2]. Generally representations use symbols, text or a mixture of the two to show the design intent.

The design representation may be physical or logical. Physical representations show the physical "real world" layout of the system and are useful for determining hazards that might be caused by a system's physical layout [2]. The logical representation shows the logical relationship between the components in the system.

In order to investigate the possible deviations of attributes from their design intent, a set of guidewords is used. These guidewords are a key part of the HAZOP procedure. A guide word is a word or phrase which expresses and defines a specific type of deviation and elicit ideas and discussion, and thus maximize the chance of identifying and studying all possible hazards [2].

The successfulness of HAZOP is dependant of several factors [20]:

- The representation and other data that is the basis for the study needs to be accurate and complete;
- The study groups technical insight and capabilities;
- The study groups ability to concentrate on the most severe dangers identified.

From this list of success factors one can see that HAZOP is heavily dependant on the human resources.

HAZOP has a number of advantages [2, 20]:

- it is team based and brings a variety of viewpoints to the identification of possible hazards;

- it is structured, thus ensuring thorough and consistent coverage;
- it can be used on operational systems as well as on a proposed design;
- it can be carried out at all stages of a system's life;
- it is particularly effective for new systems or novel technologies;
- it is well suited for both smaller and larger systems;
- it can be used to discover possible safety and operational problems due to modification of a system.

HAZOP also has some limitations [20]:

- A danger not foreseen by the study group will not be covered in the analysis;
- It is dependant on the study group composition;
- It does not allow for common causal errors and human errors;
- It is difficult to identify component errors and environmental influence as a cause of deviation.

HAZOP is not primarily a method for certifying components. It is more a method for *identifying* possible dangers with a system, and to suggest possible ways to handle these dangers.

3.5.2 COTS and HAZOP

In order for a HAZOP to be successful, the human interaction bit has to be functioning. Everyone should be competent at his job. The initiator should select a competent study leader and carry out cooperation with him. The team should perform well with each other and have sufficient knowledge about the problem area, preferably representing several different angles of attack. The study itself should be well planned and the guidewords used should be chosen with care. All of this have influence on how well the HAZOP works and how successful the results are.

When one considers whether HAZOP can be used on COTS components or not, the problems with human resources will not have an impact on the answer. These problems are bound to be the same, whether HAZOP is performed for COTS or for an in house component. If the team does not work well, the HAZOP will, in most cases, be less successful.

There are, however, other variables beside human resources to consider for to achieve a successful HAZOP. The representation of the system is important to the process, and if it is not sufficiently good, this might be a problem. Most COTS components do not come with a complete design, and the HAZOP team could have problems

finding a representation, which is sufficiently detailed. If a representation of the component exists at all, it might not cover all the functions of the system. A COTS vendor might want to give a representation of the components main functions, but for a safety-critical system *all* of the components functions should be investigated. Hence, the representation of the system meant for explaining just the main functions of the components, is insufficient for a HAZOP, since other, less distinctive functions of the COTS component could be a danger to the safety.

If there exist a sufficiently good representation of the COTS component, it may be advisable to perform a HAZOP both for the component itself, and for the entire system, with the COTS component integrated. This is because what might be safe for a standalone COTS, may not be safe when the component is integrated into a system. Another reason for performing HAZOP for the entire system is because the environment for the COTS component is fixed, but there are still possibilities for changes during implementation after a HAZOP for the component is performed. The environment for the COTS component is, however, totally fixed when the component is integrated into a system. It is not advisable to perform HAZOP for just the top-level system, because a HAZOP for the COTS component prior to integration might give hints as to the dangers of using the component. If a preliminary HAZOP for the COTS component reveals too many dangers, it could advise against using the component all together, and finding out prior to integration will be less expensive than if the dangers were discovered after integration.

3.6 What is minimum certification?

It is reasonably clear that COTS components cannot be used uncritically in a safety critical system, or in any system for that matter. There has to be some control over what the component does, and does not do, and the developer needs to be able to trust that the component behaves as expected. The problem, however, remains the same, *how* can COTS components be trusted?

When inspecting the quality of a software product, the focus may be on people, process or product [45]. The people focus is meant to evaluate the developers of a product. How skilled are the developers, how relevant are their prior experience for the task at hand? The process focus is mainly concerned with how the product is being developed. Here standards such as IEC 61508 can be of help in assessing the process. The product focus is meant to consider the product itself, without considering how it was developed or who did it. With COTS components the most likely focus when inspecting the quality of the product is to use the product focus. In most cases there will be too little information to inspect or assess the people behind the product or the process used to develop it.

COTS components come at the mercy of the COTS vendors. They control every-

thing about the component. They decide upon a requirement specification, and they decide how much documentation the component should be sold with. This means that even if they do have much documentation about their component, they may choose not to sell it, because they want to keep their business secrets to themselves. COTS vendors are interested in selling to a large market, and safety-critical systems are not their main target. It is therefore unlikely that developers of safety-critical systems can demand extra documentation or extra service of some sort from the vendors. They are not profitable enough for the COTS vendors to receive special treatment. This means that any certification of COTS component most likely will have to be done by the buyer of the component. There are, however, exceptions, and some COTS component could contain a good amount of documentation, and perhaps some sort of certificate as well.

The task of finding usable COTS components will become difficult, if you demand process assessments for the component (see section 3.3.1). Following a predefined developmental processes for safety-critical systems, such as IEC 61508, is more expensive than the internal developmental methods COTS vendors use [1]. Since the COTS vendors main goal is to make money on their product, they wish to spend as little resources as possible developing a good enough product, and sell the product to as many as possible. They will have to maintain a certain quality, since a bad product not will be profitable, and the bad reputation from bad products, will not be desirable.

The level of assurance a component needs is dependant on the level of criticality of the function it should perform. Both [41] and [42] divides functions by criticality. [42] uses the terms minor, moderate or major levels of concern to distinguish functions. (Functions with major level of concern will have a direct affect on patient, operator and/or bystander so that a failure could result in death or serious injury. Minor level of concern is used to describe functions, whose failure would not result in injury to patient, operator or bystander.) [41] uses a classification method that is based on the IEC 1226 standard. This standard proposes four safety categories A, B, C and unclassified [2], where functions in category A are critical.

Both [41] and [42] demands stricter certification for components in the most critical category of classification. For category A [41], demands that the COTS component should be developed under a rigorous Software Quality Assurance Plan. [42] demands that if a function can lead to major level of concern, a special documentation set for off-the-shelf software should be present. This special documentation should contain the developer's design and development methodologies used in the construction of the off-the-shelf software. From this it can be concluded that for

[1]The COTS vendor may use a predefined developmental process, but the focus of the COTS vendor will primarily not be on demonstrating the products fitness for safety-critical applications.

[2]In this report "unclassified" will mean "has no impact on safety"

COTS components to perform safety-critical functions, it is not enough to just test the component. Documentation of the development of the component is needed in these cases, hence, for the most critical cases the component needs to be checked with both the process and the product focus.

Another fact for certifying COTS component for use in safety-critical systems, is that the certificate should be in accordance to the environment the component is intended for, not the environment assumed by the COTS developer [43]. It is therefore important that the system the component is intended for is analysed. Such analysis should contain the writing of a safety case and a HAZOP to find the possible hazards in the system. The safety case should be based by testing done on the component, and the available documentation for the component (see section 3.4.2 about safety cases and COTS).

The testing of a COTS component should be thorough. Black box testing should be performed, and the test values should be based on the HAZOP done for the system. The values should represent both desirable behaviour (values the component should be able to compute), and abnormal behaviour (values out of range, that the component should be able to tolerate).

White box testing is difficult for COTS components, since the code for the component in most cases is not available. It should therefore be possible to certify components without performing white box testing. If you have access to the source code, white box testing should be considered, but it should not be necessary for certifying the component.

Fault injection is a technique well suited for testing COTS components. This method can help unveil problems with the component in areas that were unexpected. Fault injection values should be random. Flipping bits arbitrarily is one possibility. See section 3.3.2.3 for more information on fault injection.

This procedure for certification is similar to the one suggested by Voas in [43]. He claims, "if a component can pass desirable-behaviour testing, abnormal behaviour testing and fault injection, the component deserves to be labelled high-assurance software."

Certification is dependant on the level of criticality the component's function should have, and the environment the component is to be used in. Hence, the minimum certification depends on what the COTS component is supposed to do. If the component is to be used for critical functions (major level of concern or category A functions), the requirements for certification should be no less than if the component were specially made for the task. That is, the development process should be documented, and it should be according to some recognised standard. If the component is not designated for the most critical functions, the requirements for certification is less

strict, and the method described above may be used (HAZOP, safety case, black box functional testing and fault injection). Minimum certification will therefore depend on how the component is integrated into a system in addition to investigation of the components functions.

Testing the COTS component in the environment it is supposed to be used in is important. The certification goal should therefore not be only to certify the COTS component itself, but rather to demonstrate that a failure in the COTS component will not have catastrophic consequences for the safety-critical system [29].

Chapter 4

Finding suitable COTS products

Contents

4.1 Definitions **48**
4.2 What decides which product to choose? **48**
4.2.1 Earlier experience 48
4.2.2 Knowledge of vendor 49
4.2.3 Programming language 50
4.2.4 Open Source Software 51
4.2.5 Documentation 53
4.3 COTS and functionality **54**
4.4 Cost and benefit tradeoffs? **55**
4.5 What should the choice ultimately be based on? **56**

In the search for suitable COTS components to use in safety-critical system, there are many things to consider. Finding the best-suited component is not a trivial matter, since the problem with certifying the component needs to be kept in mind. Chapter 3 discuss the certification process. This chapter will try to describe what to consider when you are searching for COTS components to use in safety-critical systems.

4.1 Definitions

CVS - Concurrent Version System
UML - Unified Modelling Language

4.2 What decides which product to choose?

The COTS market is large. There are many suitable products for most needs. So how can one find the component *best* suited for your specific requirement?

The first step in the selection process would be to formalise the requirements for the component, that is, to write a requirement specification for the COTS component you want to purchase [33]. This specification needs to be wide enough to make it possible to find a match, yet restrictive enough to get a component that will fit your need and solve your problem. This section will elaborate some of the elements one can consider in order to find and choose COTS components for use in safety-critical systems.

4.2.1 Earlier experience

In many cases you may have some prior experience with a product, or with previous versions of the product. One type of component has successfully been used in another application, and the developers know how the component works; they are familiar with the components functions. It is therefore easy to turn to familiarity instead of trying to find a new, perhaps more suited component.

The advantage of using known components is that it is familiar. The component has been tested, and the developer trusts that he knows all the little ”secrets” of the component. The developer is also aware of the limitations and advantages of the component. Using a familiar technology will also save the developer some time getiing used to his tools of development.

Earlier knowledge with a product may also increase the knowledge of the specific COTS vendor (see section 4.2.2). This knowledge is an advantage either if the knowl-

edge is negative or if it is positive. Firs hand negative experience may save you grief, since you most likely will change vendor based on you own experience. Positive impressions will be an extra push to continue to use that specific vendors products.

A disadvantage of using familiar components is that the component is not necessarily the best suited for the task at hand. Just because the component, or a previous version of the component, once was the best alternative, does not mean that someone has not developed another component, which will fit the current requirements better.

Another reason for considering more than just earlier used components is that the component most likely has not been used in the exact same context before. The component might react in a bad way to a new type of environment, compared to the one it had been used successfully in before.

There may be tradeoffs, between the need for searching for new components, and the advantages gained by using an already familiar component. The project funding controls these tradeoffs. The time used in the search for new component, plus the time it takes to get to know a new component is not necessarily less than or equal to the costs saved by using the best suited component. Sometimes money can be saved by finding and learning a new component, and sometimes using an already familiar, but second best component, is satisfactory for the project need.

In any case, the familiarity of the components is no excuse for not testing the component properly in the designated environment.

4.2.2 Knowledge of vendor

In safety-critical systems, the trust in the component is important. Some of this trust will always be linked to the vendor, no matter how well tested the component may be. You need to know that the component works well enough, even though you can never be 100% certain that the component never will fail. It is therefore likely that a component from a known vendor with a good reputation will be preferred to a component from an unknown vendor. It is easier to trust a component made by a "respectable" developer, than a developer you have never heard of. This does not mean that the unknown developer deserves the mistrust, but since you have no information to support your belief in either direction, it would be foolish to blindly trust them.

If several other components from one specific vendor have been used with success in earlier projects, it is easy to turn to the same vendor for more components. The reputation of the vendor can also have an impact. The "word on the street" can be powerful, and it is often easier to by from the vendor with the good reputation. If the vendor has delivered a defect product more than once, it would be wise to take extra care before buying COTS products from this particular vendor.

To have a good relationship with the vendor from which you buy your COTS component is an advantage. This will ensure better service if problems occur after purchasing the product [31], and you are in a position to make demands, since you consider yourself a good customer and you have the vendors goodwill, even if you are not their main source of income. It is, however, important not to trust all the vendors' statements about the product, capabilities and support. Shifting markets, mergers and buyouts, or unforeseen technological developments can convert a vendor's best intentions into broken promises [31].

Another reason for considering the vendor carefully when you want to choose a COTS component to use in a safety-critical system is the time aspect. Many safety-critical systems are meant to be operation for a long time. 15 years operation time is not an unrealistic estimate for a safety-critical system. If you want to use COTS components in such a system, you have to be sure that the vendor is operational and willing to provide support for the product all this time, and if they are not, you have to be willing to take that risk [38]. One way to protect yourself in this case is to make an agreement with the vendor, that if they stop supporting the product you are using, you will gain access to the source code and the rest of the system documentation, and hence support the product yourself [35]. This will, however, leave you with the problem of maintaining code you have not developed yourself.

4.2.3 Programming language

The programming language used in a COTS component could turn out to be important. Many safety-critical systems demand the use of certain programming languages. Restricted sets of C, Modula and Ada are three commonly used programming languages in safety-critical systems. Depending on the use, and the degree of certification necessary for the COTS components, we could demand that the component should be written in a specific programming language. Such a demand would limit the number of usable COTS components.

COTS components often come as black boxes, and the user has little knowledge of what is inside this box, including the programming language used. It might therefore be difficult to find a component written in a specific language, and use it without further information from the COTS vendor. Vendors on the other hand, may not be to happy revealing the secrets they have placed inside the black box. Even if you did know the programming language used in a component, this information would be of little use, since you most likely would not be able to examine the code, checking it for unwanted use of the programming language.

One could argue that the use of COTS is a big step away from the "traditional" development model used for safety-critical systems. Considering such a step, what difference does it make if a specific programming language has been used or not? If

you want to use an, expectedly unsafe component, does the programming language used make the component more unsafe? Could it not be argued that if the COTS component is being treated as an unsafe component, the programming language used has little relevance? Even if the "proper" programming language has been used, this is not a guarantee that the component is safe. The COTS user still has no knowledge of how the component has been developed, nor does he know more about the internal structure of the component, just because he knows the component has been written in C. He will not be able to fix the code or recompile it either.

It could therefore be argued that demanding the use of a specific programming language would only limit the number of available components. The use of a specific programming language in a COTS component is in it self no guarantee for safety. No matter which COTS component is finally chosen, the developer of the safety-critical system needs to take extra care and design the system so that it makes it impossible for the COTS component to be a hazard to the safety of the system.

4.2.4 Open Source Software

The basic idea behind open source is that when programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, and people fix bugs [22].

Open Source Software (OSS) means more than just access to the source code. The distribution terms of open-source software must comply with the following criteria [21]:

- Free redistribution. The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.
- The program must include source code, and must allow distribution in source code as well as compiled form.
- The distribution must allow modifications and derived works, and allow them to be distributed under the same terms as the license of the original software.
- The license must not discriminate against any person or group of persons.
- The license must not restrict anyone from making use of the program in a specific field of endeavour.
- The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

- The rights attached to the program must not depend on the program's being part of a particular software distribution.
- The licence must not place restrictions on other software that is distributed along with the licensed software. For example, the licence must not insist that all other programs distributed on the same medium must be open-source software.

OSS gives the user full access to source code. This gives an excellent opportunity for inspecting the source code, to see exactly what the program does. It also gives the opportunity to test the component using white box testing and intrusive fault injection (see sections 3.3.2.2 and 3.3.2.3).

The documentation of OSS is often limited, since there are many developers, and they all contribute without actually working together. Documentation on how the component works, and a high level architecture is available in most of the OSS cases, but this is not necessarily sufficient for use in for example HAZOP (see section 3.5).

Another possible problem with OSS is the rapid changes in the components. The OSS components are under constant change, because there are many developers working to improve them in one way or another. Some of the official versions of the components contain errors or bugs, and these errors will not be corrected until users describe them, and someone contributes a version where the problem has been fixed. Therefore, you wish to use OSS components in safety-critical systems, it would be wise not to use a young version of a product (typically in the range of versions 1.1 to 2.9), and instead aim for the more mature versions. Be, however, aware that newer versions may contain new, possibly unwanted, functionality.

If one decides to use a OSS component in a safety-critical system it would be wise to have a thorough inspection of all the code, even if the amount of code seems large.

With OSS components the demand on programming language makes more sense than for other COTS components. It is easier to control the content of OSS components, and it might be easier to certify the component if it is written in a certain programming language. On the other hand, the number of actual candidates would be limited by demanding the use of one programming language.

For many OSS components there exist mailing archives. By joining mailing lists users of OSS components can ask developers questions, and report bugs, and the developers can get feedback on their product. It is also possible to give feedback in the form of correcting the code and sending it back to the developer. Most OSS operates with CVS versions of the product, and this CVS version can be downloaded and altered locally. The mailing archives may also be a great source for someone trying to certify a OSS component, since the archive more or less describe the bug-

and operation history of the component.

The greatest advantage of using a OSS component is the access to the source code. If more were using OSS components, this would create pressure on the COTS vendor not willing to give others a peek at their code. If the pressure got bad enough, other vendors may choose to grant buyers access to the code, in order to keep up with the competition. This could prove to be in the best interest of safety critical systems.

4.2.5 Documentation

In order to use a COTS component in a safety-critical system, the component need to have a certain amount of documentation. The intention with the component, and the functions it is supposed to perform, needs to be clearly formulated, and easy to understand. If a HAZOP is to be performed, the system needs an unambiguous representation, and the quality of such a representation could possibly be higher if it was made by the developer of the components. The developer would know all the functions of the component, and therefore be able to make a more complete representation than a person who has not been involved in the development. On the other hand, if there is to be no contact between the developer and the user of the COTS component, it is not certain that the user would fully understand the developer's representation.

It is possible to use formal languages to represent a system. The problem with such standards is that they often are internal to one of a few companies, and hence, cryptic to understand for others. UML is a representation language. This language is not formal, but it should be understood by many. Problems could, however, occur when people start making their own interpretation of the notation. This is a problem avoided in formal languages. It is therefore tradeoffs between understandability and consistency. If many understand the notation, the interpretations become wider, and if there is little room for personal interpretations, the representations will be known to few.

Documentation of a component should be more than just the functional specification and a representation. The limitations of the component, known insufficiencies (for instance if the component requires other specific components in order to function correctly), and the environment for which the component has been tested, needs to be written down for the user to read. Installation and configuration tips are also common in the documentation for COTS components.

The amount of, and the quality of the documentation is important if the component is to be used in a safety-critical system, and hence if two components differ only in the amount and/or quality of documentation, the component with the best documentation would be preferable.

4.3 COTS and functionality

During a search for a COTS component to match a specific need, you may find several possible candidates. Which one you end up choosing depends on more than one variable. Section 4.2 discuss some of the things you may consider. In addition to this, it might be wise to carefully consider the functionality each components delivers.

The likelihood of finding a COTS component that can perform your specific function is high, but the likelihood that such a component contains more functionality than just the bit you require, is equally high. This could turn out to be a problem. First, you need to take extra care during the testing of the component to make sure that none of the components functions act unexpectedly. Second, you have to make sure that the components extra functionality do not perform any functions that could compromise any of the functions you want the safety-critical system you are integrating the COTS component into to do. That is, the extra functionality needs to be kept under control, when the component is integrated into a safety-critical system.

Unexpected behaviour can lead to failures, and it is difficult to prevent these failures. If you are not aware that bad things could happen, you will not take steps to prevent it. For this reason you cannot just ignore the extra functionality in a COTS component. While you are busy looking after the rest of the system, some minor function you have not protected the system from, might perform an action you had not anticipated, or you thought that would not to have an effect on the system. This minor function might, in a worst case scenario, cause the entire system to move to an unsafe state, and hence endanger persons or equipment.

The extra functionality present in all COTS components complicates the problem of using these components in safety-critical systems. Not only do you have to make sure that the function you need is present and functioning correctly in the COTS component, you also have to make sure that none of the other functions will become a problem. The more extra functionality the COTS component contains, the more complex thinking is required to prevent all possible errors. It would therefore be wise to choose components with functionality closest to your need, and with as little superfluous functionality as possible. Chapter 5 discusses ways to protect your system from the COTS components you have integrated into it.

Another problem with the functionality of a COTS component is that the user of the product has no control over the product performance [31]. This could be a problem in safety-critical systems. If a component uses too much time to perform a function, the rest of the system may not have the time to wait, and the operation may time out. It should not be a problem to ensure that the safety-critical system does not enter an unsafe state, by implementing a time out handler, but the system may have other concerns as well. Imagine waiting for the train to get home from work, but the train never shows up, because a COTS component that was not optimised for

Benefits	**Costs**
Shorter time to marked for your application	Extra cost to integration, due to incompatibility or extra functionality.
Utilize other's expertise, where you have non	Difficulties with performance constraints
Reduce developmental costs, writing code is expencive	Licence agreements
COTS is tested by many, bug's reported and fixed. This should make the software more reliable	Vendor may go out of business during development or operation of the system
	Testing and certification of the component
	Searching for the best suited component
	Unexpected contents of the bought component

Table 4.1: The cost and benefits of purchaning COTS components

performance could not compute its value within a given time frame, and the train control system had to stop all trains, to be sure a collision did not occur. If this problem occurred on a regular basis, you would eventually try to find alternative means of transportation.

4.4 Cost and benefit tradeoffs?

Using COTS components shift the cost from development to integration [45]. The use of COTS components will also bring disadvantages such as integration difficulties, performance constraints, and incompatibility among products from different vendors [45]. In addition the cost of testing and certifying the component will be present if the component is to be used in a safety-critical system.

The anticipated benefits of using COTS components and the elements that may cause extra cost in the development are summarised as shown in table 4.1 (the advantages the use of COTS is anticipated to bring, is discussed in section 2.4).

From this table (table 4.1) we see that the cost-benefit analysis for using COTS components is not trivial. There are many variables, and it is difficult to make a simple calculation to find out if the costs are lower than the benefits. In some cases it may even be okay if the costs exceeds the benefits, because you may be willing to pay extra for just one of the benefits.

For instance, let us for the sake of argument say that a developer has knowledge of a certain COTS component with the desired functionality, and he wishes to use this component in his safety-critical system. First, the developer has to make sure that this component is the most suited of the ones available, because it is likely that there are more components than the one the developer is aware of. In order for the developer to use this component in a safety-critical system, he has to make sure that the component is safe. This means that the component needs some sort of certification (see section 3.6). If the component is still desirable after this, *then* it could possibly be used. If, however, he found that the process of finding, certifying and integration the component cost him more than the cost of developing the component himself, he may still want to use it, because he has little knowledge of the specific application area.

It is difficult to compare the cost with the benefits of using COTS. There are no easy comparisons. If you for instance try to compare cost of developing the functionality yourself against the licence costs of buying a COTS component, you will in most cases find that it is cheaper to buy the component. The problem is, however, that it is impossible to compare the costs this way. The variables affecting the cost of the component are not independent of each other, and a function expressing the relationship between these cost variables is complex, and it is also unknown. It is probably impossible to establish an exact function, since there are many hidden costs involved in purchasing COTS components, and some of the cost variables listed in table 4.1 are impossible to quantify. That is, it is difficult to estimate their values. It is, however, advisable to try to make an estimated cost analysis on the variables listed in this section. This analysis will not be definite, since many of the variables are difficult to measure exactly, but the analysis will probably give a hint to if the component is worth buying or not.

4.5 What should the choice ultimately be based on?

This section will try to suggest a strategy for how to choose the best suited COTS components for use in safety-critical system.

The first step towards finding a suitable COTS component is to be certain of what you want. Writing a requirement specification for the wanted component, will be a good way to formalise what you are looking for. The requirements should not be too strict, so that possible solutions will be eliminated at this early stage in the selection process.

The next you need to do is survey the marketplace, and gather information about components. By comparing the information gathered with the requirement specification, it should be possible to narrow the search down to a few (e.g. 3 - 7) possible candidates. These candidates should be compared by prise, licensing conditions, who

developed them (vendor knowledge), experience either with the developing company in general, or with that particular component and its compatibility with the rest of your system. Programming language and documentation may also be considered, and if the component is open source, the maturity of the component should also be checked[1].

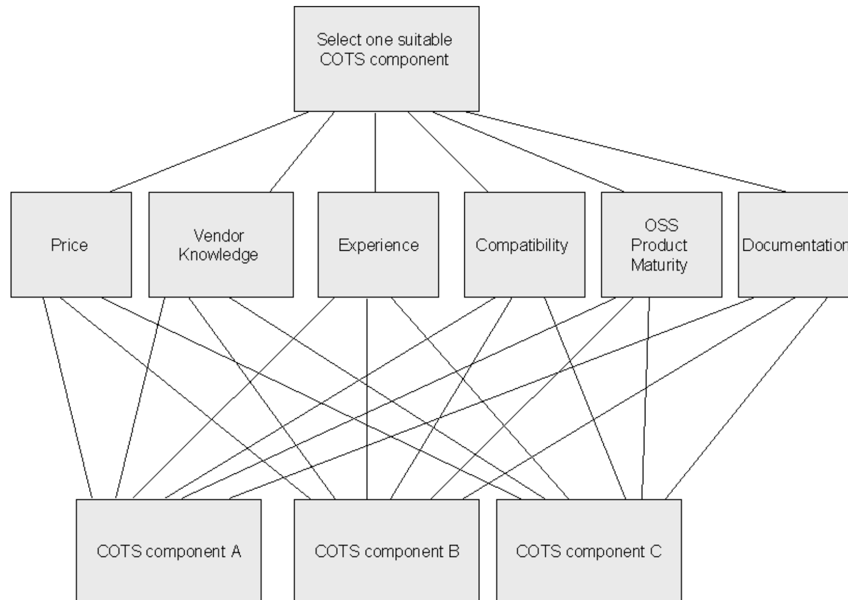


Figure 4.1: The principle of decision making [46]

After the comparison of components, it should be possible to pick one candidate that is better suited for your needs than the other components. The comparison would probably be guided by the selector's experience as well as formalised decision criteria. The principle of deciding upon one COTS component is shown in figure 4.1.

If the component was not to be used in a safety-critical system, this would probably be the end of the decision process. If, however, you want to use the component in a safety-critical system, you need to check the component more thoroughly. The component needs certification and you have to make sure that you can live with all the eventualities the component comes with. A cost-benefit analysis (as described in 4.4) needs to be done, and you need to make sure that the component can be integrated into your system without too much trouble [47]. You should also be aware of the problems that extra functionality may cause later (see section 4.3).

The final choice of whether or not to use the component in a safety-critical system

[1]There might also be other considerations such as usability, security, availability and so on.

should not be based on the criteria described in this chapter alone. The need to be able to trust the component, and to be certain that the integration is not a problem should also weigh heavily. The certification options described in chapter 3, and the integration possibilities described in chapter 5 should therefore also be considered before the final choice of use or not to use the COTS component is made.

Chapter 5

Restricting components through system design

Contents

5.1 Definitions **60**
5.2 Wrapping **60**
5.2.1 Wrapping the component 61
5.2.2 Inverse wrapping 63
5.2.3 Wrapping the system 65
5.3 Physical separation of components **66**
5.4 Redundancy **66**
5.4.1 N-version programming 67
5.4.2 More than one COTS for the same function 68
5.4.3 Temporal redundancy 68
5.5 Information polling **69**

A significant concern surrounding the use of COTS components in safety critical applications is not only whether the *component* is safe, but also whether it can be argued to be safe within the defined structure of the required system safety case [19]. Certifying a *component* is therefore not enough. Errors in the COTS component may exacerbate errors in the rest of the system [38]. This means that the component should be compatible with the rest of the system, and the integration of the component should be done in a manner that will ensure that the system, *as a whole*, is safe enough to be used. That is, you need to restrict the component from doing other things than the function it is supposed to do. This chapter will discuss how COTS components can be restricted through design, when they are integrated into a safety critical system.

5.1 Definitions

CPU -	Central Processing Unit
wrapper -	Software that accompanies resources or other software for the purpose of improving convenience, compatibility, security or safety [39]. A software wrapper is a software encasing that sits around a component and limits what the component can do with respect to its environment [30].

5.2 Wrapping

With wrapping techniques an interface layer is developed around a software component so as to detect errors and prevent their propagation from or to the component [40].

Wrapping is a technique used to ease the difficulties integration of COTS component can provoke in a system. A good wrapper can make a system safer, more secure, and/or more compatible, dependant on the intentions with the wrapper.

The basic principle of a wrapper is shown in figure 5.1. A shell is implemented around a COTS component. This shell may control the input to the component, ignoring certain inputs, and letting others pass. The shell may check the output from the component, making sure that certain constraints are satisfied [30].

Software fault injection (see section 3.3.2.3) is often used to verify wrappers [30, 40]. It is possible to write a table containing the undesirable states of the system. Values arriving to the wrapper will be compared to the table, in order to determine what to stop and what to let through. Artificial intelligence techniques may be used to

build heuristics that can recognise faulty states in the future [30]. These heuristics may be based on the fault injection results.

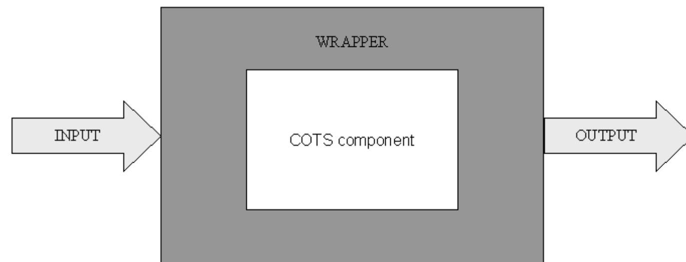


Figure 5.1: The basic principle of wrapping

Wrappers may also be used for altering components functionality, so it will fit some requirement. For instance, when a component requires a specific environment or data in one specific format, a wrapper can be built to convert all in- and outputs to and from the component.

The focus in this section will be on wrappers constructed to increase the safety of a system.

5.2.1 Wrapping the component

The most obvious way to construct a wrapper is to build it around the COTS component, as shown in figure 5.1. This wrapper technique is a code shell surrounding the COTS component, controlling the interface between the component and the rest of the system. This sounds like a simple solution to a serious problem, but when you consider *how* the wrapper should be constructed, you will discover that the solution is not trivial.

What is the wrapper supposed to control? When is the wrapper supposed to take action? You may choose to let the component perform its functions, without interference, and just check if the results dangerous or not.

With this solution you will encounter some of the same problems as when you try to certify a COTS component (see chapter 3) by testing it. You can implement a wrapper that will protect you from faulty values, but you can only handle the values you think of. If the COTS component for some reason produces another, potentially dangerous, value than the once you have thought of, the value may pass the wrapper,

and cause problems in the system.

You would want the wrapper to stop values that the COTS component were not intended to produce, but how do you know exactly what was not intended when you do not have access to the original requirement specification for the component? One solution is to check every value against a list of unwanted values, as described above. Another is to forbid the component from producing any other values than some predefined set. This solution will be further discussed in section 5.2.2.

By building a wrapper around a COTS component, you gain more control over the component. Before deciding to use a COTS component at all, the component needs to be thoroughly inspected (see chapter 3). During this investigation you will learn something about the COTS component. You will learn some of its strengths and weaknesses, and by using a wrapper, you can protect your system from any problematic functionality you discovered during the testing of the component. Wrappers help you to get a smooth integration of components, provided that the wrapper itself does not become large and complex.

The likelihood of finding a COTS component with only the one or two functions you require, and no other functions, is small. COTS components will probably not fit into the system as a specially made code sequence would, and no matter how well you test the component and get it certified, you still have to take some precautions when you wish to use the component in your system. This means that you probably will have to use some sort of wrapping around the component, in order to dispose of problems due to mismatch between the different technologies used. A wrapper may be used to smooth out the differences between the host system and the component itself, and in most cases a wrapper of some sort will be necessary in all systems containing COTS components. The complexity of the wrapper is dependant on the wrapper's intentions.

One problem with wrappers is the size of the extra code. Extra code may for one thing have an impact on the systems response time, and may be a problem in a safety-critical system. If for instance the system is supposed to shut down a function to stop a person from getting hurt, and the system experiences a delay before the shut down, the person might end up getting injured in the meantime. This problem might be helped with a timeout function, but the solution is not ideal. If the system times out often, the system will not be able to performed its functions at all, and this is not preferable.

Much extra code will also have an impact on the cost of using the component. If the COTS component is complex, and you have a large number of possible events to handle in the wrapper, the code needed to make the system safe might be too large. It is even possible for the wrapper to be larger than the COTS component it is isolating [6]. If you end up using more time and effort writing the wrapper

code, than you would implementing the COTS' functions, it would be wise not to use the COTS component in the first place. The limit for such tradeoffs is difficult to generalise, because the problem may vary from system to system. When it is beneficial to use COTS components with wrappers will also vary among systems.

As mentioned above, wrappers may also be used to slightly alter the functionality for a COTS component. If you have chosen to use a COTS component whose functions are close, but not exactly what you are looking for, you may need to alter the results slightly. E.g., if you require an integer number, but you can only find COTS components computing floating points, you may build a wrapper to convert the numbers. Note that this is just an example, and that the wrapper functions may be more complex than this trivial example.

Using wrappers as a form of maintenance could cause instability in the system, since alterations in one part of a system, will most likely have side effects, since there are few parts of a software system that are truly independent [38].

5.2.2 Inverse wrapping

Instead of letting a COTS component perform its function, and then checking the result for bad values, as described in section 5.2.1, it is possible to define a finite set of allowed values, and let *only* values listed here pass. "I will not let ANY other values than this set pass the wrapper". Hence, inverse wrapping and "normal" wrapping are similar methods, the difference is the focus of the wrapper. Normal wrappers focus on stopping some dangerous values, the inverse wrapper should only let through some defined values.

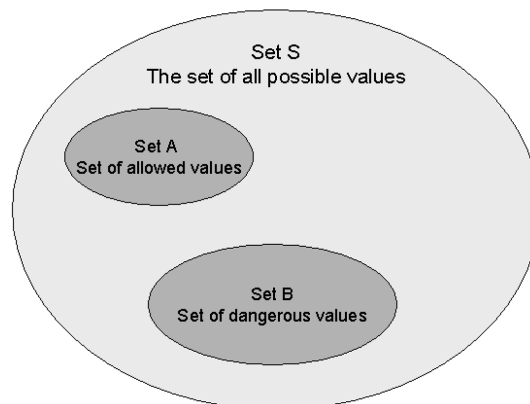


Figure 5.2: The set of values to and from a COTS component in an ideal world

Figure 5.2 illustrates the concept of normal and inverse wrappers. With a normal wrapper all values in the set B will be stopped. The rest of the values are allowed to pass freely. An inverse wrapper will only let values belonging to set A to pass.

A software system is complex. In the ideal world the situation would be as shown in figure 5.2. Here the entire set S of possible values to and from a COTS component is known, and so are the sets A and B. In this ideal situation the sizes of the sets A and B would decide which wrapping technique you should use. If set A is limited inverse wrapping would be preferable, since this is a stricter form of control. If set A were larger, and set B relatively small, normal wrapping would be the best solution. The world, however, is not as simple as this.

Figure 5.3 shows a more realistic picture of the COTS components and the values it may produce. This figure shows the main problem with COTS components, namely that the set S of all possible values does *not* equal the set U of known values. All the things the COTS component is capable of is not known to the user, and the sets of allowed and/or dangerous values may exceed the set U.

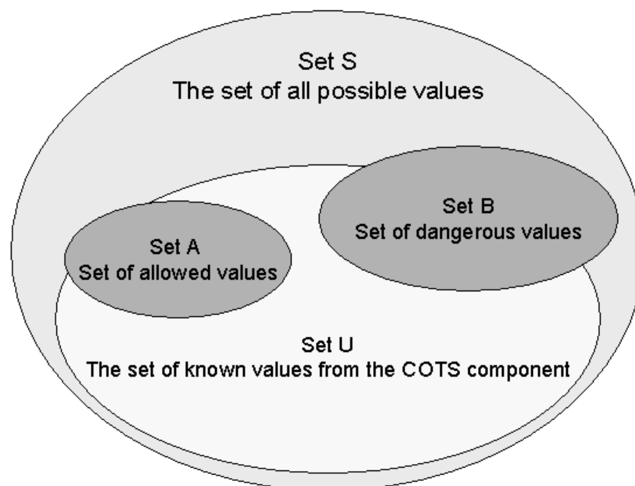


Figure 5.3: The set of known and unknown values to and from a COTS component

The inverse wrapper approach places the COTS component under strict control, and this will increases the safety of the system. The only condition is that absolutely all values that is needed from the COTS component needs to be defined. If the set of possible values is large, it will create a problem. The wrapper code will quickly become big and complex, and this may in turn be a problem for the response time. Again there are tradeoffs between the benefit of using COTS and the cost of using it. If the COTS components solution space is large, the technique of inverse wrapping is not recommended, since the complexity of the wrapper will create more problems

than desired. If, on the other hand, the set of dangerous values is not known either, normal wrapping would not be sufficient to ensure the system safety.

Even if wrapping is a step towards a safer system, it does not have the capability of removing all hazards in a system, but then again, nothing has. It is impossible to make any system foolproof, because there will always be some combination of events that is capable of creating an error in the system. The question for safety-critical system is that such errors should not cause the system to inflict harm on persons or equipment. In a safety-critical system the probability of system failure should be kept at a minimum.

5.2.3 Wrapping the system

Another possible wrapper strategy is to wrap the system, not one single component. This strategy is shown in figure 5.4. Here, the COTS component is not restricted in any way. The wrapper is built around the part of the system containing safety-critical functions.

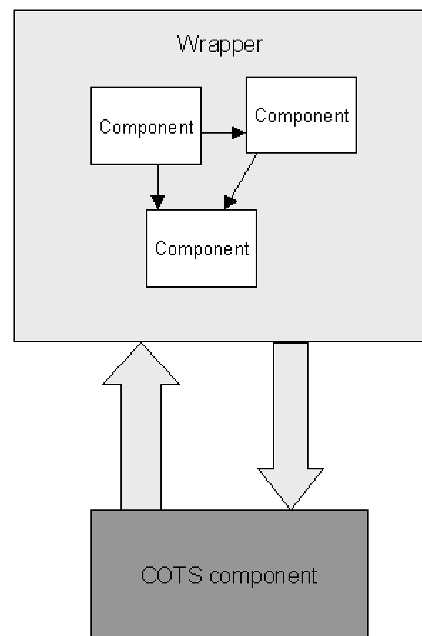


Figure 5.4: Wrapping the system

The advantage of building such a wrapper is that the influence the system can have on the performance of the component is limited. In safety-critical system containing

COTS components, such wrappers will probably have limited range of application. The main concern of these systems is to prevent the COTS component from inflicting errors on the rest of the systems, and leaving the COTS component open like this gives you little control over it. If, however, you suspect that certain events in your system will have a less fortunate effect on the COTS component, it might be wise to consider controlling the environment around the COTS component. Building a wrapper around the system, leaving the COTS component outside the wrapper this way is one way of controlling the environment around the COTS component.

5.3 Physical separation of components

Separating functions physically may increase safety, because the system will become easier to control. The more functions are isolated, the less they can interfere with each other. The unexpected combination of events is the fault that is most difficult to prevent, because it is unexpected, and by isolating functions physically their opportunity to interact with each other, is limited.

It is possible to use one separate CPU for each critical function in a safety-critical system. This will prevent the functions from stealing storage from each other, and in other ways disturbing the execution of functions. This philosophy can be transferred to COTS components. Each COTS component in a safety-critical system could be run on a separate CPU, and in this way physically separate the COTS component from the rest of the system.

In theory this would lighten the burden of controlling the component. On the other hand, separating each function in the system on its own CPU is not a guarantee that components cannot interfere with each other's functions. The components are present in the system for a reason. They are supposed to cooperate to complete a task, and even if they are physically separated, they need a common contact point in order to do their designated operations. They might need access to the same area of memory, or pass data to each other. Hence, they will be forced to interact.

With this interaction among components it becomes important to control the protocol for how functions will be handled in the system. The sharing between the CPU's needs to be clear, and all communication between components and all conflicts in the communication needs to be resolved.

5.4 Redundancy

Majority rules, is a concept much used, in a variety of situations. If four of your six watches tells you that the time is 01.15 pm, one say 01.05 pm and the last one 01.25 pm, you are likely to conclude that the time is in fact 01.15 pm. The argumentation

for believing this is that the likelihood of the majority being wrong is small. Ultimately this does not mean that the majority always is right, [1] but it will in many cases give a good indication that something is wrong somewhere if there are several different answers to a question [2].

The philosophy that many cannot be wrong at the same time, may also be applied in software systems. In this section some redundancy techniques that can be used to increase a systems safety will be discussed.

5.4.1 N-version programming

N-version programming means that a given specification is implemented N times in independent and different ways. The purpose of N-version programming is to detect and mask residual faults during program execution, and continued operation of the system [9]. During execution the same input is presented to all N versions, and a voting mechanism decides which value is correct, and if some value needs to be excluded.

The intention of N-version programming is good, but the technique has some weaknesses. There should be no similarities in the N different implementations of the specification. The people developing the components should not have a common background, i.e., they should not be educated at the same institution. They should not have read the same educational books, and they should not have cooperated with each other in previous projects. In short, they should have the same level of expertise, but totally different background and way of solving problems. This may be difficult to accomplish. Especially if the problem to be solved has limited usage, that is, few have tried to solve it earlier. There are usually more than one solution to a problem, but the choice of solution will be biased by previous known solutions.

N-version programming is intended to make sure that the result of some operation is correct. The reason for using N-version programming is often that the problem is difficult to solve, and that the result is critical, and therefore should be correct. Solving complex problems is difficult, and people tend to make mistakes on difficult tasks. The intention of N-version programming is therefore not necessarily fulfilled for complex tasks. Even if one out of three has managed to implement the function correctly, this is not enough to get the correct solution selected, since there is no way of telling which of the three different answers is the right one.

If, however, the N in "N-version" is large enough, then most of the results would be correct. Again the cost of the method may decide if it should be used or not. Finding

[1] after all, everybody thought Galileo was wrong, when he claimed that the earth orbit the sun, and not the other way around

[2] This is dependant on the nature of the question. It is assumed that the questions mentioned here are questions that will require unambiguous, clear answers, and that they are not of the "what is the meaning of life" kind.

persons for implementing the different solutions would probably be quit expensive, since the developers not should have any common background. In addition, the expense would increase with the number of solutions required. And if it is possible at all to find these people, the gain of using N-version programming may not be big enough if, the task to be solved is complex.

5.4.2 More than one COTS for the same function

To ensure that COTS components do not create dangerous situations by returning a faulty value in a safety-critical system, the philosophy of N-version programming may be applied. If we could find more than two COTS component with the same functionality, it could be possible to get a majority vote on what is correct for some specified function. Figure 5.5 illustrates this principle.

Using this method we will have the same problems as with N-version programming, but in addition you get the problem that you do not know how the components have been developed. You have no way of controlling that the developers of the COTS components are truly independent, and you will not be able to check that the internal logic in the components are different. In addition, it will be difficult to find COTS component providing the desired service, and even if more than one such component could be found, their specifications are generally not identical [9]. This may in turn lead to development of complex and error-prone interfacing and voting components, which may decrease the benefits expected from the approach [9].

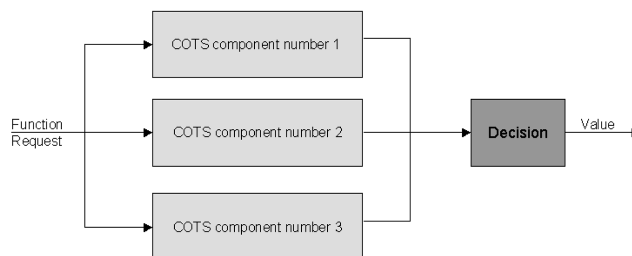


Figure 5.5: Using more than one COTS component for the same function

5.4.3 Temporal redundancy

Temporal redundancy applied to a software component means that the execution of the component is repeated when an error has been detected [7]. The error detection may be performed by the component itself, or by additional software layers, e.g. wrappers (see section 5.2).

In many COTS components such mechanisms are already implemented, and the user of the component will not have the chance of altering the code, or moving the check

for errors to a software layer outside the component. Hence, the user of the COTS component may experience delays from the COTS component if the component detects an error and automatically reruns the code to return a valid result. As mentioned before, time delays may be a problem in safety-critical systems. The problem with time delays will also be present, if the checkpoint to determine if a re-execution is needed is implemented in a software layer around the COTS component. Temporal redundancy may help a system tolerate temporary environment faults that may impact the execution of the software component [7]. Temporal redundancy will in most systems be necessary in one form or another.

5.5 Information polling

Safety in a safety-critical system needs to be considered from the start of the development. This means that is should be taken into considerations during design of the system. Wrappers, physical separation of components and redundancy are examples of efforts made during design in order to increase the system safety. Another possibility is to use information polling.

Information polling means making an inquiry about the values from less reliable sources, such as a COTS component. Figure 5.6 illustrates the principle. A checkpoint is implemented between the safety-critical system and the COTS component. All information between the system and the COTS component need to pass through this checkpoint. The checkpoint administrates sending of the information, and controls the data passed through it.

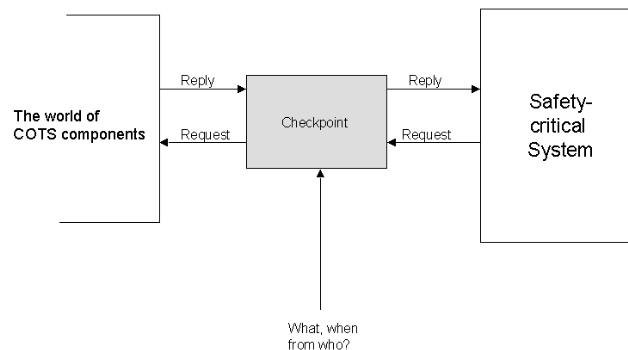


Figure 5.6: Information on request, quality ensured by a checkpoint

With this checkpoint the safety-critical system can use only the values it needs from the COTS component. The system may choose to ignore all values from the COTS component that were not explicitly requested. This will prevent the COTS component from doing things it were not supposed to do, and were not anticipated to do.

To place even stricter restrains on the COTS component, the system may demand certificates for all the communication. When the system makes a request to the COTS component, the system defines which component should do the task, and a time stamp, for when the request was issued. The checkpoint passes the request to the correct COTS component. The COTS component in turn carries out the request, and sends the answer back to the checkpoint. With this response, the COTS component issues a certificate containing its own identity and a time stamp for when the task was executed. If the response passes the test for when and who performed it, the response is sent to the safety-critical system. It is also possible to let the checkpoint test the response, to see if the COTS component has failed in its execution and produced a faulty value.

This method will increase the safety of the system, since the COTS component only will be allowed to produce values when the values are requested. In addition, the bespoke system is certain where the information it receives comes from, and that it is the answer to a request.

[32] suggests a method similar to information polling. In this article a firewall is implemented between the bespoke system and the COTS component. In this model the COTS components should only be used for non-critical purposes. With information polling it is not recommended that COTS component should perform safety-critical functions either.

Chapter 6

Discussion

Contents

6.1 Certifying COTS 72
6.2 Suitable COTS components 74
6.3 Restricting COTS components 76
6.4 Cost/benefit 78
6.5 Futher work 79

The safety and correctness of a system are in conflict. Safety is a system property, while correctness is a software property [48]. This also means that the properties are not mutually exclusive. A component may be correct, but still not safe, because some of the components correct functions may cause the system to fail. The component may also be incorrect, but still safe, because nothing the component does can cause the system to fail. For safety-critical systems it is therefore more important to prove the safety than the correctness of components.

This chapter will summarise some of the main arguments for if, why and how COTS components could be used in safety-critical systems.

6.1 Certifying COTS

The main problems with COTS components are their focus during development, the lack of documentation following the component at purchase, and the lack of allowed insight into the components interior logic, structure and program code.

COTS components are not built for "easy" certification [47]. The COTS vendors main focus during development will most likely not be on safety. Their drive is to make money on the COTS product, and they are making a product to please the masses. Safety is not the property most software developers place in focus. Performance and usability are more important properties in the general case software application.

Standards, such as IEC 61508, which are used to keep safety-critical software under control, and to make the software easy to verify and validate, are not used by COTS vendors. Following such strict standards when it is not necessary will only inflict the project with extra costs. COTS vendors wish to sell their product to as many as possible, since the sales are their source of income. They are therefore not interested in using expensive standards during development, and hence, the products from COTS vendors becomes more difficult to certify than bespoke software [47].

The content and internal structure of a COTS component is usually not known to the user of the component. COTS vendors need to keep their business secrets to them selves, and they are not interested in sharing this information with the COTS buyers. They document the functionality they believe is most important for the COTS buyer to know, and they protect the internal logic, structure and program code by refusing the buyer access to this information. Most of the COTS buyers are not interested in extensive information about the components either, because it is not critical for them if the component fails, or causes their systems to fail once in awhile. They only need information on what the component does, and how to install the component and get it to work properly in cooperation with other components.

Certification of safety-critical systems is usually based on the development process, the documentation of the system, and the tests performed on the system. For COTS components the development process cannot be used in the certification process, and the documentation is often not thorough enough for certification purposes. All arguments for why COTS components are safe enough for use in safety-critical systems therefore needs to be based on post development arguments.

A certificate for a COTS component should begin with a HAZOP for the system. Section 3.5 describes what a HAZOP is, and how a HAZOP could be constructed for a COTS component. Voas suggest in [48] that the HAZOP should begin at system level, and then descend to component level. The purpose of a HAZOP for the COTS components in the system should be to define what the components should *not* be allowed to do. If you can define the space of not allowed actions, you can protect your system from them. The problem is, however, that the set of not allowed actions could be large, and it is not certain that you are able to identify all the dangerous values. It is usually easier to define what a component is allowed to do, than what it is not allowed to do [48]. The set of allowed events is finite, while the set of not allowed events may be large and not well defined.

When a HAZOP for the COTS component has been developed, a safety case should be written for the components. The safety case should be based on the results from the HAZOP. Section 3.4 discuss how a safety case can be constructed for a COTS component. A good safety case could be what is needed for certifying a COTS component, but the quality of the safety case will still be a matter of opinion. It is up to the assessor of the system if he *believes* that the safety case sufficiently proves the components safety.

The information of the safety case for a COTS component may be based on a number of techniques. The techniques are described in section 3.3. Process assessment, such as IEC 61508 has limited application value for COTS components due to the lack of documented development process.

Third party assessment would be nice, but is still not widespread enough to be a real value for certifying COTS components. In addition the need to consider a component's environment makes third party assessment difficult.

Proven in use is a strong source for certification if all the requirements described in 3.3.4 are satisfied. The problem is that successful use of component, rarely are documented, and it is therefore almost impossible to find enough proof that the requirements for proven in use are fulfilled.

Certification based on system properties is also a possibility. The philosophy behind this technique is described in 3.3.6. The problem with this technique is to find properties that are generic enough to make components comparable, and to find

properties that will define the component good enough for the COTS buyer to know what he purchases.

Test methods are important for the certification of COTS components. Section 3.3.2 describes black box testing, white box testing and fault injection. White box testing would be difficult for most COTS components due to the lack of access to the program code.

This leaves us with the possibilities of black box testing and fault injection for certifying COTS components. Black box testing is suitable for testing the components desirable value behaviour. That is, how the component reacts to values it is supposed to handle. Black box testing could also be used to find out how the component will react to out of range values. Black box testing is discussed in section 3.3.2.1. Fault injection is useful for flushing out faults you had not thought of. Fault injection is discussed in 3.3.2.3. These two methods combined should give a picture of how the component functions.

The problem with test methods is that there is no way of testing all possible values. The value space can be large, and it is therefore difficult to find all values you should test. The unexpected combination of events is the largest source of errors in a system, and it will require a lot of experience, finding all dangerous combinations of events in a component. You can therefore never be 100 % certain that even a well tested COTS component will not eventually lead to errors in your system. You cannot predict the probability for some events either, because you do not know all the events that might occur.

Another problem with using testing techniques such as black box testing and fault injection for COTS components is that you will not be able to correct any errors discovered in the components, unless you have access to the source code. If errors are found during testing of COTS components, and you are not permitted to alter the source code for the component, you would have to restrain the component through the design of your safety-critical system by using for example wrappers.

6.2 Suitable COTS components

In order to use COTS component, suitable components need to be found. Section 4.5 describes a method for finding and choosing suitable COTS components.

The most important selection criteria for finding COTS components to use in safety-critical systems are probably the components functionality and the cost of using the component. The cost of using COTS components in safety-critical systems will be discussed in section 6.4.

The components functionality is important. You need to be able to control the component, and because of this, the component should not contain any extra functionality. It would be impossible to demand no extra functionality in a COTS component, and you should therefore use components with as little extra functionality as possible. You also need to make sure that you know exactly what the functions of the component are. If you do not know how something is supposed to behave, you cannot label anything it does as defective [48]. Another problem with not knowing what the component is capable of is that you will not be able to defend yourself against it. What you do not know, may very well be the thing that ends up hurting you.

If the COTS component you end up choosing contains extra functionality, you need to handle this somehow. You need to make sure that none of the functions present in the component may create errors in your system. This may be achieved through constraining the component. Restriction methods for systems containing COTS components will be discussed in section 6.3.

Another issue when selecting COTS components is that when a choice has been made, the component version should be frozen. A certification for a COTS component version 3.4 will have no authority for version 3.5. Newer version of a component may contain new, possibly undesirable functions, and old, wanted functions may have been removed. Changing a COTS component from one version to another may therefore be considered changing the COTS component altogether, and the safety case for the component should be rewritten, or at least updated.

Open Source Software (OSS, see section 4.2.4) is an alternative to "regular" COTS. With "regular" COTS components the buyer would not be granted access to source code, and there would probably be some sort of licence agreement attatched to the COTS component. OSS is free of charge, and the user would get full access to the source code. This opens up for white box testing of the component, and it would be possible to examine the code thoroughly.

For many OSS products, there are mailing archives and mailing lists available, so that users and developers may exchange experiences about the product. These mailing archives will contain most of the product's bug- and operation history, and would be useful for someone trying to determine the components suitability in a safety-critical system. You could argue that with "regular" COTS components you would get support from the developer whenever this is needed, but this is could also be said about OSS components. Even though the support is less formal, many of the people behind a OSS products are engaged in what they do, and they will in most cases be more than happy to answer questions about the product they have developed.

OSS does not forbid change of the product. This means that if you test the compo-

nent and discover something that would be a problem for you in your safety-critical system, you are free to change the code, so that it will suit your needs better. This is impossible for licensed COTS without code access. Problems discovered during testing of these components will have to be handled differently, possibly through wrapping or forbidding the use of the component.

OSS will most likely be distributed with minimal, and sometimes outdated documentation. It might therefore be difficult to understand the logic behind the component. On the other hand, ”regular” COTS components may also be distributed with minimal documentation, and in these cases you would not be granted access to the source code either, hence making it even more difficult to find out what the component does and does not do.

OSS is not developed with safety-critical systems in mind either. Many have contributed in the development of the product, and this might cause a discontinuity in the logic of the component. Therefore you need to be certain of what the component does, if you wish to use such components in safety-critical systems. Again the problem of guarding against an enemy you cannot see will become a problem, but with OSS, you will have the opportunity of examining the code of the component, and hence examine the component closely. If you do not like what you see, you may decide against using the component, and in that case you will only have lost some time checking the component. If you after inspection decide against using a ”regular” COTS component, you will also have lost the money you paid for the licence privileges.

6.3 Restricting COTS components

A component certified for use in a safety-critical system is not equal to a bespoke piece of software certified for use in a safety-critical system. Even if the COTS component has been certified to some extent, it should not be trusted in the same way as bespoke pieces of software. You may have tested the COTS component extensively, but you should still treat it as a black box with unknown content, and hence restrict the component in your system. Chapter 5 describes some ways of restricting COTS components though design in safety-critical systems.

Before finding and testing COTS components for use in safety-critical systems, you should analyse your system, using a HAZOP, to discover the critical points of your application. COTS components should not be left as the last defence at these critical points. No matter how well you inspect, test and check your COTS components, they will probably never reach the same level of safety as components developed with safety in mind from step one. None of the testing methods discussed in this report are able to present the proof you would need in order to trust the COTS components with the most critical tasks in a system. If COTS component should be allowed to

perform these critical tasks, it should be provable that the COTS components were developed under the same strict control as a bespoke component developed under the control of some recognised standard would have been. Provable means that there should exist written material of how the COTS component was developed, and how it was tested.

If a COTS component should be integrated into a safety-critical system, it should only be allowed to perform functions not listed as critical, but even in this situation, the developer needs to keep a close look at how the component is integrated into the system. He also needs to perform tests on the components in their intended environment. This is important, because components may act differently in different environments. The accident with Ariane 5 is a good example of this[1]. A component functioning fine in the Ariane 4 environment was reused in Ariane 5, without further testing. This lead to an error causing the rocket to explode, sending millions of dollars up in smoke.

Wrapping, discussed in section 5.2, will probably have to be used in one way or another if a COTS component is to be integrated into a safety-critical system. By isolating components as much as possible, you will gain a bit more control over them. You will then be in a position to control what the system is doing, by monitoring the flow of information in the system. Wrapping may also be used as a smoothing technique, making any component fit better into the system. In order to increase the system safety, however, you would need to take some precautions during system design. Some of the techniques described in section 5.2 are better than others to use for this purpose.

Inverse wrapping lays heavy constraints on the COTS component. Only the allowed values will be able to pass the wrapper, leaving the system safe from dangerous values. This will, however, demand a clear definition of what is allowed from the COTS component. If the set of allowed values is too large, the wrapper will become too complex, too slow and too expensive to develop.

The best restriction of COTS components described in this report, is information polling. This method ensures an isolation of the COTS component, ensuring that the component will not do anything other than what it is asked to do. It is also possible to combine some of the restriction methods. By including a list of all possible value in the checkpoint in the information polling model, this model might also implement the inverse wrapping principle.

[1]Ariane 5 veered off flight path and exploded in an altitude of 3700 m, 30 seconds after lift off. The failure was due to a error in the data conversion of a 64-bit floating point to a 16-bit signed integer value. The value that should have been converted was too large to be represented by 16 bits. An operand failure occured in the component, and this led to the rocket explosion [37].

6.4 Cost/benefit

Using COTS components in safety-critical systems is problematic, and should therefore not be done, unless some advantages could be demonstrated. The main reasons for using COTS components could be summarized in this list:

- Saved time on development, shorter time to marked
- Saved cost on development, cheaper to buy than to implement
- Utilize functionality, others are experts, and software systems typically demands more and more functionality

For safety-critical systems the purchase prise is not dominant. People or companies are willing to pay a great deal of money for a safety system. This raises the question of whether or not it is cost-effective to use COTS in such applications [47]. Are the COTS components more trouble than they are worth?

In order to use a COTS component in a safety-critical system you need to find the component, certify it through performing a HAZOP, writing a safety case and performing tests for the component and integrate the component safely into your system. All of these activities will cost time and money. Bespoke software for safety-critical systems are expensive due to the strict safety requirements. Following software safety standards, such as IEC 61508, is expensive, and hence, COTS components will be much cheaper in the construction phase of the development, than bespoke solutions. COTS component will, however, be more difficult to certify, and they need to be integrated with care. This makes the certification and integration procedures more expensive for COTS than for bespoke software. It is therefore not certain that the money saved by using COTS components should be the main motivation for using such components.

The aspect of shorter time to marked will probably be a much stronger drive for using COTS components in safety-critical systems than the money saving argument. Since software systems tend to demand more and more functionality, it would probably be bothersome to implement all the functions from scratch for each application, and the time perspective would become hopeless. If the system becomes large enough, it would be hopeless to implement all the code yourself. You would eventually have to buy some of the components in order to be able to finish your product within a reasonable time.

At some point it will probably be more trouble finding, certifying and integrating COTS components than it is worth. Where this exact point is, is difficult to estimate. It makes no sense to use a COTS component if there are no benefits in doing it, because no matter how well you test the component, a bespoke software component would probably be more safe in its place. The actual need for the COTS component,

and the possible benefits of using it should be carefully considered before it is decided that it should be used.

6.5 Futher work

Voas states in [49] that you should not oversell ideas about improving system safety, before there are actual evidence at hand. This evidence should also fairly consider both cost and limitation of obtaining new technology. This statement is also true for this report. There are some assumptions made, which are not funded on actual evidence. As further work to this diploma work the following suggestion of improving the research and providing evidence of the assumptions are made.

This report suggests a HAZOP with a subsequent production of a safety case in order to certify a COTS component. This is just a suggestion, and it is not explained what is actually necessary in order to produce such HAZOPs and safety cases. The report does not provide evidence that it actually is possible to produce reliable HAZOPs and safety cases. In order to investigate this, it is suggested that a field study over some common COTS components should be done. The goal of such a study would be to find out what information COTS components come with. Is it possible to produce a sensible evidence for the components safety from this information? It is also suggested that actual COTS components should be tested using black box testing and fault injection.

There should be developed some method for calculating the benefits of using COTS. This method should compare the costs of using COTS to the advantages gained. The function could be based on observations of developments or on estimates of anticipated benefits and costs.

There are presumably advantages to using Open Source Software instead of ”regular” COTS, in safety-critical systems. These assumptions are not based on actual evidence. The relationship between these component types should therefore be investigated further. Experiments to collect empirical data would probably be appropriate to compare the two component types.

It is in this report assumed that some of the restriction techniques described in chapter 5 are more efficient in safety-critical systems than others. This is also just an assumption without base in evidence. Experiments should be conducted to investigate the claims about restriction methods in safety-critical systems.

In section 3.3.6 the subject of certifying a list of characteristics in a component instead of certifying the component itself is discussed. This subject needs further analysis, and it is suggested a study to find out if it is possible, and to find out what could be gained by such an approach.

Chapter 7

Conclusion

The measurement of safety in a software system is never absolute. It is impossible to be 100% certain that the software system is safe. This is the case whether COTS components are present in the system or not. The question of system safety will ultimately be a question of faith. Do you believe that the system is safe? Can the system assessor *believe* in the evidence given for the system safety?

The greatest source of concern in a safety-critical system is not what you know about the individual components. The things you do not know about the component is more likely to create dangerous situations, since unanticipated chains of events may cause system failure.

In order to use COTS components in safety-critical systems, a HAZOP should be performed for the system. The main aim of this HAZOP should be to identify what the COTS component is *not* allowed to do.

Unless it can be proven that the COTS component has been developed with the same strict developmental control as the bespoke part of the safety-critical system, COTS components cannot perform critical functions in a safety-critical system.

A COTS component cannot be used in a safety-critical system, unless some proof is given that the component is safe. Such proof could be a safety case for the component. The safety case could be based on HAZOP, black box testing and fault injection performed on the component.

The main problems with COTS component for use in safety-critical systems are the lack of documentation of the component's development and structure, and the lack of insight into the components logic and programming code. COTS vendors need to keep the desired information secret for business purposes, and the components need to be treated as black boxes. This creates a discontinuity in the reasoning about the system safety as a whole, and the lack of information about the COTS compo-

nents make them more difficult and more expensive to certify than bespoke software.

It may be easier to certify and use Open Source Software (OSS) safely, than ”regular” COTS components. OSS will grant access to source code, hence making it easier to control what the component does, and making it easier to test the component thoroughly. The mailing archive of OSS could be a source for a bug- and operation history for the component.

The safety-critical system needs to be protected from the COTS component through restrictive system design. Inverse wrapping and information polling gives the strictest protection from the component. The restriction through design can protect the safety-critical system from extra functionality in the COTS component.

COTS components should not be used in safety-critical systems unless some benefits of doing this can be proven. These benefits are not necessarily measurable in money. Others expertise, and time saved by using prior made software may also be benefits from using COTS components. In some cases the use of COTS components in safety-critical systems will be more trouble than it is worth.

References

[1] Andersen and Redmill, *Safety-critical Systems.* Great Britain, Cornwall: Chapman & Hall, 1993

[2] Redmill, Chudleigh and Catmur, *System Safety: HAZOP and Software HAZOP.* England: John Wiley & Sons Ltd, 1999

[3] Jones, Bloomfield, Froome and Bishop, *Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP).* United Kingdom: Adelard for the Health and Safety Executive, 2001

[4] Morisio and Torchiano, *Definition and classification of COTS: a proposal.* International Conference on COTS-Based Software Systems, 2001.

[5] Scott, Preckshot and Gallagher, *Using Commercial Off-The-Shelf (COTS) Software in High-Consequence Safety Systems.* Fission Energy and Systems Safety Program (FESSP), Lawrence Livemore National Library, nov 10. 1995

[6] Lindsay and Smith, *Safety assurance of Commercial Off-The-Shelf Software.* Software Verification Research Centre, School of information technology, university of Queensland, Australia, May 2000.

[7] Fabre, Kanoun and Laprie, *Study note 21, Software Fault Tolerance Techniques.* Study of GNSS-2/ Galileo System Software Certification, ESTEC Contract Number 14063/99/NL/DS

[8] Jones, Bloomfield, Froome and Bishop, *Methods for assessing the safety integrity of safety-related software of uncertain pedigree (SOUP).*United Kingdom: Adelard, 2001.

[9] Phillip Robert, *Study note 12, Analysis of current use of standards and norms for safety(issue 1.1).* Study of GNSS-2/ Galileo System Software Certification, ESTEC Contract Number 14063/99/NL/DS

[10] Scott and Lawrence, *Testing existing software for safety-related applications.* Lawrence Livemore National Laboratory, Report UCRL-ID-117224, Revision 7.1

[11] Jeffery Voas, Developing a Usage-Based Software Certification Process. *IEEE computer*, vol. 33, No 8, august 2000.

[12] Stålhane, Herard, Söderberg, Malm, Kylmälä, Pöyhönen, *Safety Assessment of systems containing COTS Software.* april 2000 .

[13] *Black box testing*[online]. Webopedia. Available from: `http://www.webopedia.com/TERM/B/Black_Box_Testing.html` [Accessed May 2. 2002]

[14] *White box testing*[online]. Webopedia. Available from: `http://www.webopedia.com/TERM/W/White_Box_Testing.html` [Accessed May 2. 2002]

[15] Ian White, Defence Evaluation and Research Agency, *Wrapping the COTS dilemma.* Research & Technology Organisation, RTO-MP-048 meeting held in Brussels, Belgium April 2000. Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS", [online], available from: `http://www.rta.nato.int/Rdp.asp?RDP=RTO-MP-048` [accessed April 25. 2002]

[16] Matthew Hardy, *COTS Components in Software Development.* [online] University of Minnesota, USA, 2000. Available from: `http://mrs.umn.edu/~lopezdr/seminar/spring2000/hardy.pdf` [Accessed April 10. 2002]

[17] Voas, McGraw, Kassab, Voas, Fault-injection: A Crystal Ball for Software Quality. *IEEE Computer*, June 1997, Volume 30, Number 6, pp 29-36

[18] Jefferey Voas, *Reducing Uncertainty About Software Safety (3 hour tutorial).* [online] Available from: `http://www.cigital.com/presentations/zurich98/`. [Accessed May 7. 2002]

[19] Dawkins, Kelly, *Supporting the use of COTS in safety critical applications*. Cots and Safety Critical Systems (Digest No 1997/013), IEE Colloquium on, 1996

[20] Marvin Rausand, *Risiko Analyse, veiledning til NS 5814*, Trondheim: Tapir Forlag, SINTEF sikkerhet og pålitelighet, 1991

[21] Opensource.org, *The Open Source Definition*. [online] Available from: `http://www.opensource.org/docs/definition.html` version 1.9, 2001 [Accessed May 21. 2002]

[22] Opensource.org. [online] Available from: `http://www.opensource.org/`, 2001 [Accessed May 21. 2002]

[23] Bishop, Bloomfield, *A Methodology for Safety Case Development*. [online]. Adelard, Safety-critical Systems Symposium, Birmingham, UK, Feb 1998. Available from: `http://www.adelard.co.uk/resources/papers/pdf/sss98web.pdf`. [Accessed May 23. 2002]

[24] Bishop, Bloomfield, *The SHIP Safety Case Approach*. [online]. Adelard, SafeComp 95, Proc. 14th IFAC Conf. on Computer Safety, Reliability and Security (ed. G. Rabe), Belgirate, Italy, 11-13 October 1995 Available from `http://www.adelard.co.uk/resources/papers/pdf/scomp95.pdf` [Accessed May 23. 2002].

[25] Odd Nordland, *Veiledning for "Safety Case" i henhold til CENELEC EN 50129*. SINTEF Tele og data, Systemutvikling og telematikk.

[26] Odd Nordland, *Undertaking a safety case in a rail environment*. [online]. SINTEF Telecom and Informatics, SignalComm Europe 2000, Birmingham, UK. Avialable from `http://www.informatics.sintef.no/~nordland/tekster/Undertaking_a_safety_case_in_a_rail_environment.html` [Accessed May 24. 2002]

[27] Jeffrey Voas, Can Chaotic Methods Improve Software Quality Predictions? *IEEE Software*, Volume 17: issue: 5, Sept.-Oct. 2000

[28] Voas, Ghosh, *Software Fault Injection for Survivability* DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings, Volume: 2, 1999.

[29] Jeffery Voas, COTS software: the economical choice?. *IEE Software*, volume 15, issue 2, march-april 1998.

[30] Voas, Payne, *COTS software failures: can anything be done?* Application-Specific Software Engineering Technology, 1998. ASSET-98 Proceedings. 1998 IEEE Workshop on, 1998.

[31] Boehm, Abts, COTS integration: plug and pray?. *IEEE Computer*, volume 32, issue 1, jan. 1999

[32] Victoria Stavridou, *COTS, integration and critical systems.* Cots and Safety Critical Systems (Digest No. 1997/013), IEE Colloquium on, 1996

[33] Dean, Vidger, *COTS Software Evaluation Techniques* Research & Technology Organisation, RTO-MP-048 meeting held in Brussels, Belgium April 2000. Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS", [online], available from: `http://www.rta.nato.int/Rdp.asp?RDP=RTO-MP-048` [accessed April 25. 2002]

[34] Norman F. Schneidewind, *The ruthless pursuit of the truth about COTS* Research & Technology Organisation, RTO-MP-048 meeting held in Brussels, Belgium April 2000. Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS", [online], available from: `http://www.rta.nato.int/Rdp.asp?RDP=RTO-MP-048` [accessed April 25. 2002]

[35] Nancy Talbert, The Cost of COTS. *IEEE Computer*, volume 31, issue 6, June 1998

[36] Jeffery Voas, *Confidently integrating COTS software under worst case assumption.*Research & Technology Organisation, RTO-MP-048 meeting held in Brussels, Belgium April 2000. Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS", [online], available from: `http://www.rta.nato.int/Rdp.asp?RDP=RTO-MP-048` [accessed April 25. 2002]

[37] J. L. Lions, *ARIANE 5, Flight 501 Failure Report by the Inquiry Board.* [online]. Available from `http://java.sun.com/people/jag/Ariane5.html` [Accessed June 3rd 2002].

[38] Jeffery Voas, COTS software: the economical choice?. *IEE Software*, volume 15, issue 2, march-april 1998.

[39] *Wrapper*[online]. Webopedia. Available from: `http://www.webopedia.com/TERM/w/wrapper.html` [Accessed June 4th 2002]

[40] Fabre, Kanoun and Laprie, *Study note 24, Verifiability of Fault-Tolerance Requirements (issue 1).* Study of GNSS-2/ Galileo System Software Certification, ESTEC Contract Number 14063/99/NL/DS. Aug. 18th 2000

[41] Scott and Preckshot *A Proposed Acceptance Process for Commercial Off-the-Shelf (COTS) Software in Reactor Applications.* Fission Energy and Systems Safety Program (FESSP), Lawrence Livemore National Library, Sept 22nd. 1995

[42] U.S Department Of Health And Human Services, Food and Drug Administration Centre for Device and Radiological Health, Office of Device Evaluation *Guidance for Industry, FDA Reviewers and Compliance on Off-The-Shelf Software Use in Medical Devices.* Sept. 9th, 1999 [online], available from `http://www.fda.gov/cdrh/ode/guidance/585.pdf` [Accessed May 20th 2002].

[43] Jeffery Voas, Certifying software for high-assurance environments. *IEEE Software*, Volume 16, issue 4, july-aug 1999

[44] Jeffery Voas, Certification: Reducing the Hidden Costs of Poor Quality. *IEEE Software*, volume 16, issue 4, july-aug 1999

[45] Sedigh-Ali, Ghafoor, Paul, Software engineering metrics for COTS-based systems. *IEEE Computer*, volume 34, issue 5, May 2001

[46] Ncube, Dean, *The limitations of current decision-making techniques in the procurement of COTS software components.* [online] Available from: `http://link.springer.de/link/service/series/0558/papers/2255/22550176.pdf` [Accessed May 20th 2002].

[47] John McDermid, *COTS: The Expensive Solution?*, IEE Colloquium on, 1996.

[48] Jeffery Voas, Protecting against what? The achilles heel of information assurance. *IEEE Software*, volume 16, issue 1, jan-feb 1999.

[49] Jeffery Voas, Software quality's eight greatest myths. *IEEE Software*, Volume: 16, Issue: 5, Sept-Oct 1999