CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Conference Paper

How realistic is the mixed-criticality real-time system model?

Alexandre Esper
Geoffrey Nelissen
Vincent Nélis
Eduardo Tovar

CISTER-TR-151004

2015/11/04

How realistic is the mixed-criticality real-time system model?

Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, Eduardo Tovar

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: aresper@criticalsoftware.com, grrpn@isep.ipp.pt, nelis@isep.ipp.pt, emt@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

With the rapid evolution of commercial hardware platforms, in most application domains, the industry has shown a growing interest in integrating and running independently-developed applications of different “criticalities” in the same multicore platform. Such integrated systems are commonly referred to as mixed-criticality systems (MCS). Most of the MCS-related research published in the state-of-the-art cite the safety-related standards associated to each application domain (e.g. aeronautics, space, railway, automotive) to justify their methods and results. However, those standards are not, in most cases, freely available, and do not always clearly and explicitly specify the requirements for mixed-criticality systems. This paper addresses the important challenge of unveiling the relevant information available in some of the safety-related standards, such that the mixed-criticality concept is understood from an industrialist’s perspective. Moreover, the paper evaluates the state-of-the-art mixed-criticality real-time scheduling models and algorithms against the safety-related standards and clarifies some misconceptions that are commonly encountered.

How realistic is the mixed-criticality real-time system model?

Alexandre Esper[†‡], Geoffrey Nelissen[‡], Vincent Nélis[‡], Eduardo Tovar[‡]
[†]Critical Software S.A.
Porto, Portugal
aresper@criticalsoftware.com
[‡]CISTER/INESC-TEC, ISEP
Porto, Portugal
{grrpn, nelis, emt}@isep.ipp.pt

ABSTRACT

With the rapid evolution of commercial hardware platforms, in most application domains, the industry has shown a growing interest in integrating and running independently-developed applications of different "criticalities" in the same multicore platform. Such integrated systems are commonly referred to as mixed-criticality systems (MCS). Most of the MCS-related research published in the state-of-the-art cite the safety-related standards associated to each application domain (e.g. aeronautics, space, railway, automotive) to justify their methods and results. However, those standards are not, in most cases, freely available, and do not always clearly and explicitly specify the requirements for mixed-criticality systems. This paper addresses the important challenge of unveiling the relevant information available in some of the safety-related standards, such that the mixed-criticality concept is understood from an industrialist's perspective. Moreover, the paper evaluates the state-of-the-art mixed-criticality real-time scheduling models and algorithms against the safety-related standards and clarifies some misconceptions that are commonly encountered.

1. INTRODUCTION

In the last decade and in most application domains, the industry has shown a growing interest in developing methods and tools to implement, deploy, validate, and certify independently-developed applications of different "criticalities" in the same multicore platform, with the evident objective of improving the performance/cost ratio of the system. Such integrated systems are commonly referred to as mixed-criticality systems (MCS). All over the world, the industrial interest in MCS has manifested itself in the form of important investments placed into R&D projects and academics since long time started to manifest their interest as well. The research community that focuses on the real-time scheduling theory has actively taken part in these efforts regarding MCS. Their base application model has quickly developed into a mixed-criticality (MC) task model that is today well-accepted and used in most research works on the subject. This new MC task model is in essence the result of combining the standard hard real-time requirements (studied by the real-time research community since the 70's) with the notion of "criticality" of execution. When transposed into the industrial world, the applications that correspond the best to that MC model and its combined requirements are those in which a part of the core functionality is delivered by *safety-critical* components.

The introduction of new constraints and requirements into the theoretical models has unexpectedly unveiled a brand new research landscape. Into this virgin research field some of the seminal results in scheduling theory had to be restated and revalidated, and an entire body of knowledge was to be rebuilt. The popularity of MCS immediately soared up in the real-time research community, which has been evidenced by the sudden emergence of tracks, sessions, and workshops that are now entirely dedicated to MCS in most of the flagship conferences on real-time systems.

Since its conception, the MC model has been gradually gaining sophistication by incorporating multiple levels of criticality or probabilistic WCET estimates to mention a couple of examples. Each transformation of the model has been motivated and justified as a mean to better cope with the requirements of MCS. However, due to the price of safety-related standards that provide and formally state those requirements (as well as design guidelines and recommendations) for various application domains, many researchers have been constrained to work on models and arguments that build on top of previously published (academic) models and (academic) claims. One of the risks and major weakness in such an approach is that it facilitates the propagations of misconceptions and the drift from the intended meaning reflected in the standards. In this paper, we therefore argue that in recent academic papers, many claims and arguments supposedly taken from the standards have been progressively tweaked, misinterpreted, or simply taken out of their initial context. Throughout this paper we will to elaborate on these various misinterpretations and will identify and discuss the motivating arguments that should probably be reconsidered in future work on MCS. In our discussions we only refer to recommendations and requirements for the design of safety-critical applications using the three following standards: the IEC61508 (generic electrical and/or electronic and/or programmable electronic (E/E/PE)) [16], the ISO26262 (automotive domain) [17] and the DO-178C (aeronautics domain) [7]; those three standards being the most commonly cited in the real-time research literature on MCS.

Organisation of the paper: First, we present in Section 2

RTNS 2015, November 04-06, 2015, Lille, France

 ISBN 978-1-4503-3591-1/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2834848.2834869

an overview of the safety assessment process as required by the standards for the development of safety-critical systems. We explain how in practice the development assurance levels (DALs) are assigned to the system safety functions. With this background, in Section 3 we introduce the concept of a MCS. In Section 4 we present several architectural considerations and requirements from the three above mentioned safety-related industrial standards that are applicable to the development of MCS. In Section 5 we discuss the theoretical MCS model and some misconceptions that are commonly encountered in the academic literature. The paper is concluded in Section 6.

2. SYSTEM DESIGN AND DEVELOPMENT ASSURANCE PROCESS

During a typical development life cycle of a safety-critical system, the behavior and characteristics that are expected from the system are expressed in the form of a list of requirements. Those are developed based not only on the system operational requirements (what the system is expected to do), but also considering non-functional properties related to safety, security and performance, including timing and energy constraints. In order to ensure the safety properties of a safety-critical system, a *system safety assessment process* must be carried out as part of the development life cycle to determine and categorize the failure conditions of the system (e.g. through a hazard analysis). As a result of the system safety assessment process, safety-related requirements are derived, which may include functional, integrity, dependability requirements and design constraints. These requirements are then allocated to hardware and software components, thereby specifying the mechanisms required to prevent the faults or to mitigate their effects and avoid the propagation of failures.

To help understand the safety-critical system development lifecycle, we provide below an overview of the safety assessment process as defined by the standards. We hence explain how, in practice, the *development assurance levels* (DALs) are assigned to the system safety functions.

2.1 Safety Assessment Process

The safety assessment process starts *at the system level* with a hazard analysis. This technique identifies and evaluates hazards that are produced by the system by taking into account its environment. The software hazard analysis is a top-down technique that makes recommendations to eliminate or control software hazards and relates the hazards to the interfaces between the software and the system. Software hazard analysis should ensure that the software does not interfere with the objectives and correct operation of the system and if interference cannot be totally avoided then it must also evaluate and make recommendations to mitigate how the software can hinder the objective or operation of the system.

After the software hazards have been identified and categorized according to the severity of their consequences, a fault analysis is typically performed to support the hazard analysis in the evaluation of the effects of failures (see Tables A.10, B.4 of IEC61508-3 [14]). Several techniques exist for that purpose, including, e.g., fault tree analysis (FTA) [15][9], failure modes and effects analysis (FMEA) [14][8], common cause failure analysis (CCA) [11], etc. These techniques are widely applied across all application domains, and even though they are applied with slightly different terminologies, the concepts involved are basically the same. Note also that a fault analysis such as FMEA may also help identify hazards that were not yet identified by the hazard analysis.

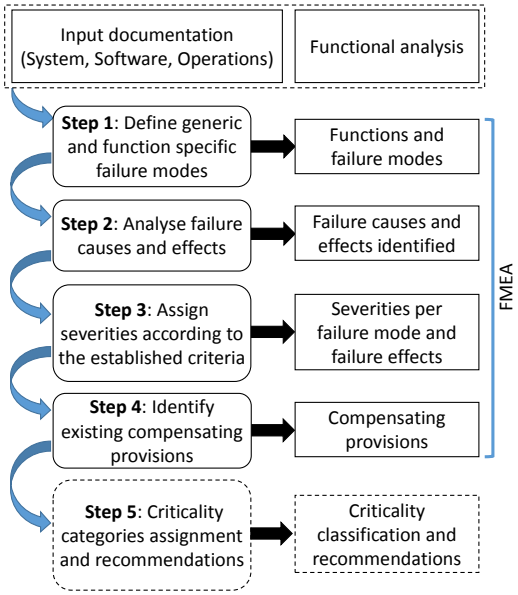

Figure 1: Generic FMECA process.

We briefly describe the fault analysis process using FMEA as an example, which is a powerful bottom-up technique for identifying potential software failures. Fig. 1 summarises the generic software FMEA process. The first step consists in performing a functional analysis, i.e. a listing and description of the software functions (rather than the items used in their implementation), and to define the generic failure modes to be applied to each function of each software component. The generic failure modes typically include incorrect execution, non-execution or late execution of a function. Therefore, based on the input documentation and on the previously identified software functions derived from the software requirements, the generic failure modes are mapped to the functions of each software component and the component's specific failure modes are then derived. Step 2 consists in analysing the component possible failure causes that can trigger the identified failure modes and on identifying the end effects of the failure mode to the system (possible failure propagation). During Step 3 severity categories (i.e., how bad is the consequence for the system) are assigned to every single failure mode. The severity assignment is based on the failure mode end effects and is performed in accordance with the severity assignment criteria used for the project. Step 4 is the identification of existing compensating provisions that can circumvent or mitigate the effect of the failure, control or deactivate product items to halt the generation/propagation of failure effects, or activate backup or standby components to (at least partially) recover from the failure. Generally speaking, design compensating provisions include:

- Redundant components or alternative modes of operation that allow continued and safe operation;
- Safety or relief feature (hardware or software) that allow effective operation or limit the failure effects.

Once the severity categories have been assigned to the failure modes and consequently to the respective software components, the software FMEA can be extended to a software FMECA (failure modes, effects and *criticality* analysis). Through the software FMECA, a classification of the analysed software can be performed based on the severity of the consequence of the potential failure modes (Step 5 in Fig. 1). For each failure mode identified, the existing compensating provisions that can mitigate the failure effects are then analysed and a *development assurance level* (DAL)[1][7] is further assigned to the software component, based on the failure modes with highest severity and on the effectiveness of the identified compensating provisions. In the end of the process, recommendations can be provided with the objective of reducing the risk associated with the potential critical faults identified (e.g. by increasing the amount and rigour of verification and validation activities).

2.2 Development Assurance Level

The software DAL establishes the necessary rigour of the development and of the verification and validation (V&V) activities that need to be performed on the software, in accordance with the adopted standard. The higher the DAL of a software, the higher the number of assurance activities that need to be performed, thus also increasing considerably the costs of its development. The process for the development of safety-critical software, which assures the software safety and dependability properties at a certain DAL is defined in several standards across several application domains. Typically, the DALs are divided into 4 or 5 levels, related to the categories of severity of a failure adopted by the standard. Under the DO-178C [7] standard (in the avionic domain), five severity categories are defined. These five categories are: (i) **catastrophic**: failures that result in multiple fatalities or the loss of the airplane; (ii) **hazardous**: failures that result in serious or fatal injury to a relatively small number of occupants; (iii) **major**: failures that reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions; (iv) **minor**: failures that would not significantly reduce the airplane safety; (v) **no safety effect**: failures that would have no effect on safety.

The software DALs are then assigned depending on the severity category assigned to the failure(s) that may be caused by the analysed software component. Specifically, there are five levels defined as level A/B/C/D/E which are respectively assigned to software components (or modules) whose anomalous behaviour would lead to a system failure of catastrophic, hazardous, major, minor or negligible consequences. That is, a software that can potentially contribute to a catastrophic system failure shall be developed according to DAL-A requirements.

So far we have used DO-178C as reference for explaining the concept of development assurance level (DAL). However, IEC61508 and ISO26262 use different terminologies for describing the development process of a safety-critical or safety-related system, although the fundamental concepts are in essence the same.

IEC61508 defines the concept of *safety function*. Safety functions are implemented by a safety-related system whose purpose is to achieve or maintain a safe state for the equipment under control (e.g. car engine) when a specific hazardous event occur. Associated to the safety functions, the concept of *safety integrity* is defined, which refers to the probability of a safety-related system to satisfactorily perform the required safety functions under all the state conditions within a specified period. There are four *safety integrity levels* (SIL). The higher the safety integrity level of the safety function, the lower the probability that the safety-related system that executes that function will fail. Software SILs are used as the basis for specifying the safety integrity requirements of the safety functions implemented by safety-related software. Although the SIL is composed of four levels, the IEC61508 does not explicitly define the failure severity categories and their association with the SIL. Only examples are provided that are not fully detailed. For instance, in Table C.1 of IEC61508-5 [16], the following failure severity category levels are provided: catastrophic, critical, marginal and negligible. It is up to the project to define and detail those categories but it is important to note that the definition of those is based only on *qualitative* rather than quantitative measures. This note will be further discussed in Section 5.6.

ISO26262 derives from the generic IEC61508 and addresses the specificities of the automotive sector. ISO26262 defines the *automotive safety integrity level* (ASIL). Similarly to the SIL defined in IEC61508, the ASIL are composed of four levels, where D represents the most stringent and A represents the least stringent level in terms of requirements and safety measures (note that this is the exact opposite to the scale used by DO-178C). The higher the ASIL, the greater the needs to reduce the risk. Fig. 2 presents the risk matrix for the ASIL determination of hazardous events of automotive systems. It uses three parameters: "severity", "probability of exposure" and "controllability"[2,3]. The severity defines the estimation of the extent of harm to one or more individuals that can occur in a potentially hazardous situation, the associated probability is the likelihood of the occurrence of harm, and the controllability is the ability to avoid a specified harm or damage through the timely reactions of the agents involved (e.g. the driver of the vehicle) possibly with support from external measures. Therefore, the ASILs explicitly consider one more parameter in comparison to the SILs, which is the ability to control failure effects. Note that, as it can be seen in Fig. 2, a high controllability (class C1) can often help reduce the ASIL of the components by 2 levels in comparison to the case where the controllability is almost inexistent (class C3).

3. THE NOTION OF MIXED-CRITICALITY SYSTEMS

Considering the process presented above for assigning the DALs, the concept of mixed-criticality becomes straightforward to understand. A mixed-criticality system basically consists of applications of different DALs coexisting in the same system, sharing the same resources (potentially including the CPUs) but still preserving the safety characteristics of each individual application as required by the domain-specific safety-related standards.

[1]Terminology commonly used in the aeronautic domain. In the European space standards (ECSS), for instance, the term software *criticality category* is used instead [12]. Other terminologies are used in other domains as described later.

[2]A detailed description of these 3 parameters are outside the scope of this work. Please refer to [17] for further details.

[3]In addition to the four ASILs, the class QM (quality management) denotes no requirement to comply with ISO26262 other than the project quality assurance requirements.

Severity class	Probability class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Figure 2: ASIL determination for hazardous events.

In order to successfully develop a mixed-criticality system, all safety-related standards in every application domain advocate the use of design techniques that assure *simplicity* and *modularity* in the design. These guidelines can be justified in a thousand ways, but in short, a simpler design guarantees a simpler conception phase and a simpler and thus less costly V&V, while modularity allows for better maintainability and easier upgradability.

The high-level process explained above for assessing the criticality of software systems is an important activity in the design of safety-critical systems, and consequently of mixed-critical systems. Based on the understanding of this process, one can conclude that there are two main solutions to reduce the criticality of a system component (i.e., to reduce the risk of severe failures):

1. Avoiding the propagation of faults between different components and in particular from low criticality components to higher criticality components;
2. Providing compensating provisions by adding effective mechanisms that could either prevent or mitigate the effects of a failure.

It also becomes clear that improving the reliability of the software, by reducing the risk of failures, is an essential step in the design of MCS.

4. REQUIREMENTS OF SAFETY-RELATED INDUSTRIAL STANDARDS

Now that Sections 2 and 3 introduced the required concepts, methodologies and terminology for understanding and discussing MCS, this section briefly summarizes some key architectural considerations and requirements extracted from three safety-related industrial standards (IEC61508, DO-178C and ISO26262) that are applicable to the development of MCS.

4.1 The IEC61508

The IEC61508 [16] is a generic safety standard widely used throughout the industry. It serves as a common base for domain specific standards such as the ISO26262 [17] (automotive), or EN50128 [13] (railway). IEC61508 is composed of a series of eight volumes addressing the complete safety lifecycle activities for systems comprised of electrical/electronic/programmable electronic (E/E/PE) elements (including software) that are used to perform safety functions. This standard define strict rules regarding the *isolation* and *independence* between safety related and non-safety related functions. For instance:

"Where the software is to implement both safety and non-safety functions, then all of the software shall be treated as safety-related, unless adequate design measures ensure that the failures of non-safety functions cannot adversely affect safety functions." [section 7.4.2.8 of IEC61508-3]

"Where the software is to implement safety functions of different safety integrity levels, then all of the software shall be treated as belonging to the highest safety integrity level, unless adequate independence between the safety functions of the different safety integrity levels can be shown in the design. It shall be demonstrated either (1) that independence is achieved by both in the spatial and temporal domains, or (2) that any violation of independence is controlled. The justification for independence shall be documented." [section 7.4.2.9 of IEC61508-3].

Under IEC61508, several safety-related software design techniques and measures are presented as detailed below.

4.1.1 Partitioning

Partitioning is a technique that allows isolating software components from each other. This isolation is essential for critical systems as it allows the containment of faults, as well as the reduction of the software V&V effort. Typically, there are two approaches to achieve partitioning between software components. The first approach is to *physically* segregate the components by allocating unique hardware resources to each component (i.e., only one software component is executed on each hardware component composing the system). The second approach is to *virtually* separate the components by establishing partitioned hardware provisions that allow multiple software components to run on the same hardware platform.

Annex F of IEC61508-3 provides further recommendations on techniques for achieving non-interference between software elements on a single computer. In this context, the term "independence of execution" is used, meaning that applications should not interfere with each other's behaviour. This independence *shall* be achieved and demonstrated in both *spatial* and *temporal* domains. Spatial isolation means that one application shall not change data used by another application. Note that spatial isolation is even more important considering the fact that the highest severity software failure modes are typically associated to data corruption (e.g., due to buffer overflows or memory violation). Temporal isolation on the other hand shall ensure that one application will not cause malfunction of another application by consuming too high processor execution time or by blocking a shared resource used by other applications, thus affecting its timing properties. In order to demonstrate the independence of execution, an analysis of the proposed design is performed to determine the causes of execution interference in both spatial and temporal domain through the application of the methodologies described in Section 2.

The standard explicitly recommends the following techniques for achieving and demonstrating spatial independence (section F.4 of IEC61508-3): (1) hardware memory protection; (2) virtual memory space; (3) rigorous design, source code and possibly object code analysis; and (4) software protection of higher integrity applications.

Ideally, data should not be passed between applications of different criticalities. However, in practice, especially in MCS, there may be a need to exchange data between applications of different criticalities. Considering this, the system should ensure that higher SIL applications are able to verify the integrity of any data received from lower SIL applications. This can be achieved, for instance, through the use

of unidirectional interfaces such as messages or pipes, rather than through shared memory.

With respect to temporal independence, the following techniques (intrinsically related to the choice of scheduling policy) are mentioned by the standard (section F.5 of IEC61508-3):

1. Deterministic scheduling methods such as cyclic scheduling and time triggered architectures;
2. Strict priority based scheduling by real-time executive (with mechanism to avoid priority inversion);
3. Time fences that terminate the execution of an application in case it exceeds its time budget;
4. Time slicing, which ensures that no process can be starved of CPU time.

However, the resource sharing protocol is also important when sharing resources between applications, because the design shall ensure that the applications will not malfunction due to a locked resource. Therefore, it is essential that the time required to access a shared resource is taken into consideration when performing the schedulability analysis of the system.

Note that the software functions used to provide spatial and/or temporal independence (e.g operating system, real-time executive) shall be allocated the *highest criticality* of the applications running on top of them (section F.6 of IEC61508-3), since such software represents a potential common cause of failure of the independent elements.

4.1.2 Diverse monitor

The diverse monitor (section C.3.4 of IEC61508-7) is an architectural design technique that allows the protection against faults in software, preventing the system from entering an unsafe state. It is an external monitor, running in an independent hardware, which continuously monitors the main application. In the occurrence of a fault, the monitor will trigger an event (e.g., fire an alarm) so that a corrective measure can be activated, e.g., through a restart of the monitored application or through a human operator action. Typically, the utilization of a monitor allows to reduce the criticality of the monitored application. Indeed, following the FMEA analysis of the main application (see Section 2), the monitor would appear as a compensating provision that would prevent the failure from propagating throughout the system, thus reducing the criticality of the monitored application. However, in this case it can be considered that the monitor "inherits" the criticality of the monitored application, because if the monitor fails, there is typically no compensating provision to compensate for that failure. Therefore, the monitor is assigned a criticality derived from the highest severity failure modes of the monitored applications. In short, if the monitor can be certified at the highest criticality level, then the criticality level of the monitored component can be reduced under the condition that an effective corrective measure is available.

4.1.3 Dynamic reconfiguration

Another architectural design technique is the dynamic reconfiguration of the system (section C.3.10 of IEC61508-7), whose objective is to maintain the system functions operational despite an internal fault. This concept is more commonly applied to the recovery from hardware faults, but it can also be applied to software, if the logical architecture of the system can be mapped onto a subset of the available resources, e.g., through "run-time redundancy" to allow a software re-try or through redundant data, which can reduce the severity of the consequence of an isolated failure.

4.1.4 Graceful degradation

Graceful degradation is a technique aimed at maintaining the more important system functions available, despite failures, by dropping the less important functions. According to the IEC61508-7, section C.3.8:

"This technique gives priorities to the various functions to be carried out by the system. The design ensures that if there is insufficient resources to carry out all the system functions, the higher priority functions are carried out in preference to the lower ones. For example, error and event logging functions may be lower priority than system control functions, in which case system control would continue if the hardware associated with error logging were to fail. Further, should the system control hardware fail, but not the error logging hardware, then the error logging hardware would take over the control function.This is predominantly applied to hardware but is applicable to the total system including software. It must be taken into account from the topmost design phase."

As it will be further discussed in Section 5, most of the academic works on mixed-criticality scheduling claim to implement a graceful degradation strategy. To help understand the discussion of Section 5, note the three following properties of the quoted example: (1) the illustrative example involves a *high priority* function and a *low priority* function (i.e., it does not refer to criticality but priority); (2) the high priority task continues to run if the low priority task fails (i.e., a failure of a low priority task does not impact on the execution of a high priority task); (3) the hardware dedicated to the low priority function is used to execute the high priority task if the hardware of the high priority task comes to fail, thus stopping the execution of the low priority task. Note also that the example assumes a hardware failure, which leaves the system with not enough hardware resource to serve all the software functions.

4.1.5 Performance modelling

Performance modelling (section C.5.20 of IEC61508-7) ensures that the system operational capacity is sufficient to meet the specified throughput and response time requirements, considering any constraint on the use of system resources. The system processes and their interactions are modelled, including their demanded resources (e.g. CPU time) under average and worst-case conditions. Performance properties such as worst-case throughput and response times of the individual system functions are then calculated. To avoid the risk of resource starvation, the systems are often designed to use only some fraction of the total available resources. It is not uncommon that engineers apply a 50% margin on the use of such resources.

4.1.6 Response timing and memory constraints

It consists in determining the temporal and memory demands under average and worst-case conditions to ensure that the system requirements will be met (section C.5.22 of IEC61508-7). One of the methods to obtain these estimates is through prototyping and benchmarking of time critical systems. In terms of schedulability analysis, this is the usual analysis that needs to be performed on MCS to ensure that all safety-critical functions will successfully meet their deadlines under the given system constraints.

4.2 The DO-178C

The DO-178C standard describes a set of important techniques that can be applied during the design of avionics systems, which may prevent software failures and/or limit or circumvent their effects on the system functions. To achieve that goal, the system safety assessment process needs to demonstrate that the software components will execute with sufficient independence. This independence must be ensured at the functional level, i.e., during the specification of the high-level software requirements, and at the design level, e.g., definition of common design elements, languages and tools.

If sufficient independence between software components cannot be demonstrated (e.g., through partitioning), then those components will be viewed as a single software component when assigning the software DAL. This implies that the DAL assigned to the components will be the DAL associated with the highest failure severity category that those components can contribute to.

Under DO-178C, the following safety-related software design methods are discussed: partitioning; dissimilarity (or redundancy) and safety monitoring. Dissimilarity is a design technique also referred to as multi-version software, where two or more different software components that perform the same functions are developed independently (section 2.4.2 of DO-178C). It intends to avoid common sources of errors to contaminate the different versions of the same component. However, in the industry this technique is rarely applied due to cost issues and is thus not further discussed in this paper. Partitioning and safety monitoring as described in DO-178C are discussed in details below.

4.2.1 Partitioning

Similarly to IEC61508, DO-178C presents partitioning as one of the most important design instruments to safety-critical systems. The decision regarding the partitioning approach to be applied to a project must be taken during early phases of the software development life cycle (section 2.4.1 of DO-178C) and must address the following aspects: (i) the extent and scope of interactions that will be allowed between the partitioned components, (ii) how to isolate the components from each other, i.e., which protection strategy will be adopted (e.g through hardware functions or a combination of hardware and software).

Regardless of the adopted approach, DO-178C establishes five requirements for ensuring partitioning between the partitioned software components. The first requirement states that the code, input/output (I/O) or data storage areas of a software component cannot be contaminated by another software component that belongs to a different partition. The second requirement refers to the consumption of shared CPU time. A partitioned software component is only allowed to consume CPU time during its scheduled period of execution. Requirement three is related to hardware failures within a partition. Each partition should be able to contain the fault, i.e., it should not propagate to the other partitions and hence cause failure of software components in those other partitions. Requirement four discusses the DAL level of the software application that provides the partitioning functionality to the system. This requirement states that the software that implements the partitioning functionality should have the same or higher DAL than the highest DAL of the software components assigned to any of the provided partitions. If the partitioning functionality is provided via hardware, the fifth requirement requires that a safety assessment must be performed on that hardware to ensure that in case of failure it will not cause failures on the software partitions and consequently affect the system safety.

4.2.2 Safety Monitoring

As already mentioned in Section 4.1.2, safety monitoring (section 2.4.3 of DO-178C) is a technique that allows the protection against specific failures through the generation of events (e.g., alarms) and activation of protective mechanisms when the monitored function enters a faulty state. The safety monitoring functions can be implemented by hardware, software, or a combination of both. From the safety point of view, the safety monitor implements a safety barrier that will inhibit the failure of a software component from propagating throughout the system and adversely affecting its safety. Therefore, through the safety monitoring technique, the DAL level assigned to a software component will be derived from the severity of the consequence of the loss of the system function associated to that component. From the schedulability point of view, monitors are commonly used in safety-critical operating systems for the monitoring of the time budgets assigned to each application (or task). In case an application exceeds its time budget, an event is raised which is dealt with at the application level, i.e., each system may take different measures to compensate for those violations. In DAL-B systems for instance, the system could simply provide an indication for the user (a human or another system) that the integrity of the system has been compromised. This can be the case of aeronautic navigation systems, where several redundant instruments are available to aid performing the same navigation functions. This means that in case the integrity of a certain system has been compromised, it is still possible to use the readings of another instrument that performs identical functions.

DO-178C describes three important attributes that should be considered when designing the safety monitor. The first attribute is related to the monitor DAL assignment. The safety monitoring software inherits the DAL of the highest failure severity category associated with the monitored function. The second attribute is aimed at ensuring that the monitors are designed and implemented in such a way that it will detect the intended faults under all necessary conditions (otherwise it cannot be trusted and thus becomes useless). In order to ensure that all fault conditions are identified, an assessment of the system faults needs to be performed to ensure that the monitor will cover all cases. The last attribute refers to the independence between the monitoring and the monitored functions. The monitor and the protective mechanisms triggered by the events generated by the monitoring function should not be affected by the same failure causing the failure condition it is supposed to monitor. For instance, a monitor that is supposed to detect non-respected timing properties of software components (e.g., due to starvation), cannot be subject to the same source of blocking as the monitored tasks. In this case, the monitor and the monitored tasks should for example be associated to different partitions.

4.3 The ISO26262

ISO26262 is an adaptation of IEC61508 addressing the specific needs of the automotive sector. Therefore, everything discussed in the two previous subsections is also applicable to this standard. For instance, software partitioning

aspects are addressed in section 7.4.11 and in Annex D of ISO26262-6. Mechanisms for error detection at the software architectural level (including monitoring techniques) are listed in Table 4 of ISO26262-6. Several techniques for temporal and logical program sequence monitoring at the hardware level are also presented in Table D.10 of ISO26262-5.

From a shared resource viewpoint, if software partitioning techniques are to be applied, the resources shared between the partitions must be used in such a way that the software components running on the different partitions do not interfere with each other.

At the software architectural design level, the standard establishes that an upper estimation of required resources for the embedded software shall be made, which includes the execution time, the storage space (e.g. RAM for stacks and heaps) and the communication resources.

Annex D of ISO26262-6 [17] also provides some common examples of timing and execution faults that can cause interference between software elements of different partitions and must therefore be assessed before certifying the system: blocking of execution, deadlocks, livelocks, incorrect allocation of execution time and/or incorrect synchronization between software elements. To prevent or mitigate these faults, some mechanisms are also referred such as: cyclic execution scheduling; fixed priority based scheduling; time triggered scheduling; monitoring of processor execution time; program sequence monitoring and arrival rate monitoring. These important aspects must be considered when designing a real-time scheduling algorithm and/or a resource sharing protocol for MCS.

4.4 MCS and the challenge of compliance to safety-related standards

In the previous subsections, we have presented a summary of several requirements from industrial standards that must be considered in the design of MCS. Those can be transversally applied to several domains of application (e.g. aerospace, automotive, railway). Although the presented safety-related industrial standards do not explicitly specify requirements for MCS, they do specify stringent requirements that must be met to ensure the *safety* of the system, especially in terms of isolation and independence between applications running on the same platform. Notwithstanding, the irreversible and inevitable appearance of multicore hardware platforms in the industry introduces several additional challenges in terms of scheduling and resources sharing that make the isolation and independence of the mixed-criticality applications even more complex. The requirements presented are clear in what concerns the isolation and independence of applications, even when they share common resources. Therefore, *when designing a scheduling algorithm and/or resource sharing protocol that is intended to be compliant with such standards, it is necessary to provide evidences that the isolation between components is sufficient to avoid failure propagation between them.* To address these challenges, several techniques have also been described that can be applied to the design of such systems, which can support the generation of the evidences required by the certification authorities.

Industry already defined solutions easing the design and certification of safety-critical systems according to the standards. An example of such solution that meets the isolation and independence requirements commonly established by the previously presented industrial standards is the ARINC-653 [2] specification. ARINC-653 specifies the baseline operating environment for application software running on an Integrated Modular Avionics (IMA)[21, 6] platform or in traditional federated architectures developed according to the ARINC-700 avionics standards [3].

The purpose of an IMA system is to support the execution of one or more avionics applications independently. Each application may have completely different requirements and thus be associated to different DALs. The separation is achieved through partitioning, providing the functional separation of the applications (mainly to inhibit failure propagation), as well as the facilitation of the V&V activities. An ARINC partition is basically an environment running a program, comprising its own data, context, configuration attributes, etc. The primary objective of ARINC-653 is to define a general-purpose interface between the avionics application software and the operating system running on an avionics computer.

The partitioning concept is central to the ARINC-653 philosophy, whereby the programs resident on the partitions are partitioned with respect to space (memory partitioning) and time (temporal partitioning). The partitioned system has to be robust enough to support applications of different criticality levels to execute in the same core platform, without affecting each other, both spatially and temporally. For that reason, the scheduling is hierarchical. Partitions are activated on a fixed cyclic basis (cyclic-executive scheduling) and whenever a partition has access to the processor, the tasks assigned to that partition are scheduled according to preemptive fixed priorities.

Although industrial solutions such as ARINC-653 exist, most of them were initially intended for single core platforms and must now be extended to multicore.

5. THE THEORETICAL MC MODEL AND ITS COMMON MISCONCEPTIONS

In the previous sections we have presented the main design principles and requirements that drive the development of industrial MCS. We now move the focus of the paper to the academic work. We will discuss the state-of-the-art and some common misconceptions.

5.1 The state-of-the-art in academy

These last years, the real-time research community has been extremely active in the domain of MCS. Almost 200 papers treating of the scheduling of MCS have been referenced in [4], and tens of related papers are still published every year. It would therefore be unrealistic to review and analyse here the whole state-of-the-art on real-time scheduling of MCS. Instead, this section evaluates the key concepts and approaches commonly encountered in real-time scheduling models and algorithms against the recommendations and requirements found in the safety-related industrial standards that were presented in the previous sections.

Most of the works about MCS published by the real-time scheduling research community are based on a model proposed by Vestal in [20]. This model assumes that the system has several modes of execution, say modes $1, 2, \ldots, L$. The application system is a set of real-time tasks, where each task τ_i is characterized by a period and a deadline (as in the usual real-time task model), an assurance level ℓ_i and a set of worst-case computational estimates $\{C_{i,1}, C_{i,2}, \ldots, C_{i,\ell_i}\}$, under the assumption that $C_{i,1} \leq C_{i,2} \leq \ldots \leq C_{i,\ell_i}$. The

different WCET estimates are meant to model estimations of the WCET at different assurance levels. The worst time observed during tests of normal operational scenarios might be used as $C_{i,1}$ whereas at each higher assurance level the subsequent estimates $C_{i,2}, \ldots, C_{i,\ell_i}$ are assumed to be obtained by more conservative WCET analysis techniques. The system starts its execution in mode 1 and all the tasks are scheduled to execute on the core[s]. Then at runtime, if the system is running in mode k then each time the execution budget $C_{i,k}$ of a task τ_i is overshot, the system switches to mode $k+1$. It results from this transition from mode k to mode $k+1$ that all the tasks of criticality not greater than k (i.e., $\ell_i \geq k$) are suspended. Mechanisms have also been proposed to eventually re-activate the dropped tasks at some later points in time [19].

It must be noted that one of the derivates/simplifications of this model is the Vestal's model with only two modes, usually referred to as LO and Hi modes (which stand for Low- and High-criticality modes). Multiple variations of that scheduling scheme exist (please refer to [4] for a comprehensive survey); some for single-core, others for multicore architectures. In the case of multicore, both global and partitioned scheduling techniques have been studied. Solutions for fixed priority scheduling, earliest deadline first and time-triggered scheduling have been proposed. Note that some works also propose to change the priorities or the periods of the tasks during a mode change rather than simply stopping the less critical ones.

5.2 The misalignment of terminology

There is a clear mismatch of interpretation of the concept of "system criticality" between the industrial standards and the academic papers based on the Vestal model [20]. Those academic papers use the terminology "system criticality" to refer to modes of execution of software tasks (e.g., high or low criticality). That is, switching from a mode k to a mode $k+1$ is usually referred to as an "increase of the system criticality level". Although this concept is not fundamentally wrong, it creates confusion in the context of industrial MCS, where the term "system criticality" is used to refer to the level of assurance (DAL or SIL or ASIL) applied in the development of a software application that implements critical system functions, i.e., safety functions.

As a second point, in the standards the word "function" is used at the system level, in reference to a system functionality, or in other words, an action that the system must be able to perform (accelerate, break, etc.). A "function" as defined in the standard may thus involve the whole chain of software and hardware components that play a role in the execution of that action. It may include sensors, processing elements, and actuators. Thus, the word "function" cannot be interpreted as a pure software function, like a C function for instance (or a real-time task). This implies the assignment of a SIL to the whole functionality and not only the individual software functions that are part of it. This is explicitly written in the following note associated to item 3.5.10 of IEC61508-4 [16]:

"SIL characterises the overall safety function, but not any of the distinct subsystems or elements that support that safety function. In common with any element, software therefore has no SIL in its own right. However, it is convenient to talk about "SIL N software" meaning "software in which confidence is justified (expressed on a scale of 1 to 4) that the (software) element safety function will not fail due to relevant systematic failure mechanisms when the (software) element is applied in accordance with the instructions specified in the compliant item safety manual for the element"."

This misunderstanding in the definition of a SIL in the MCS model has no major consequences but somehow it has misguided researchers to think that a real-time task of higher SIL is "more important" than a task of lower SIL (we further discuss that point in the next section). In real-life these tasks are part of one or several system functionalities and those functionalities are the entities which are assigned a SIL. Therefore, a real-time task must be implemented in accordance with the development rules defined for the SIL of the functionality to which it belongs. If the task belongs to more than one functionality then it must naturally be implemented in accordance with the development rules defined for the highest SIL among the SILs of all the functionality to which it belongs. Once implemented, the SIL of the functionality to which the function belongs will also impact the way the function will be deployed on the hardware architecture and it will define or restrict its interaction with other functions.

Conclusion: Some misalignments exist in the interpretation of some key concepts used in the MCS scientific literature and the safety-related industrial standards. We believe that over the years this discrepancy has generated some sort of confusion, which caused the two communities to misunderstand each others' work. In an effort to reconcile the two communities, we warmly invite the academic reader to consult the part 1 of the ISO26262 that clearly defines fundamental concepts that pertain to safety-critical systems, such as "safety functions", "safety-related systems", and "safety integrity level".

5.3 Software task assurance level and the notion of importance

The misalignment between terminologies may lead to misinterpretation of the concepts discussed in the standards. In fact, it caused a major confusion between the notions of criticality and importance of a task (or its "priority" as it sometimes called in the standards). For example, the implementation of a function A that has to be conform to SIL 4 is more costly than the development of the same function in accordance to SIL 3. But by no means this implies that a function A implemented at SIL 4 is more important than any other function B implemented in accordance to SIL 3. Irrespective of their SIL, *all these functions are safety-critical* and may cause severe damage in case of failure. Therefore, a function that is part of a high-SIL functionality cannot be considered as more important than a software function that is part of a functionality of lower SIL. For example, consider the process for ASIL determination described in Fig. 2 and consider two tasks with failure conditions FC1 and FC2, both with severity class S3 and controllability class C3, but with different probabilities of occurrence. FC1 may be assigned an ASIL A and FC2 an ASIL D. In this case it becomes straightforward to understand that one task is not more important than the other as they both lead to failure conditions with the same severity class. Hence, the criticality of each task is not assigned just as a function of its importance for the system, the severity of its failure or the capacity of the system to recover from its loss. It is instead the result of an analysis combining all those different factors (e.g. using a FMECA) that defines clear rules to be respected during the development process.

Conclusion: Different SILs do not mean different importance. Tasks of different SILs are simply subject to different development requirements but the isolation and independence between them still has to be preserved and guaranteed to ensure safety.

5.4 Assigning different WCET estimates

Different SIL imposes different development rules, including coding rules. Although the consequences of timing violations could potentially be less severe for tasks of applications of lower SIL (yet, not always as discussed in Section 5.3), there is no recommendations in the safety-related standards that advocate the use of specific WCET estimation techniques. The standards simply recommends more rigorous testing methods for higher SILs to demonstrate the timing performance of the safety mechanisms at the system level (see Table 11 of ISO26262-4). In contrast, the paper from Vestal [20] and its derivatives assume that the higher the degree of assurance of a task, the more pessimistic the estimation of its WCET[4].

Although this strategy for determining the WCET is valid in the conjecture of Vestal's paper, the most important aspect from the safety point of view that needs to be considered is that the accurate determination of the WCET upper-bound is a necessary but not sufficient condition to ensure the safety of the overall system. In addition to that, mechanisms must be implemented to handle a task overshooting its execution budget without impacting on the system safety.

Exceeding the allocated budget is an obvious failure condition identified during the FMECA. This failure condition can lead to a system failure that can potentially trigger a system hazard. Therefore, more important than accurate estimations of the WCET is the design of mechanisms to ensure that the system safety is not compromised in case of an excessive use of processor resources. In the next section we discuss some of the techniques proposed in the literature for handling those budget violations.

Conclusion: A timing violation is a potential fault which is always there in a RT system. Therefore, during the design of MCS, not only the timing behavior of the tasks must be determined but also reliable mechanisms must be designed to avoid failure propagation caused by timing violations. To do so, standards usually recommend to enforce timing isolation by using partitions.

5.5 Graceful degradation

The plethora of work based on Vestal's model could be understood by its resemblance with the graceful degradation technique described in the IEC61508 (see Section 4.1.4). However, there is a major issue that makes this scheduling model difficult to justify in the context of MCS. The low and high criticality tasks are clearly not isolated in the time domain since the timing properties (e.g., the response time) of the high criticality tasks depend on the scheduling decisions of low criticality tasks (and not only on their actual execution time). According to the safety and criticality assessment methodology described in Section 2 and to the independence requirement discussed in Section 4.1, it would be very difficult to demonstrate to the safety authority that under all foreseen operational conditions the safety properties (greatly influenced by the timing properties) of the higher and lower criticality systems would not be compromised at some point in time.

Conclusion: Graceful degradation is a technique that lends itself very nicely for the scheduling of MCS. However, suspending a lower criticality task in benefit of a higher criticality task raises several questions in terms of isolation between applications, i.e. there is a high risk that the certification authority will deem that not enough isolation between the safety functions has been achieved. Therefore, in the context of MCS it is recommended that only *non-critical* tasks be actually suspended in the event of a critical fault (e.g., budget overshooting).

5.6 Assignment of a software failure rate

Recent research on the mixed-criticality scheduling theory have introduced the concept of probabilistic WCET [5], which can be understood as the "probability of violating a timing requirement". In this model, from a mathematical point of view the WCET of a task is no longer a single rigid value. Rather, the model provides a threshold, i.e., an upper-bound on the execution time, that has a given probability of being exceeded at runtime. Informally, the rational behind the introduction of probabilities in the model is the assumption that if the probability of exceeding that threshold is shown to be smaller than the probability of experiencing an irreversible failure (like an irreversible hardware failure for instance), then that threshold can be used as a "safe" estimate of the WCET. Recent papers present various techniques to derive such probabilistic WCET [1]. In those papers the target probabilities are either taken directly from the probabilities of failure of a safety function allocated to the E/E/PE safety-related system (like in [5]), e.g., probability of failure of 10^{-9} per hour for a SIL 4 function as defined in Table 3 of IEC61508-1 (or in the FAA Advisory Circular AC-25-1309), or for the same reasons they are set to even lower values [1].

These probabilistic techniques aim at building a reliability model of the software, according to which confidence can be placed in the expected timing behavior of the application and in particular in its worst-case responsiveness. In broad terms, *software* reliability is the property of the software being "free from faults". Failures caused by software can degrade the system performance, up to the complete loss of the system or potential loss of life or major damage to the environment. According to this definition, exceeding a pre-allocated execution budget can indeed be seen as a software fault as it may bring about the same consequences. There are, however, three important points that we would like to discuss in this paper.

First, although the rationale behind this probabilistic model is well motivated and fully justified (from the community point of view), in most of recent research on probabilistic timing estimates the community has shown a particularly high confidence in the applicability of their model to real systems. In some of those papers, sometimes it seems as granted that this probabilistic model will soon be used in MC system V&V processes. However, it must be highlighted that at the time of writing this paper, although numerous consolidated reliability models have been developed to *quantitatively* assess the compliance of the *hardware* components to the reliability requirements [18], *software* reliability mod-

[4]Although this assumption is perfectly relevant (since a more pessimistic estimate may be understood as more reliable), it does not equate to the recommendations of the standards. More rigorous testing could also be understood as *less* pessimistic estimates (according to the usual trade-off between accuracy and runtime complexity of most of computation techniques).

els are still under debate within industry, academia and international standards community. Those models rely on a number of assumptions that have proven not to be fully justified in the vast majority of bespoke software and currently in the industry, confidence cannot be placed in such models to assess the reliability of the software parts. This is clearly and explicitly stated, for instance, in subsection 4.1.2 of [10] or section 12.3.3 of the DO-178C:

"Many methods for predicting software reliability based on developmental metrics have been published, for example, software structure, defect detection rate, etc. This document does not provide guidance for those types of methods, because at the time of writing [2011], currently available methods did not provide results in which confidence can be placed." – section 12.3.3, page 89 of DO-178C [7].

Second, these numbers (e.g. 10^{-5}, 10^{-9}, etc.) have been defined at the system level as function of failure rates and to the best of our knowledge, there is no paper discussing why those specific numbers could be applied to the software parts of the system. Furthermore, last but not least, until now the safety and dependability assessment of safety-critical *software* has always been performed through a set of *qualitative* processes that are applied during all phases of the software development life cycle. To the best of our knowledge, there is currently no evidence indicating that the process of assessing *software* safety is about to change from a *qualitative* process to a *quantitative* process. Unlike the typical process applied to develop the hardware, the development of a software to a certain assurance level does not imply the assignment of a failure rate for that software. In other words, there is no evidence that those specific numbers applied to function failure rates will ever apply in the assessment of software safety.

Conclusion: At this date, even though the computation of probabilistic estimates to MCS constitutes an important research direction that aims at improving the WCET estimation of real-time tasks, it cannot be taken as granted that those estimates will ever be used in industrial systems to prove the safety of a software function.

6. CONCLUSION

In this paper, we summarised and presented relevant information and basic concepts available in some of the most-cited safety-related standards in the real-time system research community, for the understanding of the mixed-criticality (MC) concept from an industrial perspective. Then, we (briefly) evaluated some aspects of the currently-accepted state-of-the-art MCS model against the safety requirements established by those standards, namely, IEC61508, ISO26262, and DO-178C. Our evaluation led to the conclusion that there is today a clear gap between some of the guidelines provided in those standards and their interpretation by the academic community. We believe that researchers should further focus on building evidences that (1) the MCS model itself is motivated, justified, and in accordance with the requirements and (2) the current MC scheduling techniques comply with those safety requirements, in particular regarding the notions of isolation and independence of execution.

Acknowledgements. This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and when applicable, co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); also by, FCT/MEC and the EU ARTEMIS JU within projects ARTEMIS/0003/2012 - JU grant nr. 333053 (CONCERTO) and ARTEMIS/0001/2013 - JU grant nr. 621429(EMC2); and also by the Portuguese National Innovation Agency (ANI) under the ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project V-SIS, QREN - SI I&DT nr. 38923.

7. REFERENCES

[1] J. Abella, D. Hardy, I. Puaut, E. Quinones, and F. Cazorla. On the comparison of deterministic and probabilistic wcet estimation techniques. In *ECRTS 2014*, pages 266–275, July 2014.

[2] ARINC-653. *Avionics Application Software Standard Interface*. ARINC, Inc., 2003.

[3] ARINC 700 series. Arinc. http://store.aviation-ia.com/cf/store/catalog.cfm?prod_group_id=1&category_group_id=4, June 2015.

[4] A. Burns and R. Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

[5] R. Davis, T. Vardanega, J. Alexanderson, V. Francis, P. Mark, B. Ian, A.-A. Mikel, F. Wartel, L. Cucu-Grosjean, P. Mathieu, F. Glenn, and F. J. Cazorla. PROXIMA: A Probabilistic Approach to the Timing Behaviour of Mixed-Criticality Systems. *Ada User Journal*, (2):118–122, 2014.

[6] N. Diniz and J. Rufino. Arinc 653 in space dasia 2005, eurospace, edinburgh, scotland. 2005.

[7] DO-178C. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc., 2011.

[8] ECSS-Q-30-02A. *Failure modes, effects (and criticality) analysis (FMEA/FMECA)*. European Cooperation for Space Standardization, 2009.

[9] ECSS-Q-40-12A. *Fault tree analysis - Adoption notice ECSS/IEC 61025*. European Cooperation for Space Standardization, 2008.

[10] ECSS-Q-HB-80-03A. *Space Product Assurance - Software Dependability and Safety*. European Cooperation for Space Standardization, 2009.

[11] ECSS-Q-ST-30C. *Space product assurance - Dependability*. European Cooperation for Space Standardization, 2009.

[12] ECSS-Q-ST-80C. *Software Product Assurance*. European Cooperation for Space Standardization, 2009.

[13] EN 50128. *Railway Applications Communication, Signalling and Processing Systems Software for Railway Control and Protection Systems*. CENELEC, 2009.

[14] IEC60812. *Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA)*. IEC, 2006.

[15] IEC61025. *Fault tree analysis (FTA)*. IEC, 2006.

[16] IEC61508. *Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC, 2010.

[17] ISO26262. *Road vehicles - Functional safety*. ISO, 2011.

[18] RIAC-HDBK-217Plus. *Handbook of 217Plus Reliability Prediction Models*. RIAC, 2006.

[19] F. Santy, G. Raravi, G. Nelissen, V. Nelis, P. Kumar, J. Goossens, and E. Tovar. Two protocols to reduce the criticality level of multiprocessor mixed-criticality systems. In *RTNS 2013*, RTNS '13, pages 183–192. ACM, 2013.

[20] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS 2007*, pages 239–243. IEEE, 2007.

[21] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *DASC'07*, pages 2–A. IEEE, 2007.