Introduction to Safety Critical Systems

R. Pietrantuono[1], S. Russo[1,2]

[1] Dipartimento di Informatica e Sistemistica (DIS)
Università di Napoli Federico II, via Claudio 21, 80125, Naples, Italy

[2] Consorzio Interuniversitario Nazionale per l'Informatica (CINI),
Complesso Universitario Monte Sant'Angelo, Via Cinthia, 80126, Naples, Italy

{roberto.pietrantuono, stefano.russo}@unina.it

Abstract. Today's software-intensive Safety-critical Systems (SCSs) are required to cover a wide range of functionalities, to do it in a safe way, and to be developed under stringent time and cost constraints. That is the challenge which the Critical Step project dealt with. In the following, an overview of the main concepts, challenges, and currently implemented solutions in SCSs development is presented.

Keywords: Certification, RAMS, Verification and Validation

1 Introduction

A system is referred to as safety-critical when the consequences of its failure can lead to loss of life, or to significant property or environmental damage. Safety-critical Systems (SCSs) are developed in many domains, ranging from transportation (e.g., avionics, railway, automotive) to space and telecommunication systems, from civil and military infrastructure (e.g., nuclear and power plants) to medical and control devices. Depending on the domain, SCSs are developed following guidelines provided by certification standard, whose typical aim is to give recommendations to developers regarding all the development process activities.

Software in such systems has by now a prevalent role. Systems are required to accomplish more and more tasks, and thus software becomes considerably large and complex to satisfy these requirements. Moreover, even though software is only a part of the entire system, its impact on overall safety has an increasingly significant weight. The numerous reported accidents due to software falls [1] suggest that its reliability is one of the weakest links of system reliability [2]. As a consequence, cost related to software development and assessment activities is among the *highest and least controllable* ones of the entire system development cycle. Thus, researchers and practitioners in this field are more and more convinced that *software is the problem.*

Although software in SCS is developed by using the most consolidated practices in software engineering, no methodology, technique, or strategy is currently able to assure the absence of software failures. More worryingly, it remains extremely

hard, and expensive, to obtain precise and reliable measures of the quality of a software product from the safety point of view. Most of difficulties depend on the intrinsic characteristics of software as compared to other physical systems, such as its “non-linearity” and discontinuity. Its unique features make it difficult to develop *effective* safety assessment methodologies in analogy with other fields of engineering. This produced, in the last decades, a proliferation of techniques to tackle software assessment issues. However, results are still far from being as satisfactory as for system or mechanical engineering.
Since in a SCS development process software development is strictly intertwined with system development, many of the used techniques have been derived directly from system-level techniques; but tailoring them to software is not so immediate and has not always produced the expected results. As a matter of fact, implementing such techniques for software often requires very costly solutions to achieve adequate performance.

Techniques in this area typically address a set of quality attributes commonly used also in other engineering fields, and referred to as RAMS (Reliability, Availability, Maintainability, Safety). The way to achieve acceptable RAMS levels for software, and then to assess the product quality with respect to them, has generated software-tailored techniques acting in every phase of the development lifecycle. Examples are: SFMECA (Software Failure Modes, Effect, and Criticality Analysis), SFTA (Software Fault Tree Analysis), ETA (Event Tree Analysis), SCCFA (Software Common Cause and Failure Analysis), HSIA (Hardware-Software Interaction Analysis) at upper level; wider techniques are then used at lower level in order to enforce and provide feedback to RAMS analysis [3], [4], [5]. They cover all the phases of the development process: design principles and techniques (e.g., reuse, modularity, partitioning, or supporting techniques as simulations, mathematical modelling), coding standards and convention, software verification and validation (V&V) techniques (e.g., testing and analysis), assessment techniques (e..g, measurements-based RAMS assessment), fault tolerance mechanisms. The key issues in actually implementing these techniques regard their cost-effectiveness in relation to the quality to assure; in the case of software, this presents unique and hard-to-tackle challenges, both in the “cost” and in the “quality assurance” aspect.

In the following, we briefly survey: what is required to developers in order to produce dependable SCSs; what are the current state-of-the-practice in industry; what is the contribution that the Critical Step Project provides.

2 What is required to do: certification standards

Software certification is a key aspect of critical systems development and assessment, and its influence on development practices and relation to systems cost is relevant. It is a matter of fact that certification of software is crucial for many companies developing mission- and safety-critical systems. As a result of software-related disasters, professionals and authorities are convinced that certification is nowadays inevitable. But at present, there is no common agreement on

what practices are more suitable to provide evidences, safety cases or insurance on which to base software certification (and consequently system certification). Several organizations (such as FAA, NRC, EUROCONTROL, CENELEC, IEC, ISO) produced, in the past, standards for developing critical systems in different domains, e.g., avionics, railway, automotive, nuclear, healthcare. These standards are conceived to provide recommendations about activities in the software development lifecycle (SDLC). For this reason, they are viewed as *process-oriented* standards, working under the assumption that high and controlled quality in software development activities along all the process implies a high product quality. Evidences are required on every produced artifact during the development in order to claim the certification of the final product.
A process-oriented certification process involves four main entities: the standard(s), an applicant, an authority and an assessment body that has to be independent on the applicant. The certification process typically implies interactions between the applicant and assessor, so as to drive the SDLC. Applicant has to provide evidences (sometimes referred to as "certification package") that the standard recommendations have been properly implemented. The assessment body evaluates the certification package and the software product (if it is available at that specific stage) in order to prove that they complain to the standards. The authority releases the certification to the applicant on the basis of the assessment body evaluation, or it can ask for further evidences.
In practice, it is not trivial to apply recommendation and to produce evidences for several reasons; for instance: *i)* standard guidelines do not prescribe a precise set of techniques, but they are recommendations; *ii)* applying a technique may yield very different results depending on the way and the extent it is applied to the specific software under assessment; *iii)* it may happen that applying a technique thoroughly requires unacceptable cost, and thus the applicant needs to find the most cost-effective way to apply it and produce the required evidences.

A lot of certification standards are in effect. We can group them on the basis of the industrial domain in which they are applied, such as nuclear, avionic, automotive. Examples are: DO-178B and DO-178C, in the Avionic/Aeronautics Domain; CENELEC EN 50126, EN 50128 and EN 50129 in the railway domain; ISO 26262 in the Automotive Domain; IEC 61508 for the industrial domain; ECSS standards for the Space Domain. Despite this wide variety of standards, they have many aspects in common. All of them require activities (and related documentation) for quality assurance along all the SDLC phases, from planning to deployment. In most of cases, software is not thought as a standalone part of the system, it does not stand on its own; thus, such activities start from system-level analysis, and then are linked to software: in a typical standard-compliant process, the following steps and related activities are outlined (e.g., [5]):

- from system-level activities, the following artifacts are given as input to the software development process: system requirements specification, system safety requirements specification and system architecture description: safety requirements are derived from the risk analysis (i.e., the activity of identifying risks, estimating their severity and occurrence, and mitigation strategies);

- from these artifacts, safety functions allocated to software are identified; safety functions are assigned an integrity, or assurance, level to satisfy (the name varies with the standard), representing the risk associated to that function (scales vary according to the standard);
- safety functions allocated to software are used in the software requirements specification phase, and in the software architecture specification (which is based on system architecture information); software requirements are then apportioned to software components in the architecture;
- from these artifacts, software is designed, implemented and verified/tested according to a selected SDLC, and according to tools, for which usage further rules are specified depending on the standard, in order to guarantee that they do not introduce faults;
- software is finally validated, and handed over to system engineers;
- the operational life of the system and its maintenance is also regulated.

Besides these phases, other aspects that are addressed by almost every standard are the following: recommendations for integration of Commercial Off The Shelf (COTS) software, reusability recommendations (also of legacy software); fault tolerance recommendations; requirements traceability recommendations; use of tools recommendations.

3 From theory to practice: current solutions and open challenges

Although certification standards provide a valuable support, one of the major problems is that the guidelines they suggest are quite general, since their purpose is not to define what techniques a company must use, or what is their impact on company's cost. For instance, cost and effectiveness issues are often neglected in such guidance documents. As a consequence, there is a gap between what they suggest and strategies, techniques, and tools that can actually be adopted by a company. For many of the proposed practices, there are contradictory studies about their actual effectiveness. This uncertainty poses serious difficulties to companies, which on one hand are constrained to meet predefined certification goals, whereas, on the other hand, are required to deliver systems at competitive cost and time.
Standards' annexes list a number of techniques recommended for each phase of the SDLC. Among the many available techniques, in this Section we survey only the most used ones, in order to have an idea about the type of activities carried out in practice for each phase.
In the early stages, when safety requirements need to be defined at system and software level, and allocated to software components, techniques for RAMS analysis first come into play. RAMS analysis starts at the very beginning of the system development, but it interacts with system (and software) development along all the development activities, providing useful information to them and, at the same time, getting feedbacks from them. As development goes on, RAMS

analysis becomes more and more accurate, since it obtains more information from results of the activities.
As for the software development activities, the best software engineering state-of-the-practice techniques and principles are adopted, from requirements to maintenance phase. However, even being the best in this field, the relative immaturity of software engineering make it challenging to provide highly safe software.

3.1 SW RAMS analysis techniques

SFMEA
Software FMEA (Failure Modes and Effect Analysis)and FMECA (Failure Modes, Effect and Criticality Analysis) are widely used to analyze failure modes and effect for software components. SFMEA/SFMECA aims at identifying software-related design deficiencies; it determines the effect of hardware failures and human errors on software operation, and the effect on the system of a (software) component failing in a specific failure mode [3].
SFMEA/SFMECA is based on the more established FMEA/SFMECA [6] for hardware, and has a similar structure. It includes an initial set up of a list of failure modes; failure modes are meant as the possible incorrect behavior of the software, and include: computational failures, algorithmic failures, synchronization failures, data handling failures, interface failures [4]. Then it analyzes the possible causes and consequences (in terms of local component-level effects and final effects); from its output, several indications for the overall development are derived, such as: recommendations for mitigating the identified software failures at design level; guidance for criticality level assignment to the components (at lower levels of the SDLC); suggestions to allocate V&V activities on the most critical software components. It also produces knowledge about possible software failures useful for successive developments.
Despite the similarity with hardware FMEA, there are relevant differences between SFMEA and FMEA making its application considerably trickier [7]. The most relevant ones are: *i)* in FMEA, system is considered free from failed components, whereas in SFMEA system is considered as containing software faults, that may lead to failures if triggered; *ii)* failure modes are totally different, and hard-to-define for SFMEA; *iii)* in SFMEA, measures taken to prevent or mitigate the consequences of a failure are different: they can, for example, show that a fault leading to the failure mode will be necessarily detected by the tests performed on the component, or demonstrate that there is no credible cause leading to this failure mode due to the software design and coding rules applied.
The main challenges in SFMEA/SFMECA come from failure modes definition. In [7] authors point out that the term failure mode is different for hardware and software. For hardware components it is straightforward and can be based on operational experience of the same and similar components; for software such history-based information is much less reliable and uniquely identifiable. Moreover, the frequency of occurrence is much harder to define for a software-based system, and their probability distribution over time is much less characterizable. Triggering and propagation depends on the operational profile and on complex

interactions between software at different layers (e.g., OS, middleware, other applications); sometimes it even appear to be non-deterministic [8]. This makes SFMEA/SFMECA trickier to apply than the corresponding hardware counterpart.

SFTA

The output of SFMEA can be used in combination with another widely used technique, namely the Software Fault Tree Analysis (SFTA). Also in this case, SFTA comes from system-level FTA [9]. FTA aims at analyzing events or combination of events that can make the system fail (i.e., that can lead to a hazard). Starting from an event representing the immediate cause of a hazard (named 'top event'), the analysis is carried out along a tree path, in which events combination is described with logical operators (AND, OR, etc). The analysis stops when basic events are reached, representing elementary causes whose further decomposition is not of interest. Probabilistic analysis are performed by assigning probabilities to basic events, and computing the top event probability. Basic FT works provided that there are no dependent events. There are many extensions to basic FT, to perform more complex analysis than the simple combinatorial one allowed by basic FT, e.g., when dependencies are involved.

Fault Tree Analysis is mainly meant for hardware systems, but it is used also for software failures analysis. When used in conjunction with SFMEA/SFMECA, the identified software failure modes are useful to construct the software fault tree. As for SFMEA/SFMECA, there are interactions between SFTA and development activities; output of SFTA may help to identify critical software components, to identify the mitigation means able to inhibit the occurrence of the top event failure, to analyze software design with respect to the top event failure occurrence and take design decisions (e.g., partitioning components), to help V&V (for instance, if used at source code level, it helps to understand the relation between faults and identified critical failures, in turn useful for test case writing and techniques selection) and testing resource allocation.

The problems of SFTA are the following: it is not suitable for state-based analyses, whit dependencies among events; it does not scale well, since trees can become very large and complex; it has the same problems of SFMEA about to the ambiguity of software failures and software faults. The latter is not an inherent problem of the techniques, but it is about the nature of software itself which is very difficult to characterize.

SCCFA

SCCFA (Software Common Cause and Failure Analysis) derives from CCA (Common Cause Analysis). The purpose of CCA is to identify any accident sequences in which two or more events could occur as the result of one common event [6]. Examples in computer systems are common physical location (e.g., if a system is in one single room, shortcomings in the air-conditioning or external events such as fire or earthquakes are common mode failures), common design processes (e.g., error in the common specification of diverse components), common errors in the maintenance procedure.

SCCFA aims at identifying these causes, relatively to software failures, and at providing recommendations to mitigate them. It can be used in conjunction with SFMEA/SFMECA and SFTA that can help in identifying dependencies among groups of components, which is essential in SCCFA. The basic steps of CCA/SSCFA include [6], [10]: *i)*: identification of critical components group to be evaluated (by identifying the physical/functional links in the system, functional dependencies and interfaces); *ii)* within the groups, check for commonalities such as physical location, a common design process that could introduce a generic design defect; *iii)* within each identified commonality, checking for credible failure modes (SFMEA/SFMECA can help); *iv)* identifying causes or trigger events that could lead to the failure modes; *v)* based on the above, draw conclusions and make recommendations for corrective action. Corrective actions include requirements redesign, invoking emergency procedures, and function degradation. SSCFA can be seen as a complement to the previous ones, since it uncover system failures caused by *common* software failures. However, this failures are very difficult to uncover, especially at requirement/design stage, and it requires expensive manual reviews/inspection activities and highly skilled and experienced personnel.

Other techniques that are worth to mention for supporting RAMS analysis are, the Hardware-Software Interaction Analysis (*HSIA*) [11], aiming at examining the hardware/software interface of a design to ensure that hardware failure modes are taken into account in the software requirements, the State Machine Hazard Analysis (SMHA), used to determine software safety requirements directly from the system design, to identify safety-critical software functions, and to help in the design of failure detection and recovery procedures and fail-safe requirements [1].

3.2 Software Engineering Techniques in the SDLC

RAMS analysis supports the SDLC activities and receives feedback from them. Standards recommend a set of techniques and practices for each of the SDLC phase. Companies try to comply the standard and at the same time to reduce the cost, by selecting techniques that they believe more suitable for their product/process.

We briefly review the most commonly used ones in practice. In the early stage, despite the advantages of formal languages, the software requirements specification is mostly based on natural language. It is the easiest way to specify the user needs, to communicate with stakeholders, as well as among developers, and to document the software product at this stage. Of course, the main problem of natural language is ambiguity; thus, often natural language is supported by structured notations, by multiple level of specification, or by semi-formal modeling techniques (e.g, requirement diagram, use cases). Whenever possible, formal methods are adopted; they are more often used on smaller critical systems (e.g., automotive or control systems domain), or on most critical parts of the system (RAMS analysis can help in identifying them). The practical problem of formal methods is the cost for specification, especially in large systems, and the cost deriving from the required skills: moreover, since interaction with stakeholder is

more and more common also in these domains, it may happen that requirements need to be specified also in some form of natural language, besides their formal specification. One more very common recommendation that is by now adopted by many companies is the support to a full traceability, from requirements to code and to the corresponding test cases.
At design stage, there are principles commonly accepted by many companies, such as modularity, information hiding/encapsulation, iterative refinement, temporal/spatial partitioning, low decoupling and high cohesion, reuse. This is often part of process flows definition in companies, at least theoretically. Instead, some of the most expensive design strategies among which producers typically are called to choose to optimize their cost-quality trade-offs are related to architectural choices to prevent, remove or tolerate faults: hardware-software redundancy, N-version programming, safety bag, recovery block, backward/forward recovery, reconfiguration, defensive programming, design by contract, design for change, formal methods. In many of these cases, there is the support either of modeling (semi-formal or formal) and/or of simulation. More recently, model-driven engineering is gaining ground in this field.
As for coding, standards may recommend specific restrictions on the coding process, such as the adoption of particular techniques, code style, or programming language, tied to a specific safety level. For instance, some kinds of programming techniques, such as recursion, may be prescribed for the highest safety levels. Some functionalities covering a critical role may require the adoption of a language that does not permit dynamic memory allocation. Companies may have their own coding standard, specifying restrictions adhering to the standard, such as avoidance of uninitialized variables, low cyclomatic complexity, limited use of pointers, use of naming conventions, correct indentation.

After implementation, **the most expensive phase** takes place: **Verification & Validation (V&V)** [12]. On this point, companies really strive to find the best cost-quality point. A lot of techniques exist, and are permitted by standards. Big families are testing and analysis techniques: the former used typically after the implementation, while the latter are more used from requirements specification to coding (especially static manual analysis). As for testing, we may distinguish three main groups: functional testing, non-functional testing, structural testing. Typical functional testing techniques are random testing and partition-based testing. Others, e.g., statistical testing, are less used. Functional testing is by far the most adopted technique, and often times also the only adopted technique at system-level, despite the criticality of systems. At lower level (e.g., unit testing) structural testing is adopted, also known as white-box testing. Techniques in this category are distinguished according to the coverage criterion: statement coverage, branch coverage, condition coverage, MC/DC coverage. Standard may prescribe coverage adequacy according to one of these criterion (e.g., DO-178B requires MC/CS full coverage [13]). Non-functional testing includes techniques to test quality requirements (in SCSs dependability requirements), such as performance testing, stress/load testing, robustness testing. These are in some cases required by standards (for high level of safety); for in-

stance, performance testing is suggested by CENELEC EN 50128 for railway [5], whereas robustness testing is required by DO-178B for avionics [13]. Performance testing aims at evaluating non-functional performance requirements fulfillment (such as constraints on response times). Stress testing evaluates the application's ability to react to unexpected loads. Robustness testing generates test cases aiming to evaluate the system behavior under exceptional conditions. Thus, it deliberately forces the system with unexpected inputs and observes its ability to manage such values. Robustness testing is often used in conjunction with functional testing techniques especially in critical systems, since its purpose is the opposite of functional testing (it has to verify that the system does not do what is not required).
As for analysis techniques, the most common ones are code/design inspection. They are static manual analysis techniques, and are the principal means by which artifacts consistency and correctness at different stage is verified, from requirements completeness to design, down to the code. Code analysis is also often supported by automatic analysis tools for determining, for instance, metrics of interest of the produced code, such as cyclomatic complexity, size metrics, Halstead's science metrics, coupling metrics, and others.
Finally, in the cases in which formal methods are used at upper level (prescribed at some highest critical levels of some standards), formal verification techniques are adopted at this stage. They include all the verification techniques that automatically verify the software's correctness against its specification: model checking, symbolic execution through pre and post condition, formal proof.
As for maintenance, a common supporting strategy is keeping records of data produced during software development process and during operation. Such data are then analyzed to facilitate software process improvement starting from relevant data from about individual projects and persons. Also, if traceability is implemented, the impact analysis is used at this stage, in order to identify the effect of a change in the product before implementing it.

4 Contribution of the Critical Step Project

The *Critical Step* project favored the development and exchange of know-how in topics of interest in the field of SCSs. With reference to the topics mentioned above, involved researchers, other than knowing about the world of SCSs, of their domains, and of the related certification standards, gained key competences covering very important parts of the SCS development cycle, namely the software dependability analysis and evaluation. The focus has been posed on two complementary perspectives, whose themes have been, during the project, subject of several knowledge exchanges and joint work: V&V issues on the one hand, and RAMS analysis process and methods on the other hand.

These topics have been dealt with in the context of certification standards, which have been the common ground on top of which knowledge has been developed and shared. Several standards have been surveyed in their basic feature, with particular reference to the V&V phase and RAMS analysis.

More specifically, V&V has been studied mainly with respect to non-functional requirements. Works on robustness testing in the field of critical systems have been jointly conducted, with the aim of reporting experiences about the usage of techniques for robustness evaluation in middleware software infrastructure in critical contexts (e.g., publish/subscribe and web services). Further testing techniques, such as fault-injection, have been also used to risk assessment and robustness evaluation, being both valuable outputs for RAMS analysis. From the RAMS perspective, researchers acquired familiarity with the whole process of RAMS analysis at software level, with its interaction with system-level RAMS and with SDLC phases, especially with testing. The main supported and gainful techniques in real industrial contexts have been taken into account.
Overall, besides single joint works, the project developed background and guidance to address software-specific challenges at RAMS and V&V level, on how one can benefit from the other, and on potential opportunities to improve the software product cost-quality balance.

References

1. Leveson, N. G.: The role of software in spacecraft accidents. AIAA Journal of Spacecraft and Rockets **41** (2004) 564–575
2. Calzarossa, M.C., Tucci, S.: Performance Evaluation of Complex Systems: Techniques and Tools. Performance 2002 Tutorial Lectures, Lecture Notes in Computer Science **2459** (2002)
3. European Organization for the Safety of Air Navigation: Review of Techniques to Support the EATMP Safety Assessment Methodology **I** (2004)
4. European Cooperation on Space Standardization (ECSS): ECSS-Q-HB-80-03 Draft (2012)
5. CENELC: EN 50128:2011 – Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems (2011)
6. Amberkar, S., Czerny, B.J., D'Ambrosio, J.G., Demerly, J.D., Murray, B.T.: A Comprehensive Hazard Analysis Technique for Safety-Critical Automotive Systems. SAE technical paper series (2001)
7. Pentti, H., Atte, H.: Failure Mode and Effects Analysis of software-based automation systems. VTT Industrial Systems **190** (2002)
8. Grottke, M., Trivedi, K.S.: Fighting bugs: Remove, retry, replicate, and rejuvenate. IEEE Computer **40**(2) 107–109 (2007)
9. Vesely, W.: Fault Tree Handbook with Aerospace Applications. NASA office of safety and mission assurance, Version 1.1 (2002)
10. Stephens, R.A., Talso, W.: System Safety Analysis handbook: A Source Book for Safety Practitioners. System Safety Society, 2nd edition (1997)
11. Von Hoegen, M.: Product assurance requirements for first/Planck scientific instruments. PT-RQ-04410 (1) (1997)
12. Pezze', M., Young,M.: Software Testing and Analysis: Process, Principles and Techniques. Wiley (2007)
13. RTCA and EUROCAE: Software consideration in airborne systems and equipment certification (1992)