Kernel-based ARchitecture for safetY-critical cONtrol

KARYON

FP7-288195

D4.5.1 – Safety Kernel Definition (Public version)

Work Package	WP4		
Due Date	M37	Submission Date	2014-11-28
Main Author(s)	António Casimiro (FFCUL), Eric Vial (FFCUL)		
Contributors	Siavash Aslani (4S), Renato Librino (4S), Rolf Johansson (SP)		
Version	1.1	Status	Final
Dissemination Level	Public	Nature	Report
Keywords	Safety management, Time and Space partitioning, Architecture, Interfaces		
Reviewers			

SEVENTH FRAMEWORK PROGRAMME

Part of the Seventh

Framework Programme

Funded by the EC – DG INFSO

Version history

Rev	Date	Author	Comments
V0.1	2014-09-15	António Casimiro (FCUL)	Initial Structure
V0.2	2014-10-23	Siavash Aslani (4SG)	Added section on Safety Kernel development as a SEooC
V0.3	2014-10-27	Eric Vial (FCUL)	Added section on Safety Kernel implementation and configuration, and section on performance evaluation
V0.4	2014-11-25	António Casimiro (FCUL)	Final draft.
V1.0	2014-11-28	António Casimiro (FCUL)	Final review and submission.
V1.1	2015-02-09	António Casimiro (FCUL)	Necessary update on the public version of the deliverable.

Glossary of Acronyms

AIR	A TSP architecture implementation
KARYON	Kernel-based ARchitecture for safetY-critical cONtrol
LoS	Level of Service
OS	Operating System
MTF	Major Time Frame
PL	Performance Level
PMK	Partition Management Kernel
SM	Safety Manager
SK	Safety Kernel
SOP	State of practice
SOTA	State of the art
SSA	System safety assessment
TFD	Timing Failure Detector
TSP	Time and Space Partitioning
Tx.y	Task belonging to work package x, with serial number y
WP	Work Package
WPx	Work Package with serial number x

Executive Summary

The Safety Kernel is a part of the KARYON architectural pattern, which is devoted to handling the information concerning the integrity of sensor data and concerning the timeliness of some system components, with the objective of determining the best possible operational mode in which the several cooperative functions will be performed safely.

This report provides a final description of the Safety Kernel, focusing on its architecture and interfaces, and providing details on the solutions that were designed to implement its components. Therefore, it extends the preliminary description provided in D4.2, which did not include details on the specific solutions. Furthermore, this report also provides the following new contributions.

First, it provides the results of a study on the requirements for developing the Safety Kernel, by applying ISO 26262 concepts, namely by considering the Safety Kernel as a Safety Element out of Context (SEooC). Given the performed hazard analysis, it is shown, perhaps not surprisingly, that the Safety Kernel must be assigned ASIL D.

Second, the report provides a performance analysis of the central component of the Safety Kernel, the Safety Manager. The question we answer is about the ability of the solution to scale, which we do using the particular implementation that was done in the project. Although the main objective of the implementation was to demonstrate the concepts developed in KARYON, it is good enough to perform the intended analysis and derive relevant conclusions. In particular, we concluded that the approach allows for reaction delays in the millisecond scale, and for the possibility of defining and easily handling hundreds of safety constraints.

Finally, given that this is one of the final project reports, we provide a revised overview of the fundamental concepts and definitions underlying the KARYON approach. To complement this overview, the deliverable includes two papers in which these concepts are elaborated in a more detailed way, discussing their implications and relevance for the achievement of safe and cost-effective cooperative systems, and also explaining why the proposed architectural pattern is generic and thus applicable to different application domains.

Editor note: All marked text (in yellow) corresponds to text that has been left unchanged with respect to the preliminary version of this deliverable (D4.2 – First report on Safety Kernel Definition).

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation, Purpose and Scope

KARYON focuses on the predictable and safe coordination of smart vehicles that autonomously cooperate and interact in an open and inherently uncertain environment. The fundamental challenge that is addressed consists in ensuring functional safety, while achieving at the same time high performance levels in a cost-effective manner. By excluding the use of costly components and solutions (e.g., specialized sensors and hardware, redundant components or sub-systems), which could bring increased robustness and shield the operation from uncertainties affecting timeliness and quality of sensor data, a trade-off between safety and performance emerges. In one hand it is possible to achieve safety by sacrificing performance: the solution will be safe as long as all assumptions are satisfied, which requires explicitly assuming the existing (possibly high) uncertainties in the design, and thus allocating large safety margins that will impact on performance. On the other hand, if an optimistic approach is followed with respect to the assumptions about relevant operational variables (e.g., assuming typical values for communication jitter or typical deviations due to noise affecting sensor measurements, ignoring less probably values that may occur due to uncertainties), the result is that performance will be good, but it will not be possible to achieve the same (high) levels of safety integrity.

KARYON addresses this paradox by proposing a new architectural pattern, which exploits three key concepts:

- First, the architectural pattern exploits the concept of **architectural hybridization**, meaning that it is explicitly assumed that different parts of the system will exhibit different properties with respect to timeliness. This is instrumental to allow dealing with the temporal uncertainties affecting the system operation. In particular, the system components lying within the non-predictable (possibly non-timely) part of the system will not need not be proven timely in design time (e.g., components performing complex computations, components depending on wireless communication), provided that another part of the system always behaves timely (as proven in design time) with the required (possibly very high) probability.
- Second, it exploits the concept of **Level of Service (LoS),** which consists in defining multiple modes of operation and/or system configurations, each with different safety integrity requirements, and each implying the execution of functionalities with different performance levels. In run-time, and depending on the observed health conditions (e.g., sensor data quality and satisfaction of timeliness requirements), it will be possible to select one LoS for which requirements are met, and which provides the best possible performance. This concept thus provides the needed flexibility to achieve high performance without compromise safety.
- Third, and finally, an **abstract sensor model** is considered, to allow abstracting the various failure modes affecting the quality of sensor data. The abstraction allows a separation of concerns between the mechanisms for detecting faults and characterizing their effect, and the solutions for managing the system configuration and operation to meet the needed safety requirements. In essence, it will be possible to specify safety conditions (or safety rules) expressed in terms of a generic data validity metric, rather than in terms of specific metrics reflecting the operational conditions of particular sensors or particular processing components. Such an abstract sensor model allows abstracting faults both in the value and time domains.

The Safety Kernel, on which we focus in this report, lies in the confluence of the above-mentioned concepts and is an integral part of the KARYON architectural pattern, consisting in set of components that are concerned with managing safety-related information. Let us explain why the described concepts are important and influence the design of the Safety Kernel.

From a functional perspective, the set of components that constitute the Safety Kernel are responsible to perform monitoring and management tasks, taking into consideration safety rules defined in design time and integrity-related information collected in run-time. The role of the Safety Kernel is to determine the highest Level of Service in which every functionality will be safely performed, providing indications on the needed operation modes and triggering configuration changes that will enforce those Levels of Service. Ultimately, the goal of the Safety Kernel is thus to ensure that the required functional safety goals are met in run-time.

Given this goal, the integrity requirements allocated to the Safety Kernel subsystem must be as high as the highest integrity requirements allocated to any other part or components of the system. Therefore, considering the implications of applying the architectural hybridization concept, it must reside in the predictable part of the system, so that it may be proven timely at design time. Additionally, it has to be as simple and confined as possible for increased robustness.

By abstracting sensor faults and characterizing the quality of sensor data through a generic validity attribute, the design and the task of the Safety Kernel can be simplified. Safety rules can be uniformly expressed in terms of validity requirements, facilitating their internal representation and evaluating run-time evaluation. In addition, interfaces for collecting integrity information can also be made generic, as this information is almost exclusively expressed in terms of validity. An overall benefit is that the Safety Kernel can be defined as a generic subsystem, which is completely independent from the application domain and from the concrete functionalities provided by the system in which it will be used.

One fundamental issue in KARYON is that it deals with cooperative systems and hence with the execution of functionalities in a cooperative manner. This implies that the Level of Service concept is extended to a set of vehicles, that is, all vehicles execute the functionality with the same LoS. Furthermore, the implication on the KARYON pattern, and on the Safety Kernel definition in particular, is that they must be applicable in a distributed system, with nodes interconnected by a possibly unreliable network. The pattern is applicable to a KARYON node, and in each KARYON node there will be one Safety Kernel. A cooperative systems is formed by several KARYON nodes, which means that in each vehicle there is always at least one Safety Kernel. Within a vehicle it is possible to consider either centralised and distributed implementations of the Safety Kernel, provided that the needed properties are satisfied. Whichever the case, we describe the Safety Kernel as a logically unique subsystem, assuming that all distribution aspects will be hidden in the implementation.

With the aim of clarifying the context in which the Safety Kernel has to operate, and the tasks it has to perform, we briefly characterize a KARYON system, according to the KARYON architectural pattern.

A cooperative functionality is realized by a set of cooperating vehicles and in each vehicle the functionality is decomposed in a number of functions provided by functional components. Each of these functions can, in fact, be used in the provision of several functionalities. For instance, a function that calculates the front distance can be used in the provisioning of a Platooning functionality and, at the same time, to implement a cooperative warning functionality. Moreover, a function can either be implemented:

a) As a single component that executes the function with a single performance level (a single operating mode);

b) As a single component admitting multiple operation modes, possibly executing different selectable algorithms or a single algorithm with different parameterizations, each leading to a different performance level;

c) By multiple components, executing independently and redundantly, where some components provide a better performance level than others due to being implemented differently, possibly with resorting to different approaches, algorithms, data inputs, etc.

Components can be of different categories, depending on their purpose. Sensor components interact with the physical environment and produce information to the system. Actuator components receive information from the system and interact with the physical environment. Computing components are within the system and may receive and produce information from/to other components. Finally, communication components are special in the sense that they can receive and produce information from/to the system, but can also interact with the physical environment. The data produced by components, in particular sensor data (but also other computed data), can have an attached data validity attribute, which is provided by the component on its output. Each data value can thus have a validity attribute.

Depending on the specific operation mode (or performance level) of each function (performed by a component), or on the components that are selected for realizing some function, the overall functionality (which is achieved by the combined use of several components offering different functions) will be performed with different Levels of Service. The overall goal of the Safety Kernel is precisely to manage the combination of operation modes or the selected components, which is necessary to enforce the required LoS of each functionality and ensure that functional safety requirements are satisfied. These requirements have to be established in design time, resulting from the safety analysis that has to be performed for each functionality, which dictates safety requirements that are allocated to each functional component.

In the design of the Safety Kernel, several issues need to be considered, of which we highlight the following:

- It will have to deal with timing failures of some functional components, namely complex components whose timely behavior cannot be guaranteed (with the desired probability) in design time;
- It will have to deal with the fact that the validity of sensor data may not be guaranteed at design time and thus might go beyond some threshold required for the operation within a certain LoS;
- It will determine the best possible (leading to better performance) operation mode and/or configuration based on the observed timeliness of complex components and on the observed data validity, and considering predefined safety rules associated to each LoS of each functionality.

In this report we describe the solutions that were defined in KARYON to address these issues and to achieve a Safety Kernel providing the intended functionality and exhibiting the required properties.

1.2 Relation to other work

Within KARYON, the work presented in this deliverable is closely related to the general architecture defined in WP2 and to the work done in Task 4.1 concerning safety requirements and constraints. On the other hand, we report on a concrete implementation of the Safety Kernel, which was done and evaluated in the scope of WP4, and applied in the context of WP5. More specifically, the Safety Kernel implementation (including software and hardware) was integrated with the miniature cars of the Gulliver test-bed, to be used in the automotive demonstration, and was also integrated (only the software part) with the avionics simulation.

Concerning related work available in the literature, we briefly review in the following paragraphs the state of the art in related areas and the state of practice in the industrial community.

Safety critical systems are typically built considering models in which assumed properties (e.g., synchrony, faults) are applied to the whole system and do not change over time. Therefore, these models are said to be homogeneous. On the contrary, we advocate that in order achieve performance improvements without sacrificing safety it is necessary to consider hybrid distributed system models [7]. These allow to better capture the real properties of the environments in which vehicles operate and in which functionality is implemented. More than that, we believe that architectural hybridization [8] is the natural way to architect systems in accordance to the considered hybrid system models. One simple example of a system well described by a hybrid system model is a system with a watchdog. The watchdog is used as a safeguard to make sure that if something goes wrong in the system then it will be possible to, at least, make the system stop in order to prevent some wrong or unsafe behaviour. Clearly, while the system is assumed to possibly fail, the watchdog is assumed to always operate correctly. Therefore, the watchdog is a subsystem with better properties than the rest of the system, which is possible because it is a simple component.

Mixed criticality [6] is the concept of allowing applications with different levels of criticality to coexist on the same system. In this case, one may want that the properties and assumptions that hold for one application be different from the ones that hold for another application, which is not easily achieved in a system based on a homogeneous model. Mixed criticality models show affinity with hybrid system models, in which assumptions and properties may vary on different parts of the system or may hold only for a period of time.

The GENESYS project [9] acknowledged the hybrid nature of systems and developed a component-based generic platform for embedded real-time system. However, GENESYS is significantly focused on the problems related to composition and component interfaces, whereas our interest is on understanding how uncertainty can be characterized and how the performance can be managed while making sure that safety requirements are always satisfied.

The recovery block concept [10] follows a hybrid model, where multiple versions for the same function are developed. First it runs the more complex version of the function (with extra features and more prone to errors). If an error is detected, then a simpler implementation is executed. Simplex [11] follows a similar approach by defining an architecture composed of two system controllers: one simple and proven safe, and one with additional features, but unreliable. It tolerates faults in the unreliable controller using a decision module that observes the plant to verify if the controller is being able to keep the controlled system within the desired operational envelope. If not, it switches the execution to the reliable controller, trading off performance for safety.

The solution is thus designed by assuming that faults are ultimately reflected on some undesired external behaviour, which can be reliably observed through the existing sensors. In KARYON we look to the problem differently, because we consider that sensor data may not always be valid due to faults affecting sensors, or due to uncertainties affecting the timeliness of communication and hence the promptness (and validity) of the other sensor data received from remote vehicles. Therefore, we define an abstract sensor model that allows the validity of sensor data to be estimated, and we consider that some components may do timing failures due to their complexity. Given that, the solutions for deciding when to change the control algorithm, or when to perform some system reconfiguration, are done in a different way than it is done in Simplex.

The coexistence of reliable and unreliable components calls for mechanisms for fault containment. Virtualization [12] has been widely used as a mechanism to run multiple systems within the same physical computing platform, allowing providing different environments in each virtual machine and isolation between them. However, most virtualization solutions do not provide strict temporal isolation. One approach to achieve mixed criticality without increased certification expense and providing a complete fault containment (including temporal isolation) between components is to use time and space partitioning (TSP) [1, 2]. TSP is a concept for safety-critical systems in which applications with different criticality levels and different requirements may coexist in the same execution platform. TSP separates the system's software components into logical containers called partitions, ensuring that faults occurring in one partition do not affect other partitions, with respect to both time and space domains. These two properties ensure that faults are contained to their domain of occurrence, preventing them from propagating to other partitions.

A prominent example of TSP system design is the adoption of the ARINC 653 [4] specification by the civil aviation domain. In the automotive industry, the top-level requirements for an AUTOSAR operating system include provisions that correspond, to some extent, to the notions of temporal and spatial isolation [8]. The specification of the AUTOSAR operating system, however, does not prescribe the use of strict partitioned scheduling as a means to achieve this temporal isolation among applications [13].

We take advantage of TSP properties to develop a solution that integrates in the same platform components of different complexity, some that are proven timely and reliable in design time, and other that may behave in uncertain ways. The latter can be used to implement improved functions, exploiting the additional information made available through cooperation, without compromising safety. The overall approach can still be viewed as sufficiently modular to be adopted by existing legacy systems.

1.3 Structure of the document

In the next section (Section 2) we provide a revised and summarized description of the main concepts and definitions considered in the KARYON project, which have been firstly provided in deliverable D2.3. The section is complemented with two papers, provided in Annex A and Annex B, which further detail and discuss these concepts and their application.

Then, Section 3 covers the issues in the design of the Safety Kernel, from tracking and assessing the safety requirements to the adaptation of the LoS. An extended version of this section is available in the preliminary version of this deliverable (D4.2 – First report on the Safety Kernel definition), from which the more relevant parts were taken.

Section 4 presents the final architecture of the Safety Kernel, with small revisions with respect to the description provided in the preliminary version. The section begins by stating the relation between the Safety Kernel and the overall KARYON architecture. Then each component of the Safety Kernel is presented together with its interfaces. Also, to guarantee the timely information flow between the components, the required scheduler support is discussed.

Section 5 provides an analysis of the requirements on the Safety Kernel development by considering the Safety Kernel as a SEooC (Safety Element out of Context), as defined in ISO 26262.

Then, Section 6 provides a description of the design and implementation of the several Safety Kernel components. In addition, the section also explains how the Safety Kernel can be configured. The final part of the section includes a use case to exemplify, in concrete terms, how to configure the Safety Kernel for operation in a system that executes two cooperative functionalities.

Section 7 is devoted to performance analysis and provides evaluation results concerning the implemented Safety Kernel. The analysis is meant to assess the ability of the Safety Kernel to scale, as necessary to deal with complex systems involving many functionalities and safety rules. The evaluation, on the other hand, provides concrete performance results that allow assessing the possible reach and limitations of the solution when used in real applications.

Finally, Section 8 concludes the report.

2. Fundamental concepts and definitions

In this section we provide a brief and revised overview of the main concepts, definitions and terminology employed in the KARYON project, which are relevant, in particular, to make the description of the Safety Kernel clearer. We explain how cooperation and cooperative systems are understood in the context of KARYON, we review the concept of data validity and its implications on the KARYON architectural pattern and we revisit the concept of Level of Service and explain why it is so important in KARYON. Finally, we consider functional safety aspects in the scope of cooperative systems and we explain why the proposed architectural pattern is generic and thus applicable to various domains. In both cases, we provide two papers as annexes to the deliverable, which include detailed discussions.

2.1 Cooperation

In KARYON we focus on cooperative systems or vehicles, like automobiles, robots, airplanes or Remote Piloted Vehicles (RPVs). We understand by **cooperation** that systems actively help each other in order to achieve some common goal, or to realize some cooperative functionality. Cooperation can be used to improve coordination among vehicles. **Coordination** is more general in the sense that it can take place by following pre-established rules dictated beforehand and embedded in local control rules. Vehicles can thus coordinate in the traffic by following some well-known rules, without the need to explicitly communicate. Cooperation can take place when vehicles are able to communicate with each other. This may allow improving the traffic flow, reducing energy consumption and satisfying safety requirements more easily.

Cooperation entails a number of aspects, of which communication is perhaps the most prominent.

2.1.1 Communication

Since the ability to communicate is absolutely required to achieve cooperation, the well-known uncertainties affecting wireless communication become a serious issue. These uncertainties are due to mobility and to interferences affecting the signal quality, and their effects range from occasional omissions to the impossibility of transmitting any data for long periods of time. Given that the considered cooperative systems must satisfy (critical) functional safety requirements, it is necessary to deal with these uncertainties.

Our basic approach in this respect is to accept the fact that communication might not always be possible, devising solutions allowing vehicles to safely switch from cooperative to non-cooperative modes of operation when communication is not possible, and switch back to cooperative modes when communication is regained. We thus focus on the dynamic aspects associated to communication with varying levels of quality, and on ensuring safety requirements despite such dynamics. Ensuring safety of autonomous modes of operation falls out of the scope of the project.

We note that communication can also take place between vehicles and entities in fixed positions, like road-side units or air traffic management sites. Therefore, cooperation might also take place even if not based on direct vehicle to vehicle communication.

Finally, we also note that some non-conventional forms of communication could be used to achieve cooperation. For instance, it would be possible to devise sound-based or light-based communication means, which could be used to send and receive information as an alternative to using radio-based communication networks. However, in KARYON we are only considering the typical communication networks used in vehicular scenarios, such as 802.11, 802.15.4 or ADS-B.

2.1.2 Cooperation scope

Cooperation takes place between a set of cooperating entities. Therefore, one fundamental issue concerns the definition of the entities that are included in this set. In other words, it is necessary to define the **scope** in which some cooperative functionality is realised.

Given that the considered entities (cars, airplanes) are mobile, and that it is usually only relevant to cooperate with geographically near entities, it is not possible to define a fixed scope that will hold for the lifetime of some functionality. On the contrary, the scope is varying over time. And the dynamic characteristics of this variation may constitute a limiting factor on the possibility of achieving performance improvements through cooperation.

In KARYON we do not deal with the specific problem of defining the scope of cooperation, that is, we do not propose particular solutions for this problem. Instead, we assume that when some cooperative function will need to know the cooperation scope, this information will be provided by some functional component that will be included as part of the application.

We note that the problem is not trivial. In fact, it is made difficult by the fact that we are considering a distributed system that is essentially asynchronous (in the sense that no strict bounds can be defined for the communication latency), subject to failures (because communication may not always be reliable), and dynamic, due to the need to consider joining and leaving vehicles. Existing solutions for agreement in asynchronous distributed systems can be helpful to address the issue of defining a cooperation scope. In the automotive or avionic scenarios it may also be possibly to adopt practical solutions like considering that there exists a central entity within the infrastructure, which provides scope information to all vehicles that are in a certain vicinity of this entity.

2.2 Data validity

The considered autonomous vehicles rely on sensor data for decision-making. This data is collected from local sensors, but might also be received from external entities, like other vehicles, road-side units or ground control centres. Assuring the quality of this data is hence very important for safety reasons. When it is not possible to know, a priori, how good will be the data quality, as it happens when the operational environments are not fully controlled, when sensors can be affected by a varied set of faults or when information is received through communication links with loosely defined timeliness characteristics, it becomes necessary to characterize the varying data quality in run-time.

In KARYON we look at this problem with particular attention. We generalize the problem of failures affecting sensor data quality and define an **abstract sensor model** for that purpose. Abstract sensors provide at their interface two values: the actual sensor data and a corresponding **data validity** value, which characterizes the confidence on the provided data. The notion is reflected both in the defined KARYON architectural pattern, as well as in the way the safety reasoning in developed. The abstract sensor model is introduced and described in detail in deliverable D2.5.

2.3 Level of Service

Achieving the optimal balance between performance improvements and the needed functional safety, implies that ability to dynamically change the way in which cooperative functions are realised, sacrificing performance when this is the only way for being safe. The concept of Level of Service (LoS) is introduced in this context and must be clearly explained. It is also important to explain the exact meaning of “performance”.

In vehicular cooperative systems, in which vehicles are moving in a physical shared space and perform a number of possible manoeuvres, we can intuitively characterize how well these manoeuvres are executed in terms of a number of metrics like the speed of execution, the smoothness of the movement or the distance between the vehicles. All of these are important traffic flow metrics, also allowing to evaluate how well the shared space is used and to reason in terms of energy (fuel consumption) costs. Other metrics could as well be considered, like the passenger comfort during the execution of the manoeuvres, which may be not so important from an economic perspective, but will surely be important for the acceptance of involved technologies. All of these are **performance** metrics, and the objective in KARYON is to allow these metrics to be improved.

It is clear that the control algorithms employed in the execution of the cooperative functions must care about performance. At design time, the objective is to define control algorithms that will allow vehicles to move faster, closer, smoothly and following straight trajectories. These algorithms will also consider the operational contexts, including road conditions, weather conditions, traffic rules, etc. The resulting control system is what we call the **nominal control system**, which performs the intended functions. Quite clearly, it is also necessary to ensure that the resulting system will perform the functions safely.

When designing the nominal control system, the designer will thus have to make a number of assumptions about the physical processes, context, and so on. And the function may eventually be proven safe, provided that the assumptions hold in reality. So far we have just talked about assumptions that are strictly related to the **application semantics,** which are outside the KARYON scope. However**,** there are also assumptions that need to be made concerning the control system itself, like fault and synchrony assumptions.

In KARYON we consider that different **operational modes** (or **performance levels**) for each component can be defined, each relying on its own set of assumptions. A functionality requires possibly several components for being realised, and can thus be performed with different **Levels of Service (LoS),** depending on the combination of components being used, or on their operational mode.

To ensure that a functionality is performed safely, all assumptions must be satisfied. And given that a functionality can be provided with different Levels of Service, different sets of assumptions have to be satisfied for each LoS. Therefore, depending on the system health (which faults might be affecting sensor data, how good is communication, how timely is the execution), which dictates which assumptions are satisfied at a given moment, the LoS under which the function can be safely provided may be chosen.

From a safety perspective, it must be shown in design time that the functionality will always be safe for all the possible configurations (all LoS) under the assumptions considered in the design of components used in each of these configurations. One additional assumption has necessarily to be considered, which must be proven to hold in design time with the highest probability. This is an assumption on the time that it takes to switch from any higher LoS configuration, to the baseline LoS configuration. This time will have to be known and taken into account in the design of the control functions, in order to ensure that the functionality will always be safe.

One final clarification is needed, to explain how the concept of Level of Service is understood in a cooperative context. It should be clear, from the previous discussion, that each LoS is associated with a certain configuration of the nominal control system. But since in a cooperative scenario there are multiple vehicles, then there is a question of what can be expected concerning the consistency between all vehicles' configurations.

From a performance perspective, it is better if the LoS, whatever it may be, is consistent across the cooperating vehicles, and if this consistency can be assumed in the design of the control algorithms. In fact, this allows restricting the possible heterogeneity between control decisions, because each vehicle is aware of the limits under which the cooperative function is being executed, which are the same limits imposed in each vehicle. On the other hand, if such consistency is not ensured, then this will be reflected in the control algorithm, which will have to embed larger safety margins to encompass for the potential (and unknown) discrepancy between the control decisions taken in each vehicle. This will have a negative impact on the performance achievable in each LoS, but it will not be an impediment for achieving cooperative solutions. In KARYON we consider that the LoS is consistent among cooperative vehicles. It is thus a **cooperative LoS**. Given that each vehicle as a local perception of the best possible LoS under which it can execute a functionality, we refer to this as the **local LoS**.

Ensuring a consistent LoS in the execution of the cooperative functionality may seem infeasible, because this requires some form of agreement, which may not always be achievable in the asynchronous communication environments that we consider. However, it must be noted that when communication is not possible between all the vehicles in the cooperation scope, then all the vehicles will end up executing in the baseline LoS, thus becoming consistent. On the other hand, if communication is possible, then it will also be possible to run some distributed algorithm to reach consensus on the cooperative LoS. The only issue is that the switching between two LoS may not be done precisely at the same time in all vehicles, and hence an inconsistency interval will have to be considered in the design of the control algorithms.

2.4 Functional Safety for Cooperative Systems

One of the application areas where the KARYON architectural pattern shows to be very valuable is for cooperative vehicles. In Annex A is investigated the problem of identifying and distributing the safety requirements among the vehicles in a road train (platoon). The problem KARYON addresses is how to achieve safe systems even though some components cannot be shown to have very high safety integrity in worst case. The two generic candidates for this problem are sensors and communication links. Depending on how a cooperation is defined, the safety implications on the communication will be different. When investigating road trains, this represents a very high degree of interdependence between the cooperative parties, which implies that the communication link becomes a critical part of the cooperative system.

In Annex A is shown that it is important to take the entire cooperative system as the scope, instead of looking at the vehicles and the communication links separately, when addressing the safety problem. This is very much in line with the general KARYON pattern where the concept of Levels of Service is well suited for cooperative functions like road trains. Furthermore this cooperative perspective is an enabler for identifying the most efficient means for redundancy in the communication balancing the redundancy concept in the respective vehicles. In Annex A this is further elaborated in a jointly performed effort with the FP7 project SARTRE on road trains.

In Annex A: reprint of the paper "**Functional Safety for Cooperative Systems**". Josef Nilsson, Carl Bergenhem, Jan Jacobson, Rolf Johansson and Jonny Vinter. In proceedings of SAE International Congress 2013.

2.5 Deployment of the safety architecture pattern

When deploying the KARYON architecture pattern it is important that it can be aligned with existing patterns and standards. This is especially true for complex products like in avionics and in automotive, where there are complex structures of suppliers and sub-suppliers involved. This is why it is important to show that the general KARYON architectural pattern can be instantiated according to AUTOSAR (automotive domain) and Integrated Modular Avionics, IMA (avionic domain). In both cases these standards address the issue of getting an integrated rather than a federated E/E architecture. This means that the functional application software components can be rather freely distributed over the different hardware resources. This is enabled by the specification of the interface between application and platform, and the set of services that the platform should guarantee.

When deploying the KARYON pattern to these domains it implies some extensions of the specifications of ARINC 653 and of AUTOSAR respectively. The details of these extensions are further elaborated in Annex B. Generally speaking, one implication is that for both domains an extension is needed specifying a dedicated Safety Manager in the platform. Furthermore, each software component signal is required to be extended by some safety integrity level attribute, complementing the nominal value. For the implication of the change of level of service, the existing means of mode management might be sufficient. The conclusion is that the KARYON architectural pattern is well suited to be adopted in the avionics and automotive domains, respectively.

In Annex B: reprint of the paper "**An Architecture Pattern enabling safety at Lower Cost and with Higher Performance**". Rolf Johansson, Jörg Kaiser, António Casimiro, Renato Librino, Kenneth Östberg, José Rufino, and Pedro Costa. In proceeding of Embedded Real Time Software and Systems (ERTS2) 2014.

3. Issues in the design of the Safety Kernel

There are several key issues that have to be considered in the design of the Safety Kernel. These issues are the focus of this section and are organized under three headings that correspond to the three stages of the Safety Kernel's operation: a) gathering safety-related information, b) assessing the safety requirements and c) adapting the LoS by adjusting the operation mode of system components.

3.1 Gathering safety-related information

Safety-related information consists in safety rules that are defined in design time, and in data validity and other health information collected in run time. While design time information can be statically stored in some safety information database, run time information must be continuously and periodically obtained. Both design time and run time safety information is required to determine, for each functionality, the highest LoS in which it can be provided. In this section we discuss how this information is gathered.

3.1.1 Defining design time information

Safety requirements should be stored as rules in a database accessible to the Safety Kernel. These rules refer to validity attributes that need to be gathered in runtime, which are provided at the output of some components, as well as to temporal bounds for the execution of some components, which must be monitored by the Safety Kernel. The rules must also express, for each LoS of each cooperative functionality, how to assess the validity attributes and timeliness of the components and adapt the LoS.

The Safety Kernel, whose role and operation do not depend on the semantics of cooperative functionalities, only evaluates the rules using an appropriate rule evaluator engine, which is generic and not developed for particular rules or functionalities. For example, if a certain LoS of a cooperative functionality requires that variable V1 is lower bounded by some value (e.g., V1>0.9), then the Safety Kernel will just have to know the bound, the run time value of V1, and the comparison that needs to be done, in order to determine a Boolean value indicating if the LoS is sustainable. The specific meaning of the bound, or of the current value of V1, is irrelevant from the perspective of the Safety Kernel.

Nevertheless, the design of the Safety Kernel requires the specification of the rules format and their interdependencies. The complexity of the rules can vary from a collection of independent checks of data validity to a sequence of interdependent checks of data validity and timeliness information.

These rules should support the following decisions:

- Determination of a maximum local LoS, at the node, for each cooperative functionality that constitutes an upper bound for the effective LoS;
- Determination of the effective LoS for each cooperative functionality based on the maximum local LoS and, possibly, information from other nodes concerning their own perspective on the LoS of the cooperative functionality;
- Determination of the performance level of each functional component at the node.

The way these rules are generated is outside the scope of the Safety Kernel definition.

3.1.2 Collecting run time information

Run time information refers to data validity and to execution delays, which can be collected from functional components. The focus on this specific data stems from the considered fault model, which, in the case of KARYON, includes both value faults (affecting sensor data that is needed by control algorithms providing the functionality) and timing faults (of some components required to provide the functionality).

The Safety Kernel implements an interface to allow the needed information to be retrieved from, or provided by the functional components. This interface will be known to the designer of functional components, and must be used when implementing the components, whenever necessary for the sake of collecting data validity or timeliness information. The details on this interface are provided ahead in the deliverable.

The collection process is continuous and periodic. That is, the Safety Kernel will be periodically collecting information and analysing it, thus allowing some upper bound to be established on the time needed to detect a significant change of validity or timeliness.

3.2 Assessing the safety requirements

Assessing safety requirements means, in practice, verifying if safety rules established in design time are satisfied in run time. This is done using the periodically collected information on validity and timeliness, which is fed into an engine that performs the necessary checks, as defined by each rule. The definition of this engine is dependent on the syntax of the rules, and is described in Section 6.2.

Based on this assessment, the Safety Kernel is able to determine the LoS for each cooperative functionality and, from that, the performance level at which each component must operate.

3.3 Adapting the level of service

In general, the main objective of the Safety Kernel is the adaptation of the LoS of the cooperative functionalities under its supervision.

The actual LoS of a cooperative functionality may not only be dependent on the assessment of the local components but may also be determined by information received from other vehicles. Due to the cooperative nature of the performed functionality, the Safety Kernel may have to consider the maximum LoS that is possible on other vehicles realising the functionality in the same scope. This depends on the specific functionality, and on how it is designed. There are basically two options: a) the functionality may be designed assuming that all involved vehicles are coherently executing the functionality in the same LoS, or b) it may be designed assuming that each vehicle executes the functionality in a different LoS (possibly knowing in which LoS are the other vehicles executing the functionality). In the first case, the Safety Kernel will locally enforce a LoS that takes into account the LoS information received from other vehicles (through a cooperative LoS evaluator component, described in Section 3). Otherwise, the locally enforced LoS will be the one that is determined upon the assessment of safety requirements.

For example, in the case of a cooperative functionality where its LoS is based on an agreement between the participating nodes, the LoS enforced by the Safety Kernel would be the highest that can be maintained across these nodes. Whenever the LoS has to be degraded in a certain node, e.g., because this node is experiencing some failures, this change must be communicated to the other participating nodes and reflected on their own enforced LoS. Likewise, in the opposite situation, in which a certain node is able to operate at a higher LoS, this information is propagated to the other participating nodes and, if they can all perform in this higher LoS, then the locally enforced LoS is raised.

When the locally enforced LoS changes, this implies some sort of reconfiguration of the system functions. For this matter, we consider that every functionality has, in general, more than one LoS and that there may be several functions involved in the implementation of the functionality. Each of these functions can be necessary for the provision of several functionalities, as exemplified in Figure 1. In the figure, system function 2 is used in both cooperative functionalities (it could be, for instance, a function to determine the front distance, which is used both in Platooning and in cooperative lane change functionality).

	Cooperative Functionality 1	**Cooperative Functionality 2**
System Function 1	X	
System Function 2	X	X
System Function 3	X	
System Function 4		X

Figure 1: Example of functions being used in the provision of different functionalities.

Adapting the LoS of a cooperative functionality requires changing the mode of operation of specific related system functions, which is in fact a way of changing the performance of these functions. Given that a function can be implemented as a single component (with multiple modes of operation) or by multiple components (each one executing a different algorithm), changing the mode of operation, or the performance, can be done by:

a) Reconfiguring a single component, or

b) Selecting one component among the several components that (redundantly) implement the function with different performance levels.

These different options are reflected on the architecture of the Safety Kernel, that is, on the mechanisms and components that need to be included in the Safety Kernel.

The LoS has to be adapted in a timely manner. Consequently, a change in the mode of execution of specific system functions has to be guaranteed to happen within certain temporal bounds.

4. Architecture of the Safety Kernel

This section describes the architecture of the Safety Kernel to manage the LoS of cooperative functionalities. The Safety Kernel includes the necessary components to perform the tasks identified in Section 3.

More specifically, the Safety Kernel has to perform the following tasks:

- For the components that have their outputs monitored, their data validity information must be gathered and validated against the safety rules. Possibly, the LoS of cooperative functionalities and the performance level of components may change.
- For the components located above the hybridization line, that is, complex components whose timeliness might not be guaranteed at design time, their timeliness is monitored, which required observing their execution time, so that the system knows whether the defined execution bounds are being fulfilled.
- For the functions that have multiple components, where each implementation produces an output, the Safety Kernel must choose which of the produced outputs will be the function output that is forwarded to other functions.

In the next sections, we begin by reviewing the role of the Safety Kernel within the overall KARYON architecture. Then, we present the components of the Safety Kernel and also external related components that play an important role in the operation of the Safety Kernel. This includes functional support from the Operating System. Next, we describe the interfaces between the components of the Safety Kernel and the nominal control system components. Finally, to guarantee the timely operation of the Safety Kernel components, the required scheduler support is discussed.

4.1 Relation with overall KARYON architecture

According to the defined KARYON architecture, system components are organized in three levels, separated by the hybridization line and by the semantics line. The hybridization line differentiates components that are proven timely in design time and those that are not (and thus might do timing faults in run time). The semantics line differentiates the components that realize the functionalities (and thus are developed with awareness of functionality semantics) and the components that provide support (and generic) functions. These functions are developed independently on the specific functionalities (and thus are unaware of functionality semantics). The Safety Kernel is positioned in the lowest level, below both the hybridization and semantic lines. This means that the Safety Kernel must be proven to behave correctly and in a timely way in design time. Besides that, it means that the Safety Kernel is not aware of the functionality semantics, that is, the components included in the Safety Kernel are designed independently from the considered cooperative functionalities.

The functional components of the system are located in the two upper architectural levels, whose difference is the timeliness guarantees each one provides. The task of the Safety Kernel is to control the components in these levels, ensuring that they operate with the necessary performance levels to meet some desired LoS for the different functionalities. The required LoS is also determined by the Safety Kernel, and will be the one that is necessary to satisfy the functional safety goals.

4.2 Safety Kernel Components

To perform its role, the Safety Kernel exchanges information with other components in the KARYON architecture. The exchanges with different types of components embody different aspects of the Safety Kernel's operation, like receiving validity data and sending commands for controlling the operation mode of functional components. For this reason, we see the Safety Kernel as a set of components, with clearly defined and separated concerns, which are combined to verify and guarantee the operational conditions for safety. For the Safety Kernel to be relied upon for the provision of safety-critical functionalities, its components have to be proven to exhibit the necessary reliability and timeliness in design time. This section describes these components.

The Safety Kernel collects the data validity or timeliness information made available by the monitored functional components, assesses it and adapts the LoS of cooperative functionalities by reconfiguring the functional components according to the predefined rules. The components of the Safety Kernel always involved in this control loop are: the Rules Database, the Local LoS Evaluator and the Safety Manager. The assessment is done by the Local LoS Evaluator and its result is forwarded to the Safety Manager.

As mentioned before, each function of the nominal control system can be implemented in a single or with multiple components. When a function has two different implementations, where each one corresponds to a specific performance level, the Safety Kernel will have to assess the execution time of the components above the hybridization line, comparing it to some predefined execution bound. An implication of having a component of this type is that the Safety Kernel has to guarantee that only the output of one of the components (the selected one, according to the LoS) is forwarded to other components. Two other components of the Safety Kernel will also cooperate in this process: the Data Component Multiplexer and the Timing Failure Detector.

Another possibility is for the functions to have different modes of operation. In this case the performance level of a function can be adjusted through a reconfiguration of its mode of operation.

Finally, for every cooperative functionality, the Safety Manager uses the result produced by the Local LoS Evaluator and, possibly, the result from the Cooperative LoS Evaluator and decides, based on rules, if there will be a change in the effective LoS. The Cooperative LoS Evaluator is an external component, which will be described in section 4.3.1.

All these components and their interactions are represented in Figure 2. They will be described in the following sections. The figure also shows two example components, where function A has two implementations and produces an output to function B.

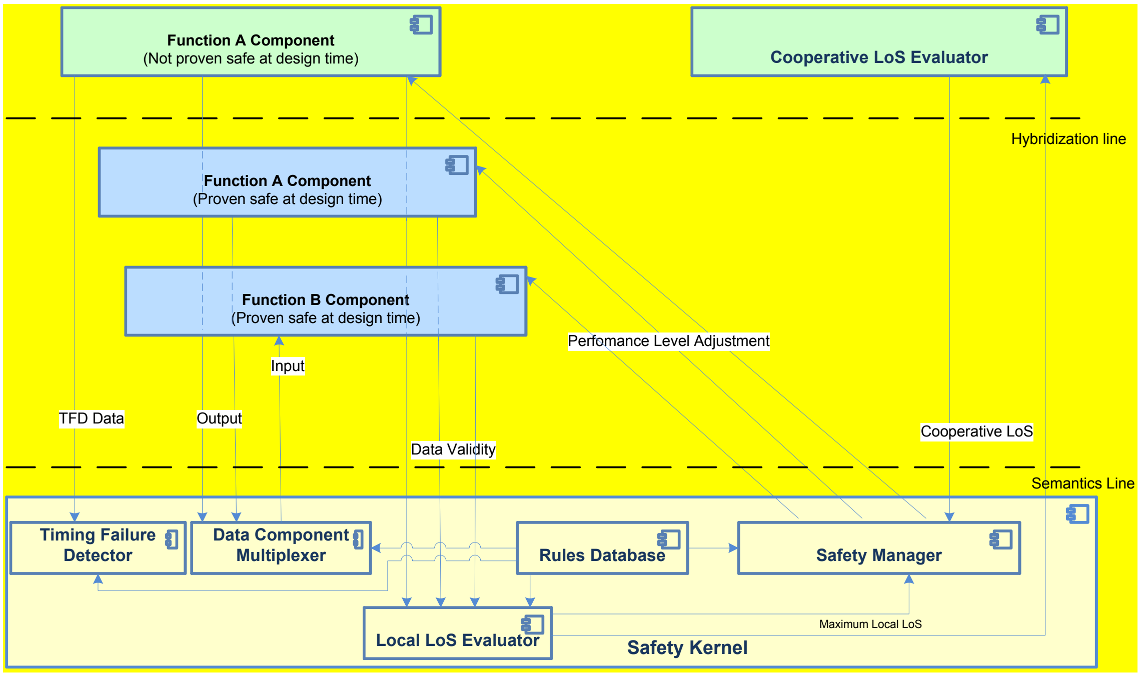


Figure 2: System components overview and interaction.

4.2.1 Rules Database

The rules database contains the safety rules derived in design time, as mentioned before. The safety rules include:

- Rules used to assess, at runtime, under which LoS a specific cooperative functionality may operate. The assessment is done by comparing data validity and timeliness information with respect to the bounds expressed in the safety rules.
- Rules used to define the performance levels for every reconfigurable component in dependence of the LoS of each cooperative functionality.
- Rules to guide the Safety Manager's decision on how to handle the input provided by the Cooperative LoS Evaluator, in addition to the input received from the Local LoS Evaluator.

The complexity of the rules can vary from a collection of independent checks of data validity to a sequence of interdependent checks of data validity. A solution for expressing these rules within the Safety Kernel is provided in Section 6.4.

4.2.2 Data Component Multiplexer

As explained in Section 4.1, some functions may have components above and below the hybridization line. Some, simpler, are proven to behave in a timely way. Others, more complex, have unpredictable execution times. More complex implementations produce a better result, with higher quality than simple implementations.

As functions depend on other functions as data sources, the result of a function with multiple implementations with a quality that is lower than what is possible will negatively influence other functions that consume this data. The result of a function with multiple components should always correspond to the output of the component that better satisfies the safety requirements of the LoS of all cooperative functionalities that makes use of it. For instance, when a complex implementation of a function misses a deadline, the result provided by the function must be the output from a simpler component.

To avoid any time penalty whenever a complex component misses a deadline, all components (complex and simple) of a function may be executing simultaneously. In this case, the output of a timely component should be selected as the actual result of the function. And therefore, the deadline miss only affects the quality of the result but not the time at which it is produced.

For example, considering two components that perform the same function, one with a complex, but unpredictable, algorithm and the other with a simple, but predictable, algorithm. When these two implementations are concurrently in execution, the result produced by the simpler but reliable implementation can always be used when the result from the more complex but unreliable implementation has not arrived in time. This way, it is possible to ensure that a valid output (produced by the simpler implementation) will always exist, and if a better and valid result (produced by more complex implementations) exists the later will be used.

The task of the Data Component Multiplexer is to decide which component's output will be the result of the function, discarding the others. To do so, each multiplexer must access the Rules DB and act accordingly.

These components of the Safety Kernel are crucial to achieve safety in hybrid architectures such as KARYON, since they allow masking failures in complex components by using the result of another component.

Functions that use the produced output forwarded by a multiplexer are independent from the function before it.

The Data Component Multiplexer does not apply to functions with a single implementation.

4.2.3 Timing Failure Detector

The Timing Failure Detector (TFD) component is in charge of detecting failures in the time domain in an implementation of a component above the hybridization line, since these are the ones whose execution time is unpredictable and not bounded. Hence, this component of the Safety Kernel acts as a watchdog, looking up permanently for delays and crashes.

In order to achieve this, each real-time complex component (above the hybridization line) must send a periodic heartbeat to the TFD. When executed, the TFD must, for all the components, check whether their heartbeats are still valid, or not, by evaluating their freshness. If a heartbeat is too old then this means that a delay or crash has happened. This information about the violation of a timing bound will be used in the evaluation of safety rules.

The Timing Failure Detector does not apply to functions with a single implementation.

4.2.4 Local LoS Evaluator

The role of the Local LoS Evaluator is to evaluate and assess the data validity and timeliness information of the monitored components against the Rules Database. Based on this assessment, the Local LoS Evaluator determines the maximum LoS at which each cooperative functionality is able to safely perform from the perspective of the local node. This result is then made available to the Safety Manager (discussed on section 4.2.5) and to the Cooperative LoS Evaluator (discussed on section 4.3.1).

4.2.5 Safety Manager

The Safety Manager supplements the operation of the Local LoS Evaluator by also considering the output of the Cooperative LoS Evaluator in the production of the Effective LoS of a cooperative functionality. Another job of the Safety Manager is the reconfiguration and selection of components whenever the LoS of a cooperative functionality changes.

The way to produce the Effective LoS from the inputs received from the Local LoS and the Cooperative LoS is not necessarily fixed by the Safety Manager. The idea is that this may be configured according to the specific functionality. One possible way of performing this configuration is by defining rules for this purpose. These rules will define the function that will be performed for determining the effective LoS, that is, we will have that `Effective LoS = Function(Local LoS, Cooperative LoS)`. For example, the function could be `Min(Local LoS, Cooperative LoS)`, where the Effective LoS would be the lowest value between the two inputs of the Safety Manager.

The reconfiguration of components is necessary to change their performance level in response to the LoS change of any cooperative functionality. The information about which components are affected by a change in the LoS will come from the Rules DB. The Safety Manager is only responsible for propagating these changes to the respective components.

Therefore, periodically, the Safety Manager makes available the Effective LoS of all cooperative functionalities and reconfigures and selects the respective components, whenever required. The actual reconfiguration and adjustment mechanisms are executed within each component, and the Safety Manager just has the responsibility of triggering these changes on the right components.

4.3 External related components

This section describes other components, external to the Safety Kernel, that play an important role in its operation. These descriptions are deliberately not detailed, rather consisting of the knowledge the Safety Kernel has of these components.

4.3.1 Cooperative LoS Evaluator

For each cooperative functionality, the Cooperative LoS Evaluator has the purpose of exchanging data with similar components of other participating nodes and, eventually provide information to the Safety Manager about the LoS of other vehicles.

The operation of the Cooperative LoS Evaluator and the algorithms it uses to exchange information with other vehicles is not dealt as part of the Safety Kernel. In fact, it is possible that a different Cooperative LoS Evaluator is defined for each cooperative functionality. At each periodic execution, this component may influence the output of the Safety Manager. Since this component is defined as a complex component (because it is not possible to guarantee in design time that communication with other vehicles is always possible), the solutions concerning what it does are varied and depend on what may be more desirable for some functionality.

A possible approach for the operation of the Cooperative LoS Evaluator is to have it producing a Cooperative LoS based on an agreement between the participating nodes. This LoS would correspond to the lowest Local LoS that is possible at every participating node. In this case, this LoS would become the Cooperative LoS for the cooperative functionality at every participating node. For this to become effective, the Safety Manager will need to perform the function `Min(Local LoS, Cooperative LoS)`, so that the agreed LoS becomes an upper bound for the Effective LoS at each node.

It is also possible to consider that some cooperative functionalities will not require an agreement on the LoS, in which case the Cooperative LoS Evaluator does not have to produce any value. In that way it will not influence the Effective LoS at each node. In this case, cooperation is achieved just by the exchange of other information relevant for the functionality, without relying on any assumption about a consistent execution of the functionality (in the same LoS) by the involved vehicles. This has necessarily to be reflected in the control algorithms.

It can be added that, because the Cooperative LoS Evaluator is application dependent, it could in fact have multiple modes of operation. For instance, its behaviour could change in accordance with the availability of a communication channel with other nodes. This component may also decide to remain silent and not produce any value. This could happen, for instance, when an agreement could not be achieved. This can be exploited for setting safety rules involving the timeliness or the validity of information provided by the Cooperative LoS Evaluator, forcing the LoS to be reduced in case these safety rules are not satisfied.

Although the Cooperative LoS Evaluator plays an important role towards safety, it cannot be part of the Safety Kernel mainly due to the uncertainty in the communication with other nodes. Therefore, it is located above the hybridization line, outside the Safety Kernel.

4.3.2 Operating System support

Communication between functional components and the Safety Kernel is handled by the Operating System (OS). As such, the interfaces required by the Safety Kernel (described in Section 4.4) shall be provided by the OS, which ensures that the primitives composing these interfaces are provided in READ–WRITE pairs constituting an information flow channel with one writer and one or more readers. In each pair of READ–WRITE primitives, one of the primitives is intended to be used by one of the Safety Kernel components, whereas the other one shall be used by a software component external to the Safety Kernel (either a functional component or the Cooperative LoS Evaluator). Both types of primitive should be non-blocking, atomic, and the OS should provide the following guarantees:

- **READ** calls: the value that is read is the one written in the last invocation of the corresponding WRITE call; until overwritten, a value can be read multiple times and/or by multiple readers;
- **WRITE** calls: the provided value overwrites the value previously provided by the same writer.

The way these information flow channels are implemented is abstracted by the OS, and should be transparent to the remaining components. It is the responsibility of the OS to ensure, by whichever means necessary, that READs are consistent with the latest WRITE.

The OS must also provide scheduling mechanisms which allow temporal predictability of the interaction flows we here describe. Hence, it is assumed that internal communication, i.e. inside one vehicle, is based on a real-time network (e.g. CAN) and is, therefore, reliable and time bounded. These requirements are described in detail in Section 4.5.

4.4 Interfaces

In order to build the components described in Section 4.2, some interfaces must be defined and implemented, in order to allow interaction between cooperative functionalities and the Safety Kernel. These interfaces are depicted in Figure 3.

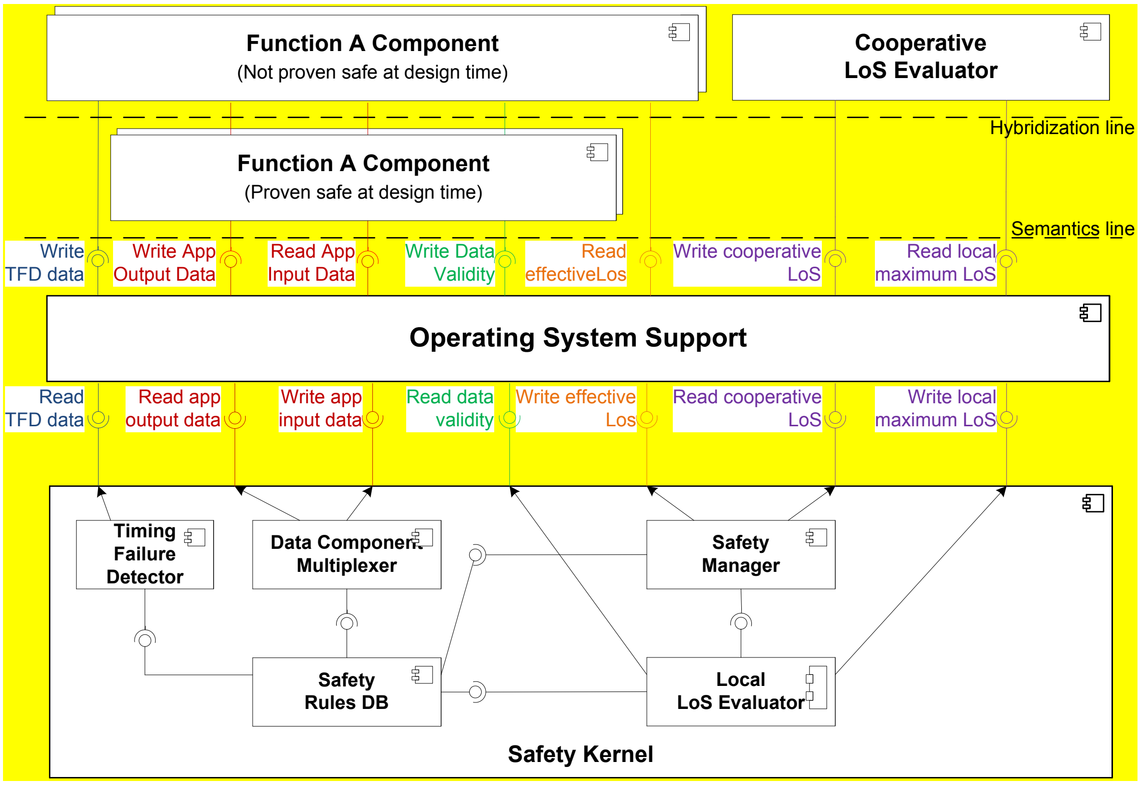


Figure 3: Interfaces between the Safety Kernel and external related components.

Interface	Primitive	Involved components
Data validity interface	Write data validity	Functional components
	Read data validity	SK – Local LoS Evaluator
Timing Failure Detector interface	Write TFD data	Functional components (Above hybridization line)
	Read TFD data	SK – Timing Failure Detector
Cooperative LoS Evaluator interface	Write local maximum LoS	SK – Local LoS Evaluator
	Read local maximum LoS	Cooperative LoS Evaluator
	Write agreed LoS	Cooperative LoS Evaluator
	Read agreed LoS	SK – Safety Manager
Data Component Multiplexer interface	Write app output data	Functional components
	Read app output data	SK – Data Component Multiplexer
	Write app input data	SK – Data Component Multiplexer
	Read app input data	Functional components
Mode switch interface	Write enforced mode	SK – Safety Manager
	Read enforced mode	Functional components

Table 1: Interfaces of the Safety Kernel

4.4.1 Data Validity Interface

This is the interface used to feed the Local LoS Evaluator component with the data validity sent from the different functional components of the system. These components must then be able to, in runtime, send this data to the Safety Kernel, so that when it executes it can determine the LoS at which the cooperative functionalities are able to perform.

The primitives used to support these operations are the following:

- **WRITE_VALIDITY_DATA** – This primitive, to be used by the applications, allows any component to send its validity data to be checked and evaluated by the Safety Kernel;
- **READ_VALIDITY_DATA** – This primitive, to be used by the Safety Kernel's Local LoS Evaluator, allows the Local LoS Evaluator to read the data sent by components using the previous primitive.

One example of a workflow is pictured in Figure 4. In this diagram and in those which follow, the grey dashed arrows inside the Operating System represent the provided communication channel, as described in Section 4.3.2.

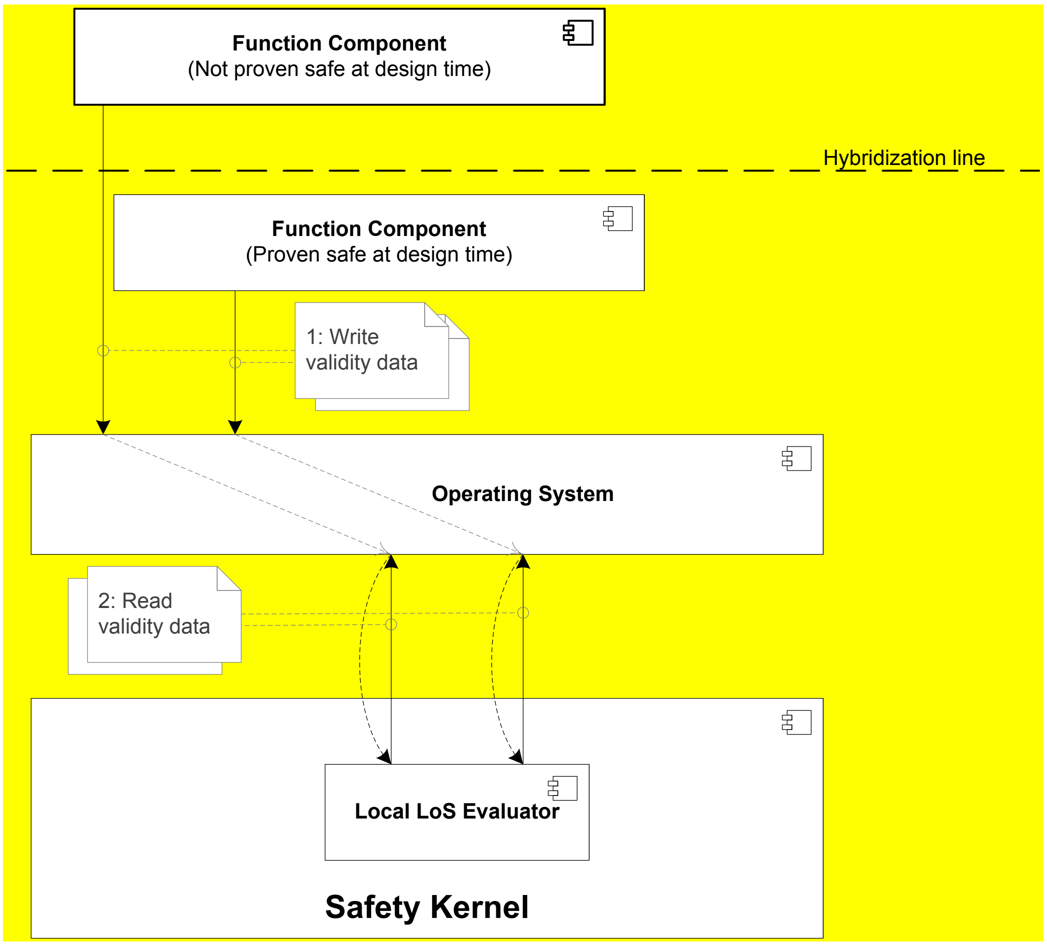


Figure 4: Interaction with the Data Validity Interface.

As pictured, this interaction is done in two-steps. In the first one, each functional component calls the Write Validity Data (1) interface to send its own validity data to the Safety Kernel. The Operating System transfers this data from the origin port to the destiny port. The second step is done by the Local LoS Evaluator that, for each component, uses the Read Validity Interface (2) to get the data sent by the components.

4.4.2 Timing Failure Detector Interface

This interface supports the detection of timing failures. In the Safety Kernel, this interface is realized by the Timing Failure Detector (TFD) component.

Each functional component above the hybridization line must, periodically, and at a predefined minimal rate, send a heartbeat informing the Safety Kernel that progress is being made and that its planned schedule is being fulfilled. For each of these components, the TFD component must be able to check if any heartbeat has been sent, and if it is valid for the defined timeout or if, in the other hand, its time validity has expired.

- **WRITE_TFD_DATA** – This primitive, to be used by the components, allows them to inform the Safety Kernel about their progress. The time elapsed since the last call should be reset after this call;
- **READ_TFD_DATA** – This primitive, to be used by the Safety Kernel's Timing Failure Detector, allows the TFD component to, for each component, know if the time elapsed since the last WRITE_TFD_DATA call is greater or lower than the predefined period.

One example of a workflow is pictured in Figure 5.

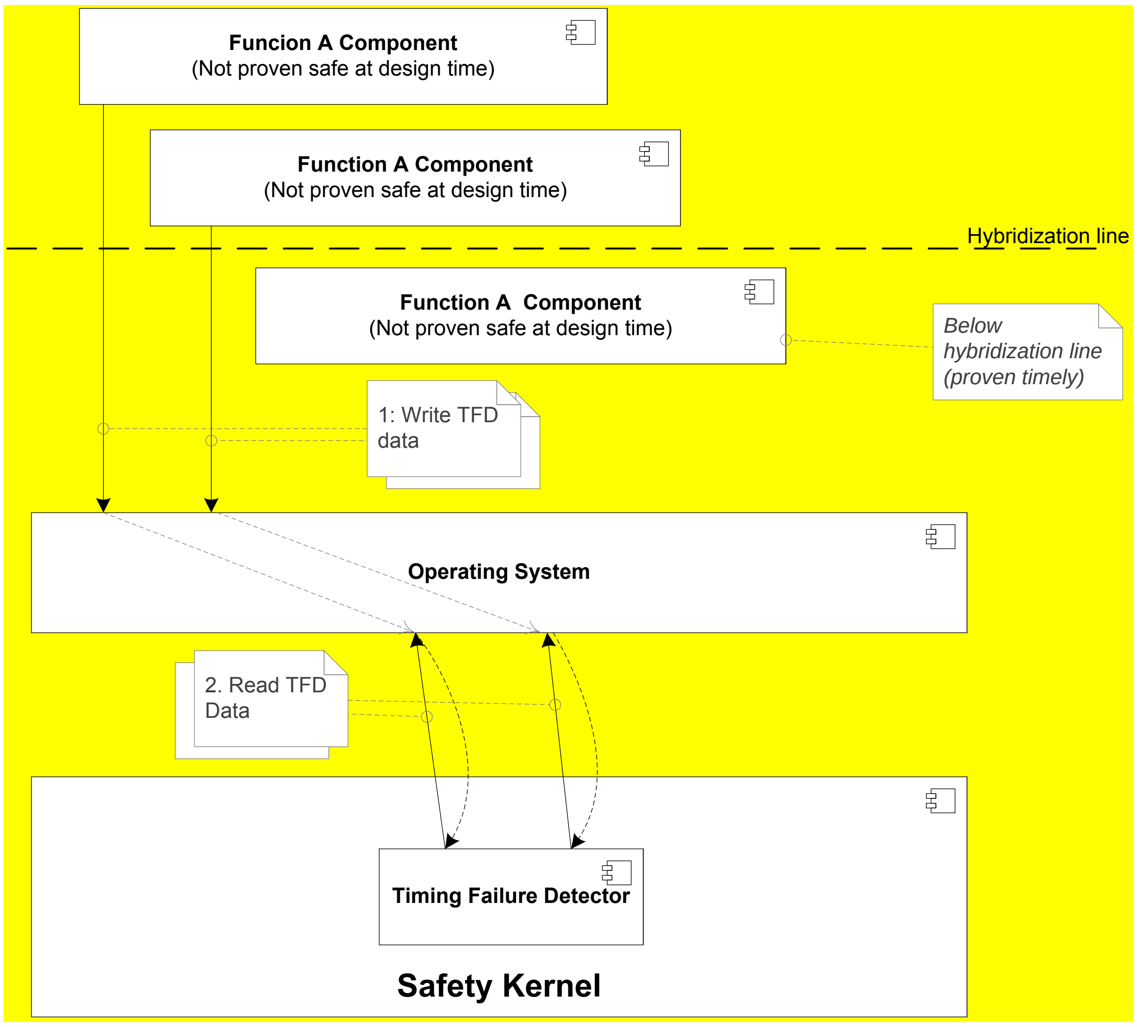


Figure 5: Interaction with the Timing Failure Detector Interface.

As pictured, each component above the hybridization line uses the Write TFD (1) data interface to send a heartbeat to the TFD. Components under the hybridization line (which are proven to be timely safe) do not need to be monitored. For each component above the hybridization line, the TFD component calls the Read TFD Data (2) interface to check their progress.

4.4.3 Cooperative LoS Interface

This interface implements the mechanism used to support the LoS management by the Safety Manager and both Local and Cooperative LoS Evaluators.

It allows the Local LoS evaluator to send the Local Maximum LoS to both the Safety Manager and the Cooperative LoS Evaluator and, also for the latter to inform the Safety Manager of the Cooperative LoS.

- **WRITE_LOCAL_MAXIMUM_LOS** – This primitive is used by the Local LoS Evaluator to make available the information of the maximum LoS to the Safety Manager and to the Cooperative LoS Evaluator that is possible at the node;
- **READ_LOCAL_MAXIMUM_LOS** – This primitive is used by both the Safety Manager and the Cooperative LoS Evaluator to read the LoS written using the previous primitive;
- **WRITE_COOPERATIVE_LOS** – This primitive is used by the Cooperative LoS Evaluator to inform the Safety Manager of the Cooperative LoS;
- **READ_COOPERATIVE_LOS** – This primitive is used by the Safety Manager to read the Cooperative LoS written by the Cooperative LoS Evaluator using the previous primitive. Since the Cooperative LoS Evaluator is not proven timely safe, this primitive must also allow the Safety Manager to know if the available Cooperative LoS is still valid or not (i.e. that the time elapsed since it was written is lower than a predefined delta/timeout).

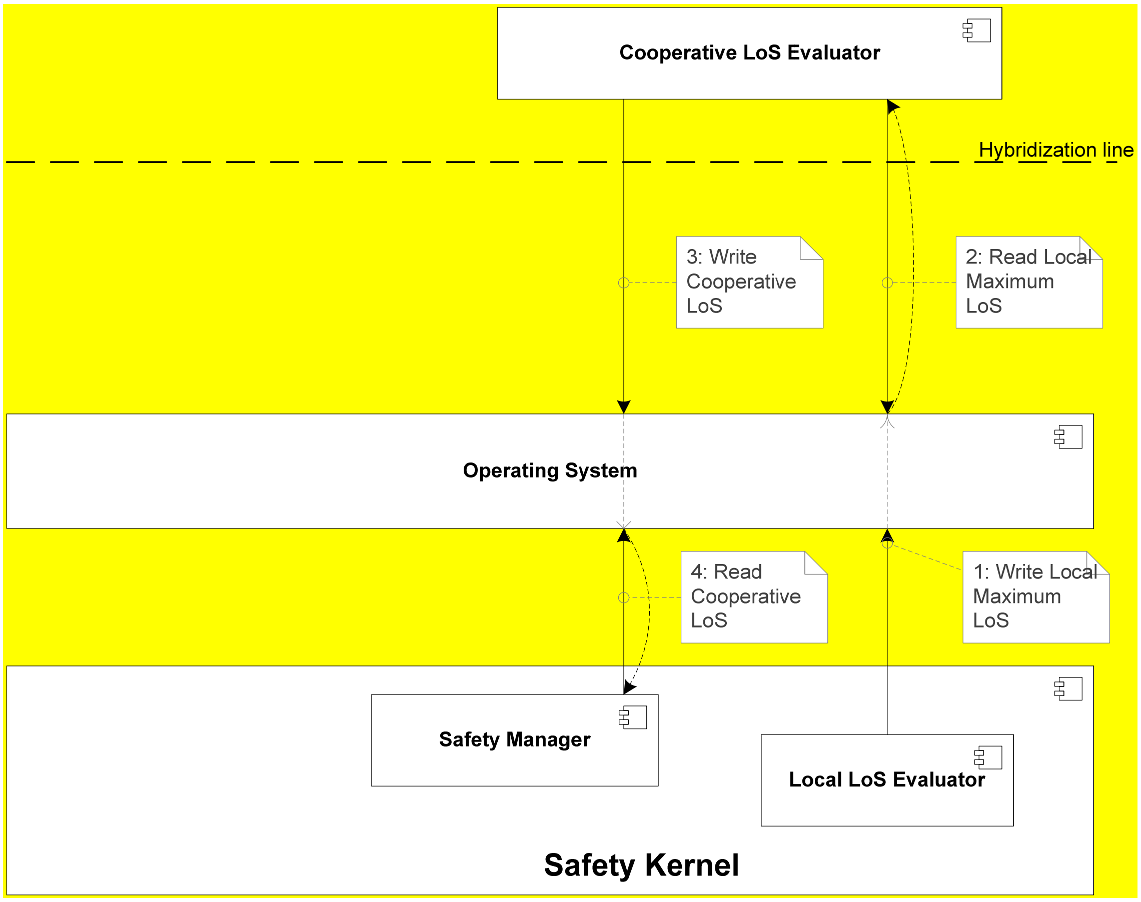


Figure 6: Interaction with the Cooperative LoS Interface.

In this interaction, the Local LoS evaluator uses the Write Local Maximum LoS interface (1) to inform the Cooperative LoS Evaluator of the local capabilities that are possible to offer by the functional components. This is read by both the Cooperative LoS Evaluator with the Read Local Maximum LoS (2) interface.

The Cooperative LoS Evaluator uses the Write Agreed LoS interface (3) to inform the Safety Manager of the Cooperative LoS and the latter calls the Read Agreed LoS interface (4) to read that LoS value. By using the validity period of this port, the Safety Manager is able to know if this value is fresh, and whether it is still valid, or not.

4.4.4 Data Component Multiplexing Interface

This interface supports the functioning of the Data Component Multiplexer. It allows the different implementations of the same function to make their output values reach the Data Component Multiplexer. Other functions that receive the output from that function may then read the appropriate value, which has been previously selected by the Data Component Multiplexer. The Data Component Multiplexing interface abstracts this whole process, both to the component providing output (which we will call Component A for the description of this interface) and to the component seeking input (Component B). The Data Component Multiplexing interface consists of the following primitives:

- **WRITE_APP_OUTPUT_DATA** – This primitive, to be used by the applications, allows each implementation of Component A to communicate its output value to whichever other functions may need it (including Function B). The value is provided along with a data validity measure, and reflects the output of Function A at a given LoS.
- **READ_APP_OUTPUT_DATA** – This primitive, to be used the Safety Kernel's Component Data Multiplexer, allows the Component Data Multiplexer to read the values provided by different implementations of a Function A - i.e., the outputs of Function A for the various LoS.
- **WRITE_APP_INPUT_DATA** – This primitive, to be used by the SK's Component Data Multiplexer, allows the Component Data Multiplexer to communicate (to whichever functions may need, including Function B) the appropriate value to be considered as the output Function A. This value is selected by the Component Data Multiplexer among the outputs provided by the implementations of Function A at different LoS.
- **READ_APP_INPUT_DATA** – This primitive, to be used by the applications, allows the implementation(s) of Function B to read the output provided by Function A when needed. In case Function B has multiple implementations (for different LoS), some of them may not use the output from Function A at all. Through this primitive, an implementation of Function B may read the output from Function A without needing to know about the variety of implementations of Function A.

The workflow for the use of the Data Component Multiplexing interface is pictured in Figure 7.

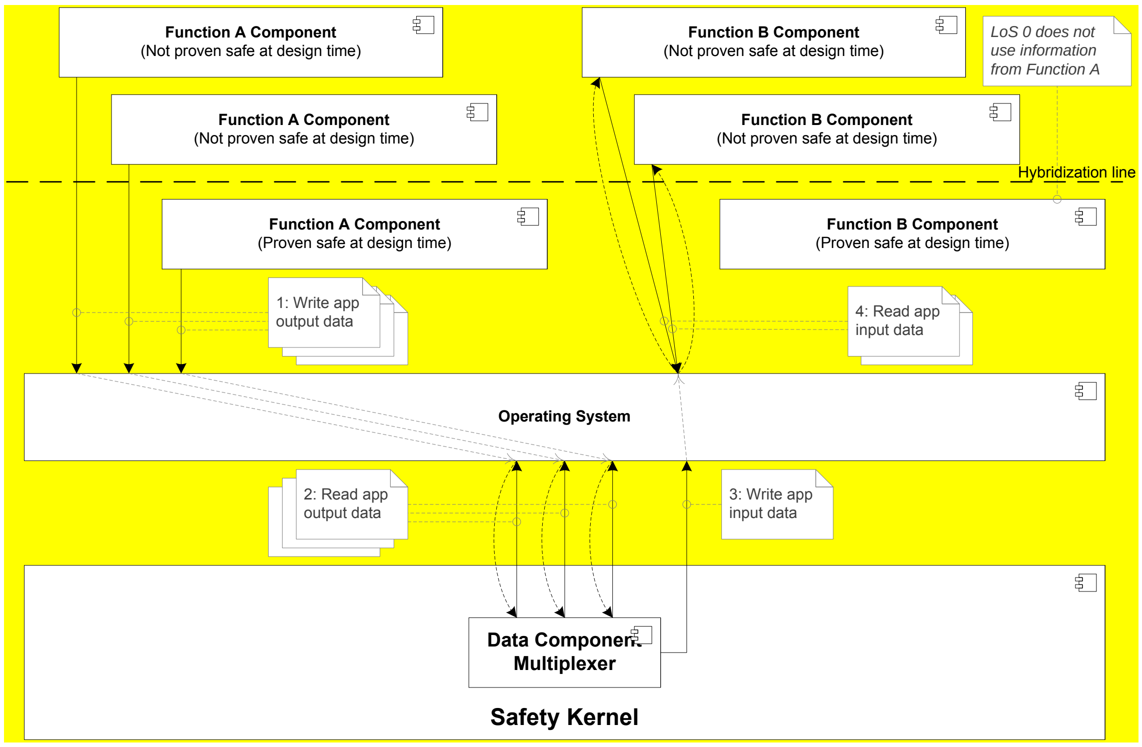


Figure 7: Interaction with the Data Component Multiplexing Interface.

In this interaction, the Data Component Multiplexer acts as intermediary between a function with multiple implementations and other functions. Each component writes its output using the Write App Output Data Interface (1), which is read by the Data Component Multiplexer with the Read App Output Data Interface (2). This interface is called by each component. From the multiple readings, one value is written by calling the Write App input data interface (3). All components of that use this this value as input call the Read App Input Data (4) to read it.

4.4.5 Mode Switch Interface

This interface implements the mechanisms to allow the Safety Manager to force a functional component to switch its mode of execution to a different one, or simply to reconfigure it.

- **WRITE_PERFORMANCE_LEVEL** – This primitive is used by the Safety Manager to reconfigure a specific component for a certain performance level.
- **READ_PERFORMANCE_LEVEL** – This primitive is used by each component to read the performance level set by the Safety Kernel using the primitive above.

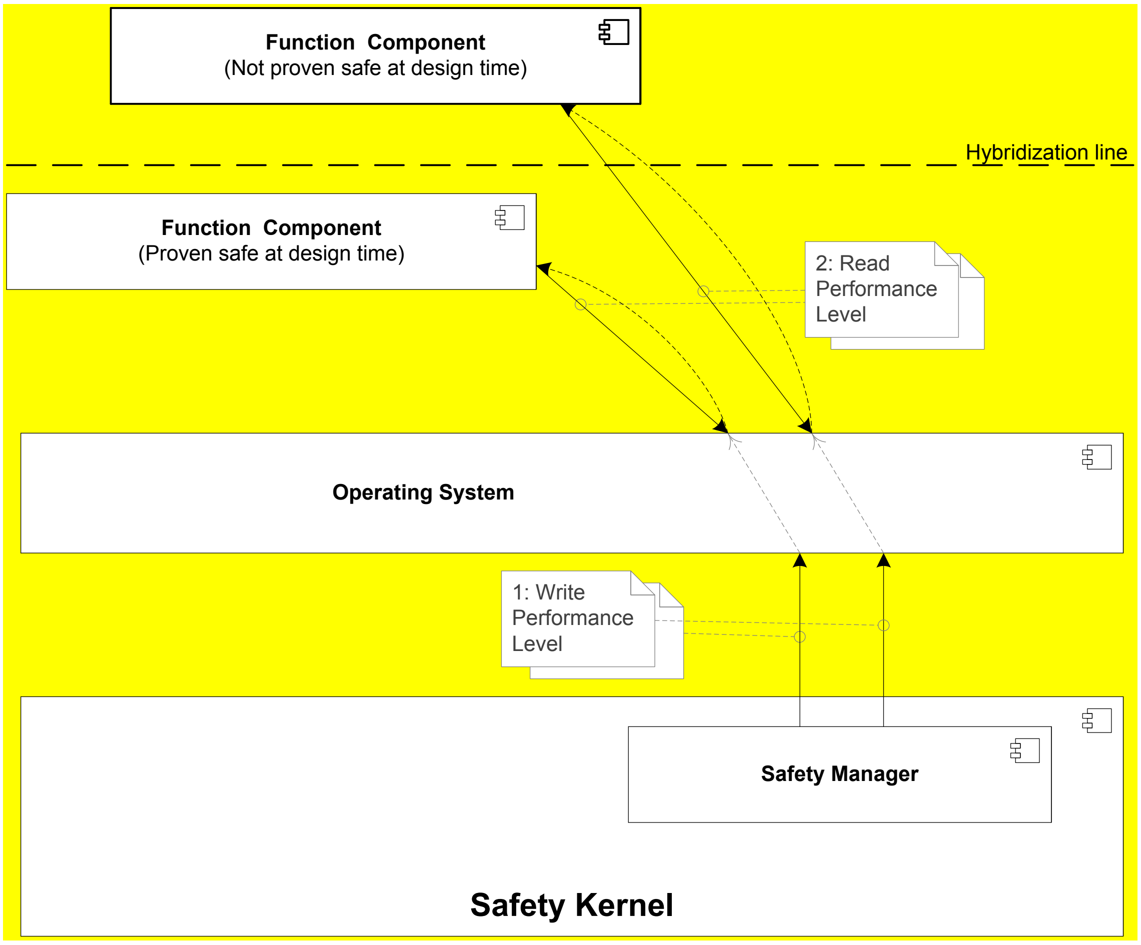

Function Component
(Not proven safe at design time)
Hybridization line
Function Component
(Proven safe at design time)
2: Read Performance Level
Operating System
1: Write Performance Level
Safety Manager
Safety Kernel

Figure 8: Interaction with the Mode Switch Interface.

This interaction is performed between the Safety Manager and the different functional components of the system. The Safety Manager uses the Write Performance Level (1) to inform each component of the mode in which they should operate from now on. This value is read by each component with the Read Performance Level interface (2).

4.5 Scheduler support

In Section 4.3.2, we described the functional support that the operating system should provide to the operation of the Safety Kernel, namely the mechanisms (underlying to the provided interfaces) that guarantee information flow in an asynchronous fashion. However, for this information flow to behave in the timely manner needed to achieve the goals of the Safety Kernel, adequate scheduling of all software components is necessary.

The functionality associated to the Safety Kernel is ensured by interactions between components below the hybridization line (such as the Safety Kernel) and above the hybridization line (such as the Cooperative LoS Evaluator). The safety and timeliness of components below the hybridization line must hold in the event of timing faults in the components above the hybridization line. For this reason, the different components must be scheduled in a way which guarantees that the effects of any timing faults are contained in the scope of their occurrence - i.e., to the component where they happen.

Scheduling must thus be certifiably deterministic and predictable. We can achieve this by scheduling components strictly according to a fixed schedule, defined at design time with windows of activity to fulfill the demand expected for each component's workload. The policy according to which each component schedules its workload (the local scheduler of each component) must be known, so as to determine the minimum guarantee each component should receive. In the event that local scheduling inside a component above the hybridization line diverts, in execution time, from what was assumed in design time, the temporal properties of other components (which get their designated window of activity in any case) are not affected.

Naturally, we cannot have a schedule which covers the whole of the system's lifetime. We can instead have a schedule which covers a bounded time interval and is subsequently repeated. The length of the schedule, which is consequently its period, must be defined to provide a minimum periodic guarantee to each component; each component's minimum periodic guarantee should be such that the timing requirements of the component's workload are fulfilled. Different components may require that their minimum guarantees are specified in relation to different periods. For this reason, the length of the schedule should be the least common multiple of these periods (or a multiple thereof).

As mentioned above, the local scheduling policy of each component is, in general, only relevant to determine the minimum guarantee each component should receive. However, when dealing with components below the hybridization line, the local scheduler's policy must also be certifiably deterministic and predictable; in the case of the Safety Kernel, we need to make sure that, after determining the timing requirements of each module, the scheduling policy guarantees them.

5. Development of Safety Kernel in Terms of SEooC

Since the focus of the KARYON is the Safety Kernel, and this is not an "item" in the sense of ISO 26262 because it is only an element of the architecture and is not related to a specific vehicle model, the SEooC (Safety Element out of Context) approach has been considered appropriate for the analysis. On the other hand, the analysis result achieved by SEooC approach can be extended to similar cases easily, which means a higher efficiency of the effort in context of reusability.

The development of a SEooC involves making assumptions on the prerequisites of the corresponding phase in product development and each information on requirements or design prerequisites is pre-determined in the status "assumed" [14]. In order to well identify the requirements and assumption to develop the Safety Kernel SEooC in a high level of abstraction, the Concept Phase of ISO 26262 is applied. The starting point has been the definition of a general architecture as an abstract item for cooperative driving functions, including all the fundamental elements usually considered to perform such functionality, namely for sensing, data processing, decision making, navigation, HMI, security elements, actuator control, etc. The architecture considered includes data exchange with infrastructures, assuming the availability of the ITS services according to ETSI standards. It has to be pointed out that the architecture so defined does not include the LoS concept and the Safety Kernel, which, according to ISO 26262, shall be considered as safety measures and therefore shall not be taken into account for the subsequent Hazard Analysis and Risk Assessment.

With the above analysis, although related to few but significant enough, critical situations (e.g. intersection crossing, platooning, roundabout merging), the various hazardous events were assessed to determine severity, controllability and time exposure. The result was what had been expected, i.e. that in the case of input data failure, the hazardous events had to be classified ASIL D, and that the safe state had to be driver's takeover in a short time. This preliminary analysis lead to the trivial conclusion that the architecture elements involved in cooperative driving shall be realized at ASIL D and that cooperative driving implies frequent switching between automated and manual driving. However, it should be noted that a higher ASIL (ASIL E) has been recently proposed for cooperative driving functions.

Of course this conclusion is not satisfactory for reasons of cost and market acceptance. Switching from automated driving to manual mode for every failure, requires the driver to stay highly conscious such that it makes no sense for him/her concerning the level of comfort. In addition, several switching between these modes cause the product seem unreliable to user's eyes. Therefore, it is decided to consider input data failures introducing a scale with various levels, according to the intuition that has been the base of KARYON proposal regarding LoS and data validity concepts. In fact, the idea of LoS was clicked first in order to manage the operating modes according to operational conditions, but in continue it got extended to deal also with failures and so the safety mechanism.

This failure data severity scale enabled the introduction of several safe states, which can be associated to different automation and performance levels, and have the capability to limit the need of stringent ASIL requirements for all the parts of the architecture.

According to the above concept, the safety mechanisms to address data failures have been identified in the shape of functional architecture elements that include some of the functions that are in the focus of KARYON, namely the Safety Manager, LoS Evaluation and Timing Failure Detection.

This section describes the application of the ISO26262 Concept Phase in which the Safety Kernel is derived from a preliminary architecture, so that at the end the initial assumptions together with the resulting requirements form the SEooC fundamental assumptions and requirements[1].

5.1 Abstract Item Definition

An item is a system or array of systems that implement a function at vehicle level. Since the KARYON architecture is not in context of a particular vehicle, an abstract item is needed to be defined. In this part of the approach the aim is to perform the item definition according to ISO 26262-3 such that the item be abstract from a specific vehicle, but also generalizable to different cooperative functions.

In the item definition the boundary of the item is defined, its interactions with the environment and also its dependencies are clarified such that an adequate understanding of the item be available to support subsequent activities. For this purpose, the functional view of the abstract item is illustrated in Figure 9. In this figure, the abstract item is displayed as a gray block containing internal functions which have interactions with external ones.

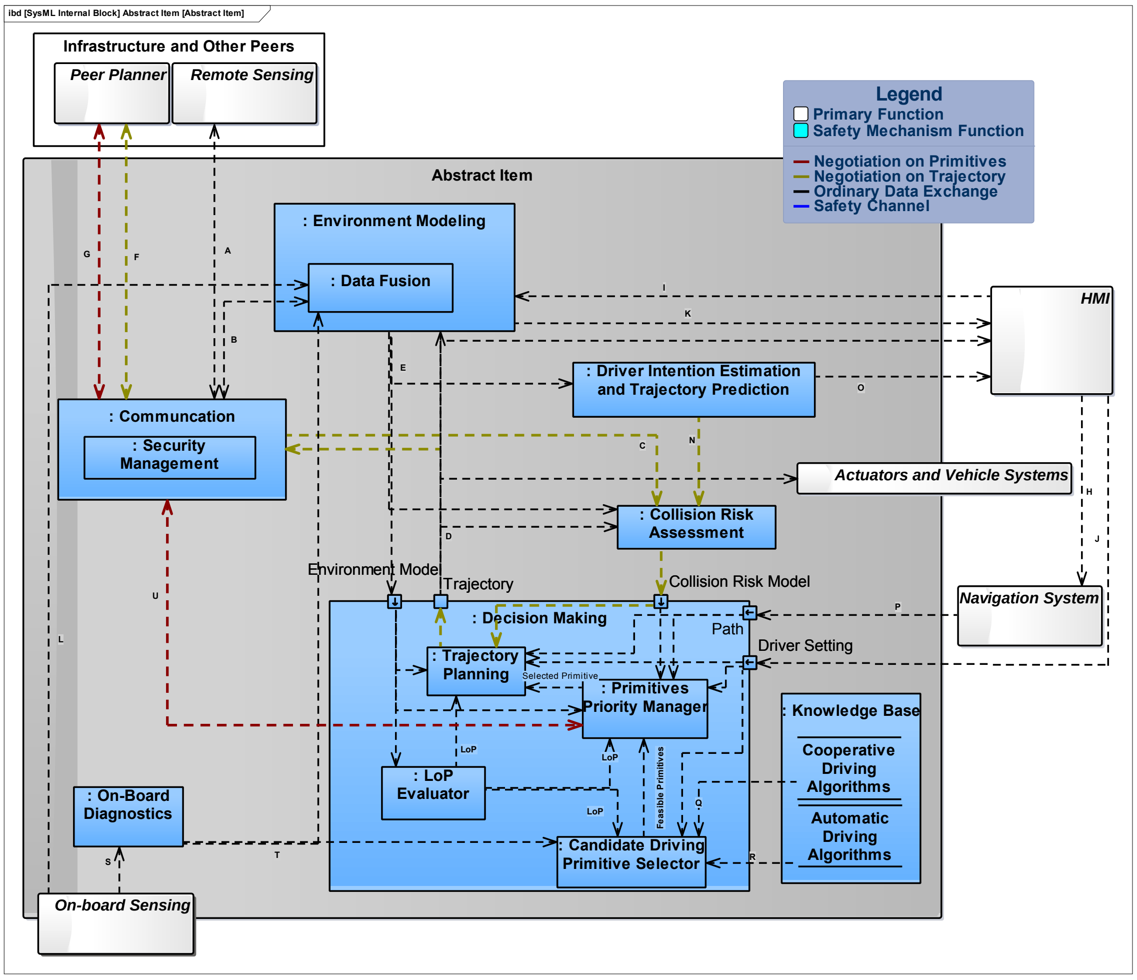


Figure 9: Abstract Item Preliminary Architecture

[1] The presentation of the approach in this section is supported by the diagrams in which the SysML as a semi-formal language is utilized to model and illustrate the corresponding concepts, requirements, elements and their relation.

It's obvious that this preliminary architecture does not contain the Safety Kernel. Table 2 supports the definition of the architecture by supplying more detailed information for each function. In concern of the Safety Kernel SEooC, these definitions and specifications are considered as assumptions external to SEooC.

Table 2: Primary functions description. The starred* links are bidirectional.

Function / Element Name	Type	Input	Output	Task
Environment Modeling	Internal Function	B*, I, L, D, T	B*, E, K	All the information collected by onboard sensors or received via communication with other vehicles and infrastructure are centralized here. It provides the necessary view to each function. The update rate of information coming from external resources shall be compliant with ITS standards. No assumption for ASIL of data is defined.
Communication	Internal Function	A*, B*, D, F*, G*, U*,	A*, C, F*, G*, U*	It's responsible for setting up the communication with other vehicles and infrastructure. Atomic commitment is required to interact with external systems. This feature is necessary for synchronization of high level information as planned trajectory. Redundant communication channels are assumed to available for some critical functions to assure an adequate ASIL for specific information relevant to cooperative functionalities. ASIL of this function depends on application.
Driver Intention Estimation and Trajectory Prediction	Internal Function	E	N, O	Since there could exist some vehicles not cooperating, this function estimates those vehicles intention and their possible trajectory.
Security Management	Internal Function	A*, F*, G*	A*, F*, G*	Monitors the incoming communication to avoid malicious attacks. It could include cryptographic for authentication of ITS and also other peers. The level of security will be variant.
Collision Risk Assessment	Internal Function	C, D, E, N	Collision Risk Model	Using the information collected by Environment model and also trajectories intended by other vehicles, this function assesses the collision risk and provides the model to the Decision Making unit. This function shall be provided with ASIL D.
Data Fusion	Internal Function	B*, L, T	B*	Receives the data from different resources and combines them to have high level usable reliable information. It considers the data quality to achieve a result with acceptable confidence.
Decision Making	Internal Function			It's responsible for negotiation with other vehicles and design of a trajectory to be followed by the vehicle.
Trajectory Planning	Internal Function	E, P, selected primitive, LoP,	D	Regarding the selected primitive and considering the environment model, LoP and other vehicles trajectory, produces and updates a trajectory that shall be followed

		Collision risk model, Driver setting		by the vehicles. It also negotiates with other vehicles to agree on the proposed trajectory. This function shall be provided with ASIL D.
Primitives Priority Manager	Internal Function	U*, E, P, LoP, Collision risk model, feasible primitives , Driver settings	U*, selected primitive	Among the different feasible primitives, it selects the one with highest priority regarding the collision risk model, LoP and also parameters given in design time or settings by the driver. It negotiates with other cooperative vehicles to agree on the primitives to be executed by each of them.
LoP Evaluator	Internal Function	E	LoP	Evaluates all the available information to calculate the Level of Performance (LoP). LoP includes levels of both Controllability of the vehicle and Determinability of information. This function shall be provided with ASIL D.
Candidate Driving Primitive Selector	Internal Function	Q, R, T, LoP	Feasible primitives	Since there are specific requirements for different driving primitives regarding information availability this function is required to filter the unfeasible primitives.
On-board diagnostics	Internal Function	S	T	On-board Diagnostics monitors the on-board sensors and systems to detect any malfunction. It receives the feedbacks about systems malfunctions and also data quality.
Knowledge Base	Internal Function		Q, R	This function includes the functional related rules as algorithms for cooperative functions and also their requirements.
Remote Sensing	Interface Function	A*	A*	It includes the infrastructure and other vehicles which share their captured information. For those information it's assumed that there's a remote sensor interacting via wireless communication.
Peer Planner	Interface Function	F*, G*	F*, G*	The other vehicles function which is responsible for negotiating on driving primitives and trajectories to be followed.
On-board Sensing	Interface Function		L	The onboard sensors are installed on the vehicle. They would provide data quality index as well as data itself.
Actuators and Vehicle Systems	Interface Function	D		It's responsible for interpreting and executing the commands given as trajectory.
HMI	Interface Function	D, O, K	H, I, J	Human Machine Interface. Interacts driver intention and displays the environment model captured by the vehicle and trajectories to be followed.
Navigation System	Interface Function	H	P	Using the destination given by the driver and his preferences, it plans a path to be followed.

The data exchanged between so-called functional elements, are better described in Table 3.

Table 3: Primary functions interactions. The starred* links are bidirectional.

Link Name	Source	Destination	Data / Information
A*	Remote Sensing	• Communication	Local map, vehicles and their status, obstacles, road conditions, road obstructions, etc. Including event-driven communication
B*	Environment Modeling::DataFusion	• Communication	Local Dynamic Map
C	Communication	• Collision Risk Assessment	Other Vehicles Planned Trajectories
D	Decision Making::Trajectory Planning	• Environment Model • Collision Risk Assessment • Actuators and Vehicle Systems • HMI • Communication	Ego Trajectory
E	Environment Modeling	• Driver Intention Estimation and Trajectory Prediction • Collision risk assessment • Decision Making::Trajectory Planning • Decision Making::LoP Evaluator • Decision Making::Primitives Priority Manager	A view of the environment model specific to each function/application
F*	Peer Planner	• Communication	Selected Primitive to negotiate
G*	Peer Planner	• Communication	Intended Trajectory to negotiate
H	HMI	• Navigation System	Driver Intended destination
I	HMI	• Environment Modeling	Driver settings

J	HMI	• Decision Making::Trajectory Planning • Primitives priority manager • Candidate driving primitive selector	Driver settings
K	Environment Modeling	• HMI	Scenario
L	On-board Sensing	• Data Fusion	Quality / Confidence of Information
N	Driver Intention Estimation and Trajectory Prediction	• Collision Risk Assessment	Predicted / Estimated Trajectory of other vehicles
O	Driver Intention Estimation and Trajectory Prediction	• HMI	Predicted / Estimated Trajectory of other vehicles
P	Navigation System	• Decision Making::Primitives Priority Manager • Decision Making::Trajectory Planning	Path to be followed by the vehicle
Q	Knowledge Base	• Decision Making::Candidate Driving Primitive Selector	Cooperative driving algorithms parameters and requirements
R	Knowledge Base	• Decision Making::Candidate Driving Primitive Selector	Automatic driving algorithms parameters and requirements
S	On-board Sensing	• On-board Diagnostics	Sensors statues
T	On-board diagnostics	• Date Fusion • Decision Making::Candidate Driving Primitive Selector	Onboard sensors and systems statues
U*	Decision Making::Primitives Priority Manager	• Communication	Negotiation for the primitive to select
Selected Primitive	Decision Making::Primitives Priority Manager	• Decision Making::Trajectory Planning	Selected primitive
LoP	Decision Making::LoP Evaluator	• Decision Making::Candidate Driving Primitive Selector	Level of Performance

		• Decision Making::Primitives Priority Manager • Decision Making::Trajectory Planner	

According to ISO 26262 Concept Phase, the next step after item definition is the initiation of the safety lifecycle. The main objective of this part is to distinguish between a new item development and modification of an existing one. However, since the KARYON project concerns a completely new development, referring to ISO 26262 part 3-6.4.1.1 there is no much things to do in this part and the development shall continue with the Hazard Analysis and Risk Assessment (HARA).

5.2 Hazard Analysis and Risk Assessment

The Hazard Analysis and Risk Assessment (HARA) is a process dedicated to identification of the malfunctions resulting in hazards. In order to avoid unreasonable risks, the hazards are classified to formulate safety goals for mitigation or prevention of hazardous events. During HARA all of such failures shall be identified and mitigated, but here in KARYON the novelty of the work relates to communication and cooperation. So only some samples of the failures which affect the concerning functionalities are analyzed.

In Table 4 some scenarios are defined in which specific failures can cause identified hazardous events. Applying the requirements of ISO 26262-3 clause 7-HARA, for each case the ASIL is determined, and then the safety goal and safe state are defined.

Table 4: Hazard Analysis and Risk Assessment (HARA). Three of the safety goals are numbered for later references.

Speed	Maneuver/ Function	Malfunction/ Failure	Hazard	Severity (S0-S3)	Controllability (C0-C3)	Exposure Time (E0-E4)	ASIL	Safety Goal	Safe State
high	Platooning	V2V Short Range Communication with Latency	Tailing vehicle loses synchronization with leading vehicles	S3	C3	E4	D	Increase forehead distance and inform other vehicles to stop platooning operation (1)	Switch to Adaptive Cruise Control
low	Joining Platoon	V2V Short Range Communication with Latency	Joining vehicle loses synchronization with platooning vehicles	S1	C2	E4	A	Keep the current lane and stop platoon joining operation	Switch to Adaptive Cruise Control
medium	Joining Platoon	V2V Short Range Communication with Latency	Joining vehicle loses synchronization with platooning vehicles	S2	C3	E4	C	Keep the current lane and stop platoon joining operation	Switch to Adaptive Cruise Control
high	Joining Platoon	V2V Short Range Communication with Latency	Joining vehicle loses synchronization with platooning vehicles	S3	C3	E4	D	Keep the current lane and stop platoon joining operation	Switch to Adaptive Cruise Control

high	Platooning	Onboard Distance Sensors Failure (e.g. LIDAR)	The safe forehead distance is not respected	S3	C3	E4	D	Slight Deceleration and inform other vehicles to stop platooning operation (2)	Switch to Adaptive Cruise Control
low	Joining Platoon	Onboard Distance Sensors Failure (e.g. LIDAR)	The safe lateral distance is not respected	S1	C2	E4	A	Keep the current lane and stop platoon joining operation	Switch to Adaptive Cruise Control
medium	Joining Platoon	Onboard Distance Sensors Failure (e.g. LIDAR)	The safe lateral distance is not respected	S2	C3	E4	C	Keep the current lane and stop platoon joining operation	Switch to Adaptive Cruise Control
high	Joining Platoon	Onboard Distance Sensors Failure (e.g. LIDAR)	The safe lateral distance is not respected	S3	C3	E4	D	Keep the current lane and stop platoon joining operation	Switch to Adaptive Cruise Control
low	Entering Roundabout	Missing/Interrupted V2I Communication	Entering critical region asynchronously	S2	C1	E4	A	Decelerate before entering to roundabout and stop automated driving (3)	Switch to Manual Mode
medium	Entering Roundabout	Missing/Interrupted V2I Communication	Entering critical region asynchronously	S3	C2	E4	C	Decelerate before entering to roundabout and stop automated driving	Switch to Manual Mode
high	Entering Roundabout	Missing/ Interrupted V2I Communication	Entering critical region asynchronously	S3	C3	E4	D	Decelerate before entering to roundabout and stop automated driving	Switch to Manual Mode

As it was expected the HARA results proves that the failures of communication while running a cooperative function, can lead to an ASIL D hazardous events.

5.3 Functional Safety Concept

The safety goals and safe states are the root requirements for deriving other functional safety requirements. For each safety goal at least one functional safety requirement shall be specified which then shall be assigned to item elements.

In this section the starting point consists 3 of the safety goals resulted from HARA which are numbered in Table 4.

5.3.1 Functional Requirements

The safety goal and safety states are the root of functional requirements derivation. At least one functional safety requirement shall be specified for each safety goal but it's a one-to-many relationship meaning that a functional safety requirement can be valid for several safety goals.

In the following figures the requirement diagrams are presented in which the fault tolerant time interval, operating mode and other functional safety requirement are derived from safety goals and safe states. Then the functional safety requirements are assigned to the preliminary architecture elements and also the new elements that should be contained in the architecture satisfying those requirements.

Figure 10 corresponds to the safety goal "increase forehead distance and inform other vehicles to stop platooning operation" that should be followed by transition to Adaptive Cruise Control as safe state.

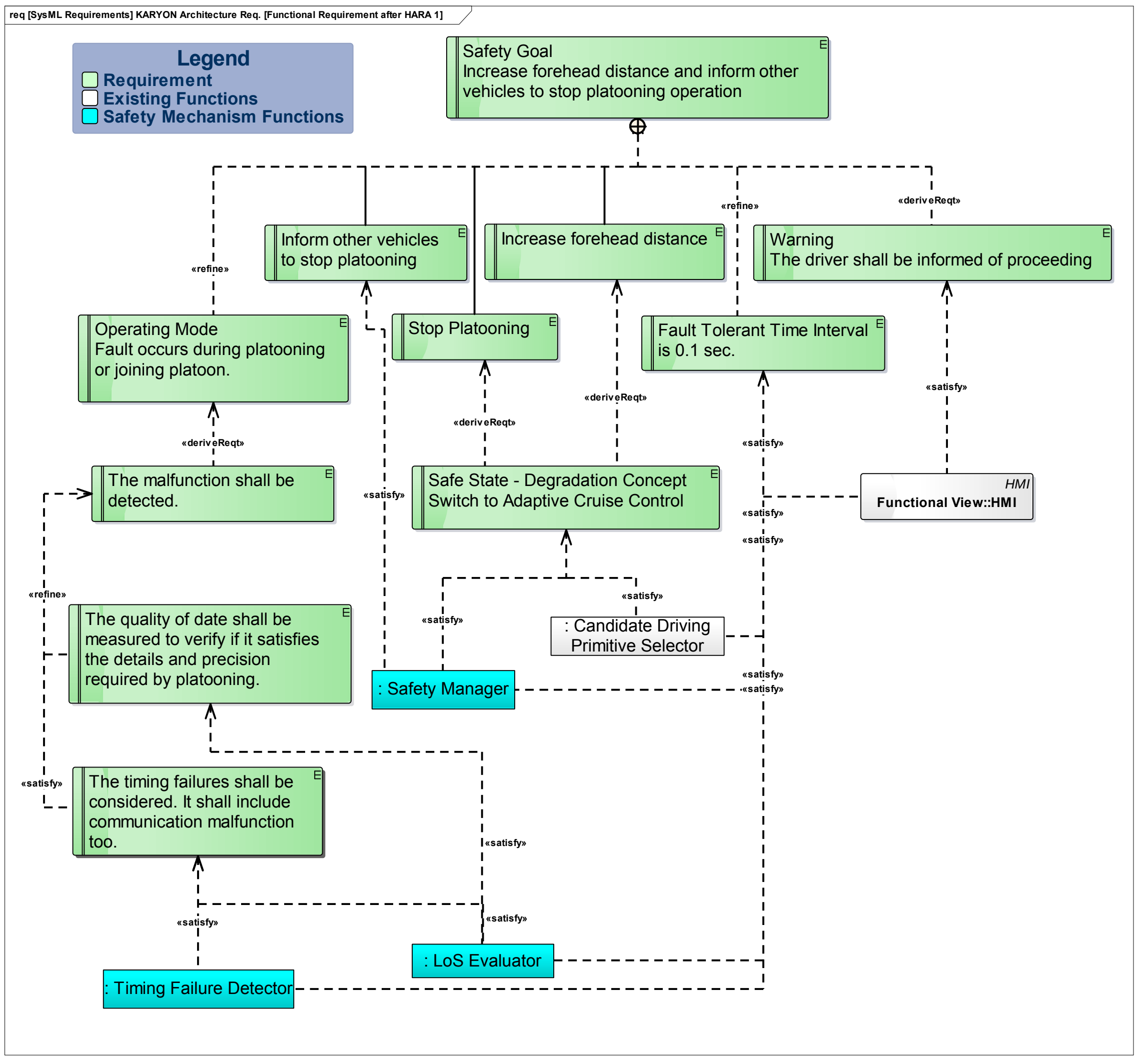

req [SysML Requirements] KARYON Architecture Req. [Functional Requirement after HARA 1]
Legend
Requirement
Existing Functions
Safety Mechanism Functions
Safety Goal
Increase forehead distance and inform other vehicles to stop platooning operation
Inform other vehicles to stop platooning
Increase forehead distance
Warning
The driver shall be informed of proceeding
«refine»
«deriveReqt»
Operating Mode
Fault occurs during platooning or joining platoon.
Stop Platooning
Fault Tolerant Time Interval is 0.1 sec.
The malfunction shall be detected.
Safe State - Degradation Concept
Switch to Adaptive Cruise Control
HMI
Functional View::HMI
«satisfy»
The quality of date shall be measured to verify if it satisfies the details and precision required by platooning.
: Candidate Driving Primitive Selector
: Safety Manager
The timing failures shall be considered. It shall include communication malfunction too.
: LoS Evaluator
: Timing Failure Detector

Figure 10: Functional safety requirements specification and assignment for safety goal 1

Figure 11 corresponds to the safety goal "slight deceleration and inform other vehicles to stop platooning operation" that should be followed by transition to Adaptive Cruise Control as safe state.

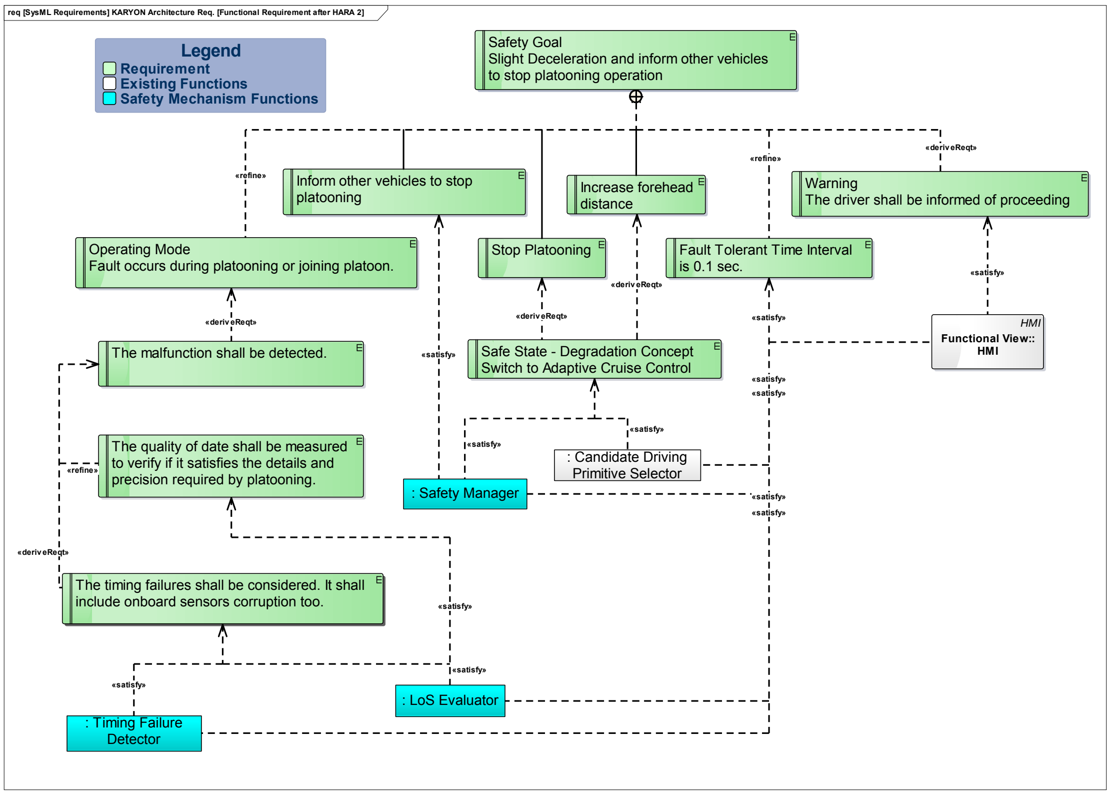

req [SysML Requirements] KARYON Architecture Req. [Functional Requirement after HARA 2]
Legend
Requirement
Existing Functions
Safety Mechanism Functions
Safety Goal
Slight Deceleration and inform other vehicles to stop platooning operation
«refine»
«deriveReqt»
«refine»
Inform other vehicles to stop platooning
Increase forehead distance
Warning
The driver shall be informed of proceeding
Operating Mode
Fault occurs during platooning or joining platoon.
Stop Platooning
Fault Tolerant Time Interval is 0.1 sec.
«deriveReqt»
«satisfy»
«satisfy»
HMI
Functional View:: HMI
The malfunction shall be detected.
«satisfy»
Safe State - Degradation Concept
Switch to Adaptive Cruise Control
«satisfy»
The quality of date shall be measured to verify if it satisfies the details and precision required by platooning.
«refine»
«satisfy»
: Candidate Driving Primitive Selector
: Safety Manager
«deriveReqt»
The timing failures shall be considered. It shall include onboard sensors corruption too.
«satisfy»
: LoS Evaluator
: Timing Failure Detector

Figure 11: Functional safety requirements specification and assignment for safety goal 2

Figure 12 corresponds to the safety goal “decelerate before entering to roundabout and stop automated driving” that should be followed by transition to manual mode as safe state.

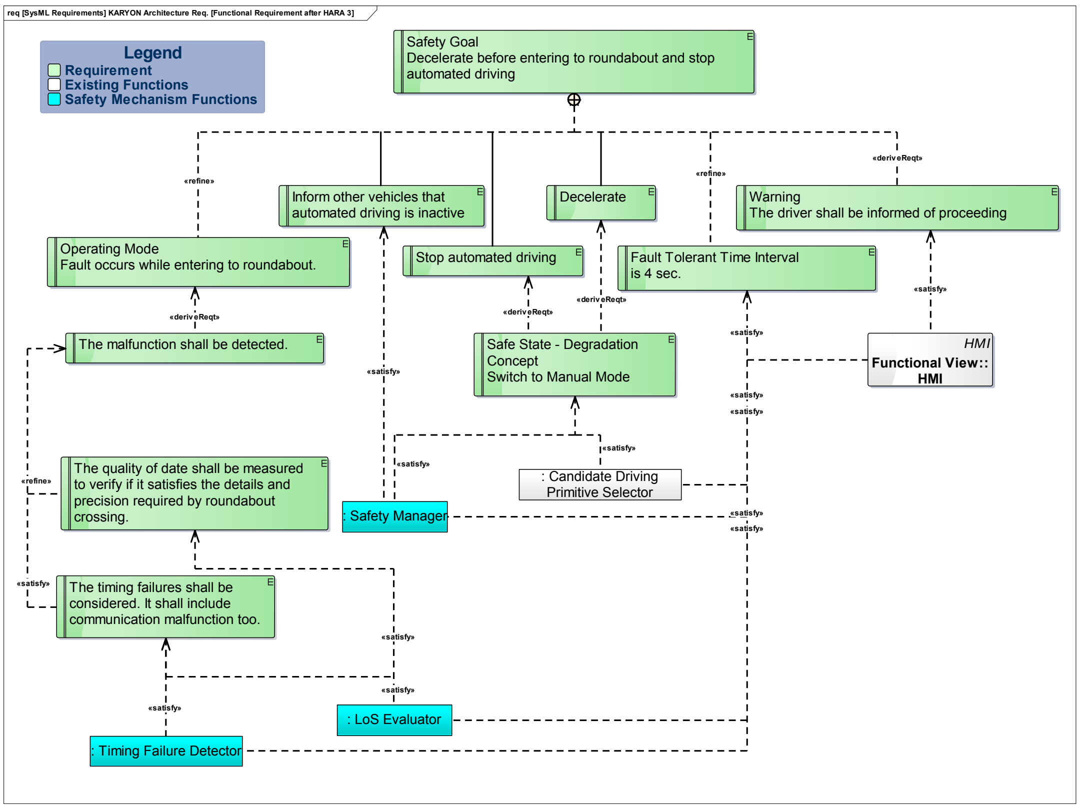

req [SysML Requirements] KARYON Architecture Req. [Functional Requirement after HARA 3]
Legend
Requirement
Existing Functions
Safety Mechanism Functions
Safety Goal
Decelerate before entering to roundabout and stop automated driving
«refine»
«refine»
«deriveReqt»
Inform other vehicles that automated driving is inactive
Decelerate
Warning
The driver shall be informed of proceeding
Operating Mode
Fault occurs while entering to roundabout.
Stop automated driving
Fault Tolerant Time Interval is 4 sec.
«deriveReqt»
«deriveReqt»
«satisfy»
The malfunction shall be detected.
Safe State - Degradation Concept
Switch to Manual Mode
«satisfy»
HMI
Functional View:: HMI
«satisfy»
The quality of date shall be measured to verify if it satisfies the details and precision required by roundabout crossing.
«refine»
: Candidate Driving Primitive Selector
: Safety Manager
«satisfy»
The timing failures shall be considered. It shall include communication malfunction too.
: LoS Evaluator
: Timing Failure Detector

Figure 12: Functional safety requirements specification and assignment for safety goal 3

Having an abstract view over mentioned safety goals and requirement diagrams a unique general solution is provided as safety mechanism. This solution contains the warning and degradation concept and introduces the Level of Service (LoS) as a fundamental concept in KARYON approach.

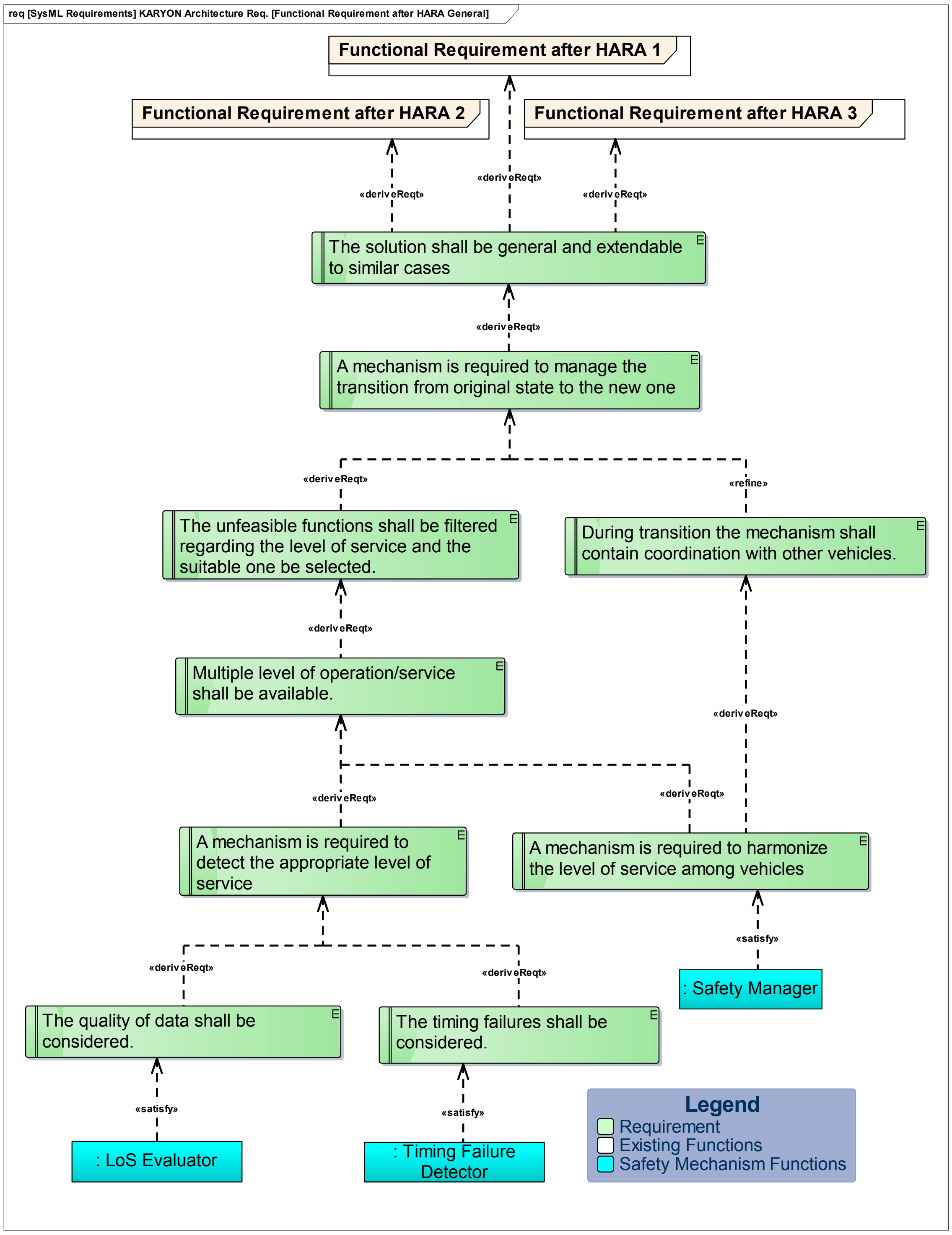

req [SysML Requirements] KARYON Architecture Req. [Functional Requirement after HARA General]
Functional Requirement after HARA 1
Functional Requirement after HARA 2
Functional Requirement after HARA 3
«deriveReqt»
The solution shall be general and extendable to similar cases
A mechanism is required to manage the transition from original state to the new one
«refine»
The unfeasible functions shall be filtered regarding the level of service and the suitable one be selected.
During transition the mechanism shall contain coordination with other vehicles.
Multiple level of operation/service shall be available.
A mechanism is required to detect the appropriate level of service
A mechanism is required to harmonize the level of service among vehicles
«satisfy»
: Safety Manager
The quality of data shall be considered.
The timing failures shall be considered.
: LoS Evaluator
: Timing Failure Detector
Legend
Requirement
Existing Functions
Safety Mechanism Functions

Figure 13: Overall functional safety requirements diagram introducing Safety Kernel functions and Level of Service (LoS) concept

5.3.2 The Architecture Containing Safety Mechanism

Satisfying the functional safety requirements already defined, the safety mechanism is introduced. The safety mechanism is an extension to the preliminary architecture provided in abstract item definition.

In KARYON architecture, the functional elements that should take the responsibility of so-called requirements satisfaction are: timing failure detection, LoS evaluator and safety manager. These functions together compose the Safety Kernel. Figure 14 illustrates how they interact with each other and also with existing item elements to satisfy the corresponding functional safety requirements.

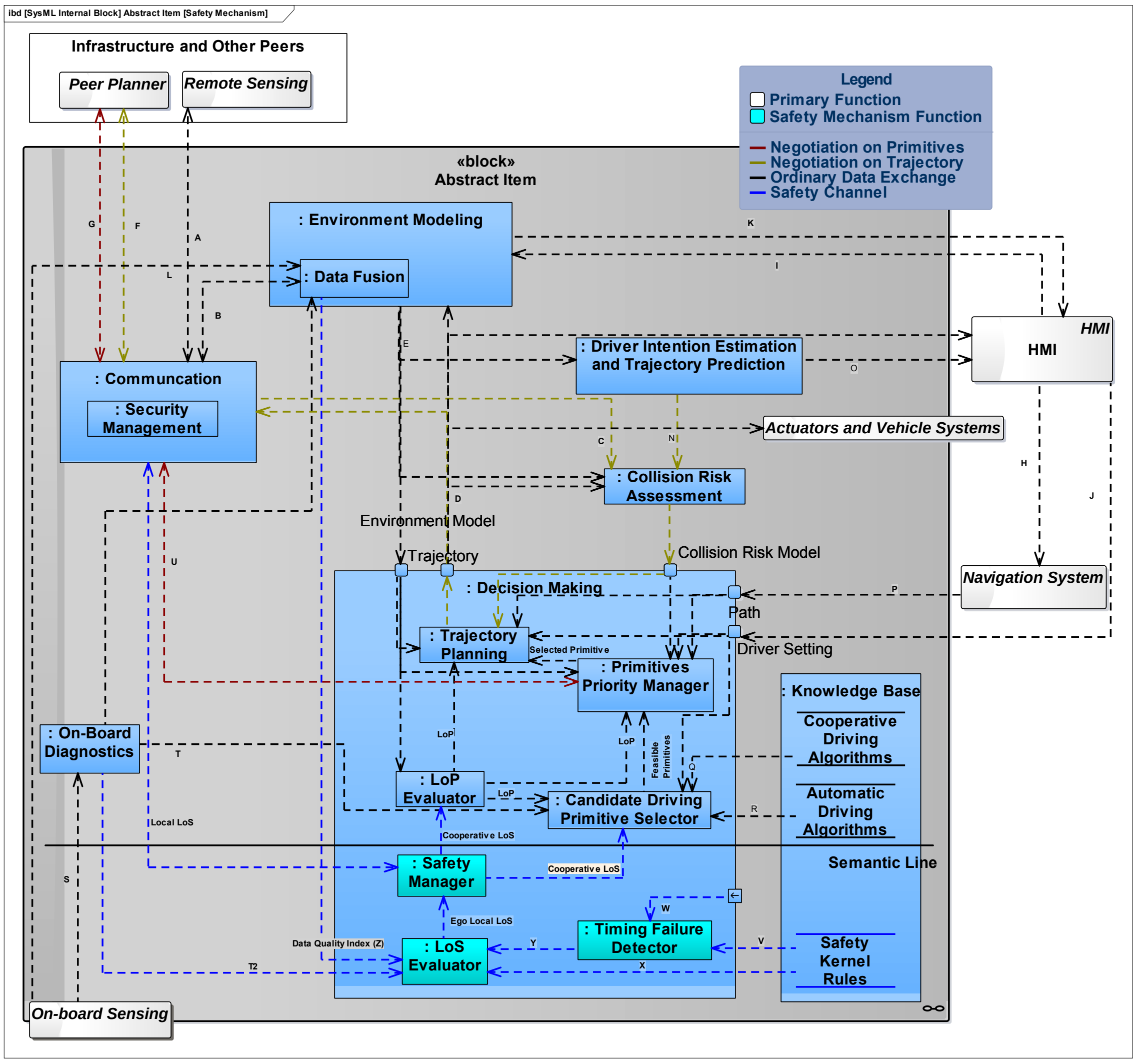


Figure 14: Safety mechanism as an extension to the preliminary architecture

Table 5 provides a brief description of each Safety Kernel function and also those primary functions whom their operation is modified in the new architecture.

Table 5: Safety Kernel functions. The starred* functions and I/O are added as part of safety mechanism and safety channel.

Function / Element Name	Type	Input	Output	Task
LoP Evaluator	Internal Function	E, Cooperative LoS*	LoP	Evaluates all the available information to calculate the Level of Performance (LoP). LoP includes levels of both Controllability of the vehicle and Determinability of information. This function shall be provided with ASIL D.
Candidate Driving Primitive Selector	Internal Function	Q, R, T, LoP, Driver settings, Cooperative LoS*	Feasible Primitives	Since there are specific requirement for different driving primitives regarding information availability this function is required to filter the unfeasible primitives.
Safety Manager*	Internal Safety Mechanism Function	Other Vehicles Local LoS*, Ego Local LoS*	Local LoS*, Cooperative LoS*	Collects the local LoS of other vehicles and decides on acceptable LoS. The output is called Cooperative LoS that shall be used as basic parameter of decision making.
LoS Evaluator*	Internal Safety Mechanism Function	Data Quality Index (Z)*, Y*, X*, T2	Ego Local LoS*	Evaluates the local Level of Service which belongs to the vehicle independent of other vehicles. It's just based on on-board systems state and sensors data quality. This function shall be provided with ASIL D.
Timing Failure Detector*	Internal Safety Mechanism Function	W*, V*	Y*	Monitors data arrival times such as data received from communication or onboard sensors. It detects the failures according to rules defined in knowledge base.

The data exchanged between new functional elements are better described in Table 6.

Table 6: Safety mechanism functions interactions. The starred* links are bidirectional.

Link Name	Source	Destination	Data / Information
Cooperative LoS	Safety Manger	• Candidate driving primitive selector • LoP evaluator	The cooperative level of service calculated on the basis of local level of service of ego and other vehicles
Local LoS*	Safety Manager	• Communication	Local level of service calculated on the basis of quality indexes applying the Safety Kernel rules
Data Quality Index	Data Fusion	• LoS evaluator	The indexes that determine how much the data forming environment model are precise

T2	On-board diagnostics	• LoS evaluator	Status of onboard systems and sensors
V	Safety Kernel rules	• Timing failure detector	Safety Kernel rules in concern of timing failures
W	All the real time functions	• Timing failure detector	Heartbeat of real time functions
X	Safety Kernel rules	• LoS evaluator	Safety Kernel rules in concern LoS evaluation
Y	Timing Failure Detector	• LoS evaluator	Timing failures

In addition, in order to have a better measurement and demonstration of the effectiveness of the Safety Kernel mechanism in mitigating faults, a FTA analysis is performed. FTA is a deductive top-down approach in which an undesired state of the system is analysed by combining a series of lower level events using Boolean logic. For such purpose:

- The top event shall be defined which is the violation of a safety goal.
- The basic events shall be identified which are the source of the failures. They could refer to internal faults, or external ones propagated through inputs.
- The loops in the system shall be eliminated by removing backward flows.
- The “and” and “or” gates shall be specified which determine how the faults coming from different paths should be combined.

In our case, the top event is defined as “The safe forehead distance is not respected” which is violation of an ASIL D safety goal. The basic faults are introduced for both on-board and remote sensors in addition to the communication function. The failures caused by other faults in the system has been out of scope of this approach and so are not modelled. The analysis is performed using a specific ASIL Decomposition tool by 4S Group. This tool demonstrates the fault propagation through the system, but also calculates the ASIL for each element in the faults route according to ISO262626 ASIL decomposition rules.

The result of the analysis has not been far from the expectation. The Safety Kernel functions are assigned ASIL D. The on-board sensing is assigned ASIL C and it’s in accordance with the fact that the vehicle shall be able to operate safely even in the absence of remote information. In compliance with this rational, the communication and remote sensors are assigned QM as expected. The “Data Fusion” which is a critical function is assigned ASIL C too.

5.4 Safety Kernel SEooC Concept

A SEooC is based on assumptions. These assumptions define the purpose, functionality and external interfaces of the SEooC. Then while integrating the SEooC in the context of the actual item, the validity of those assumptions must be established [21]. The aim so far has been to derive Safety Kernel functional aspects based on some primitive assumptions. The following concludes those initial assumptions and their results from SEooC point of view.

Purpose and functionality: A part of the assumptions relate to the functionality of the SEooC. As it’s been described the Safety Kernel is purposed as a safety mechanism that mitigates hazards caused by the failures corresponding to cooperative driving functions.

Functional requirements: To define a SEooC in a high level of abstraction it's necessary to introduce its functional requirements. The functional requirements of the Safety Kernel SEooC are in fact those requirements that are identified in Section 5.3.1. Figure 10 to Figure 13 illustrate the reasoning process of derivation of those requirements starting from the safety goals. In addition three functions (safety manager, LoS evaluator, and timing failure detector) are introduced which satisfy those requirements. Figure 14 and Figure 15 display these functions and their interactions composing Safety Kernel.

Assumptions external to SEooC: Another set of the assumptions required to define a SEooC, consists those requirements and assumptions made on the elements which are out of the boundary of the SEooC but have interactions with it. In this approach, a major set of the assumptions external to Safety Kernel SEooC is given by abstract item definition in Section 5.1. Table 2, Table 3, Table 5 and Table 6 in addition to Figure 14 provide the description of functional elements external to Safety Kernel functions. For a better understanding of the Safety Kernel SEooC scope, Figure 15 illustrates its boundary and interfaces with external elements.

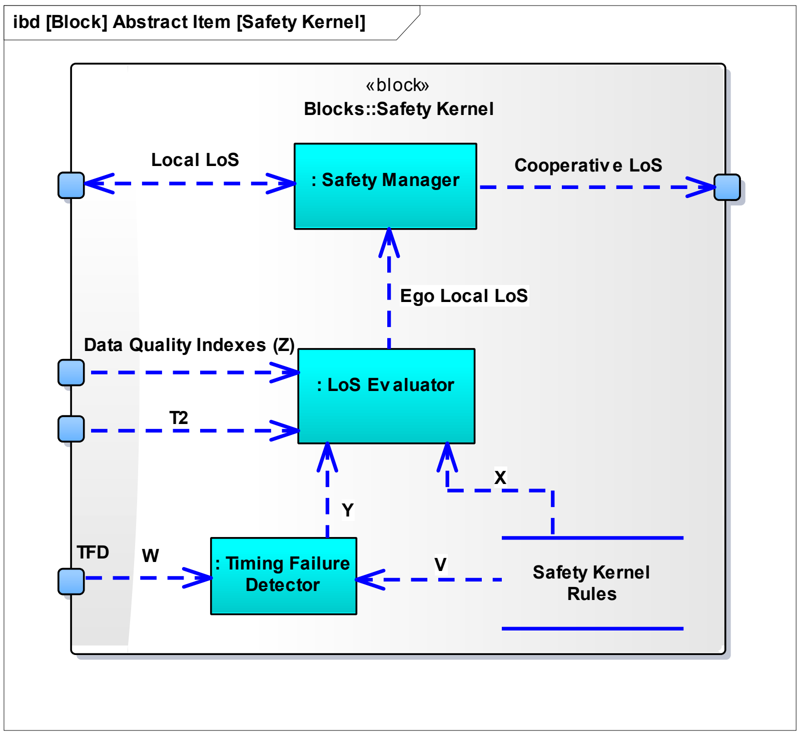


Figure 15: Safety Kernel boundary, interactions and interfaces. The description of the labeled interactions is available in Table 5 and Table 6.

6. Components design and configuration

This section focusses on the design and the implementation of an engine that performs the run-time verification of safety requirements expressed in safety rules. This engine is one of the main components the Safety Kernel and hence has to perform the verification in a timely and efficient way. In addition, we also devised a solution that deals with scalability issues and may thus be useful for complex systems, involving a large number of safety rules. This section explains how the safety rules can be expressed using the XML notation, how they are parsed and stored in memory and what is the algorithm performed by the safety manager engine to evaluate safety based on collected safety information.

6.1 Architecture overview

Figure 16 gives an overview of the Safety Kernel components. At start-up the *XML Parser* reads the local configuration, builds a *Safety Rules* repository and initializes *Run-time Safety Information* (*RSI*) structures. Therefore, the configuration file includes both safety rules and *unit* definitions. A unit corresponds to a Safety Kernel input (collected data), output (adjustment data – typically a component performance level) or locally calculated values (for instance, the acceptable LoS for some function).

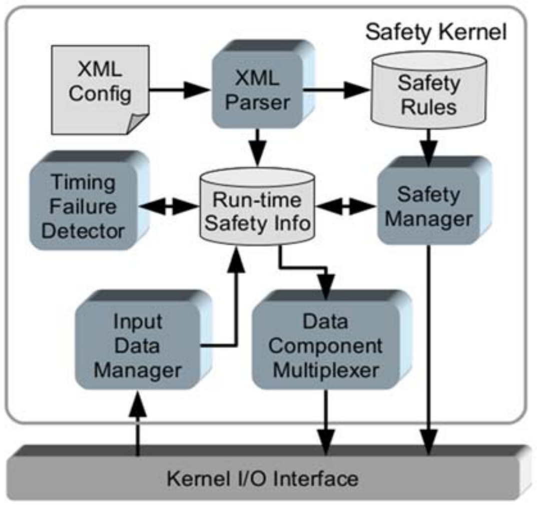


Figure 16: Safety Kernel components.

A safety rule is a Boolean expression involving combinations of static values (bounds) and unit identifiers. A safety rule is meaningful for a specific LoS of some function. For instance, function 2 can only be safely executed in LoS 1 when data validity V_0 is greater than 50 and data validity V_1 is greater than 70. This is expressed as:

$$F_2(LoS1) \rightarrow V_0 > 50 \wedge V_1 > 70$$

The *Input Data Manager* receives data inputs from the external (nominal system) components and updates the RSI. The *Timing Failure Detector* (TFD) is responsible for checking if certain data inputs have been received from external components within predefined temporal bounds. This TFD executes periodically, during each execution round of the Safety Kernel. When the TFD detects a timing failure (i.e., some data, which might be just a heartbeat, has not been timely produced at the Safety Kernel interface), it stores this information in the RSI unit corresponding to the untimely data. The *Data Component Multiplexer* selects, from two or more data inputs (collected from nominal components), one that is forwarded to its output. This is useful, for instance, when the nominal function has two components providing the same data (e.g., a front distance value), one providing data with high validity, but taking an uncertain amount of time to produce this data, and the other providing data with lower validity, but always in a timely way. The Data Component Multiplexer selects, among the two values, the better one, if timely produced, and the lower validity one, otherwise. Finally, the Safety Manager is the central component as it evaluates at run-time if *Safety Rules* are satisfied given the RSI data.

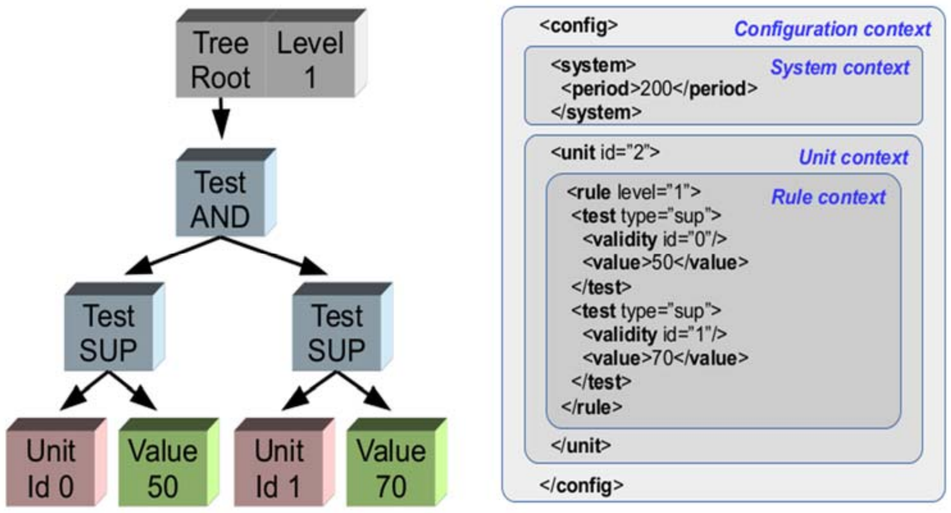
6.2 Design and Implementation

In this section, we start by describing how the *Run-time Safety Information* and the *Safety Rules* are represented in memory. Then we explain the solution for parsing and storing safety rules. We continue by addressing the unit types and the three main Safety Kernel modules: The *Timing Failure Detector*, the *Safety Manager* and *the Data Component Multiplexer*.

6.2.1 Data Structures

Data structures must be simple to provide code robustness, but they are designed as well with the aim of reducing the computation time during the rule evaluation phase. The Run-time Safety Information (RSI) repository is initialized during system bootstrap and is updated at run-time with collected safety-related information. The RSI size depends on the number of units (inputs, outputs and internal variables) declared in the configuration file. As this size is not changed at run-time, we use a single dimension array to store the units. Each unit structure contains several fields, including a pointer to related safety rules, which set requirements on this unit, a timeliness status, which may be relevant for units with timeliness constraints, a data validity value, a level value that may be used to store performance levels or levels of service (this is clarified ahead in the text), and some other attributes.

The safety rules are also built at bootstrap from the configuration file. We note that one possible design approach would be to simply hard code the safety rules within the Safety Kernel, thus avoiding the need for specifying them in a configuration file, and consequently processing them at bootstrap. However, we decided to follow an approach that provides some additional flexibility and leads to a generic Safety Kernel implementation. Safety rules can be updated without the need for recompiling the code and loading it on the board, which is particularly advantageous during the development process. And the Safety Kernel core is totally independent of the specific application, which can facilitate verification and validation activities.



```
<config>
  <system>
    <period>200</period>
  </system>
  <unit id="2">
    <rule level="1">
     <test type="sup">
       <validity id="0"/>
       <value>50</value>
     </test>
     <test type="sup">
       <validity id="1"/>
       <value>70</value>
     </test>
    </rule>
  </unit>
</config>
```

Figure 17: Basic safety rule definition.

Given that safety rules need to be checked in every execution cycle, within a limited amount of time, a fundamental requirement is to devise a solution for storing them in memory, such that safety management is efficient and scalable, as well as time bounded. This is particularly necessary if dealing with complex systems, in which the number of safety rules will tend to be very high. The concrete number will depend on the amount of functionalities that may be performed by an autonomous vehicle, on the number of system variables that may have to be checked in run-time, and on the number of levels of service considered for each functionality. We addressed this requirement by adopting a tree-based data organization, where the root node for each safety rule contains the associated LoS and a pointer to the top child node of the tree. This tree is created during the XML Parsing. This kind of structure allows for efficient rule parsing at run-time, using the algorithm described in Section 6.2.5.

The tree corresponding to the basic rule example from Section 6.1 is shown in Figure 17 (left). Three different types of nodes can be used in the tree: test nodes, unit id nodes and value nodes. Test nodes store Boolean operations, like *AND*, *OR*, *EQUAL*, *DIFF* or *SUP*, among others. Each unit id node contains the index of a unit in the RSI array. According to the way a unit id is defined in the configuration, it contains either a data validity value or a level of service/performance level value. In the example, the two units (ids 0 and 1) will contain data validity values. Different rule trees can refer to a single unit when there are multiple constrains (safety rules) related to a certain safety-related variable. When this happens, the several rule trees are level-sorted (from the higher to the lower level, as defined in the root node) in a linked list to which the unit will point. Finally, value nodes contain constants (bounds) against which the unit values will be checked.

6.2.2 XML Parsing

A lot of *XML Parsers* are described in the literature and many of them are available for free. These parsers usually offer a large range of functionalities and may be not portable to RTEMS environments. As a consequence, we chose to develop our light XML Parser with only some basic features. Its architecture is similar to the open mark-up parser available in the GLib library (Gnome low-level C library) [3].

The *XML Parser* is a simple context-based parser. In each context there are two call-back functions that are used for opening and closing mark-up tags. During the parsing, these functions are called to initialize the RSI array and the safety rule trees. Context switching is performed inside the call-back function according to the parsed XML tag.

Figure 17 (right) shows an XML configuration file implementing the basic safety rule example given in Section 6.1. A configuration file admits four context levels. The *configuration context* is the default one, while the *system context* is used to initialize global system parameters. For instance, in this example the Safety Kernel period is set to 200 ms. The purpose of the *unit context* is to define a new unit. Finally, the *rule context* is used to build a safety rule tree associated to a given unit. Note that besides the output unit with id 2 (which allows to set the performance level of some application component), two additional units are created (ids 0 and 1) to store data validity values. The output unit will hold the value 1 when the (only) rule evaluates to true, and 0 otherwise. A node stack allows to internally store the nodes and assemble the tree.

6.2.3 Unit inputs and outputs

The Safety Kernel asynchronously receives the unit inputs through an *External Component Interface* (ECI). If an input update is sent by the same component before the previous input is processed, the old one is overwritten since the Safety Kernel keeps the more up-to-date information from a component.

The Safety Kernel runs periodically the *Time Failure Detector* (TFD), the *Safety Manager* (SM) and the *Data Component Multiplexer* (DCM) to update the output units. When all outputs have been processed, the Safety Kernel returns back these values to the ECI. The Safety Kernel can periodically send these output values or not depending on the sending mode configured for each output

The Safety Kernel is not event-triggered but time-triggered. In other words, the Safety Kernel manages two independent processes: the first one to receive the inputs and keep the more up-to-date values and the second one to check the unit timeliness, evaluate the rules and send the outputs. So in the worst case scenario, a period of X milliseconds (where X is the Safety Kernel period) can elapse between the reception of a validity update and the output of the corresponding LoS update.

The Safety Kernel uses different message types to implement unit inputs and outputs. For example, components located above the hybridization line are considered as not proven safe, which means their liveliness and timeliness have to be periodically verified by the TFD module. These components can send a *HEARTBEAT* message or the *VALIDITY* message (containing a data validity value) to signal the Safety Kernel that they are still alive. The *HEARTBEAT* message has been introduced due to the fact that some components may not be designed to periodically produce a certain output and its corresponding validity information.

The different message types are depicted in the figure below:

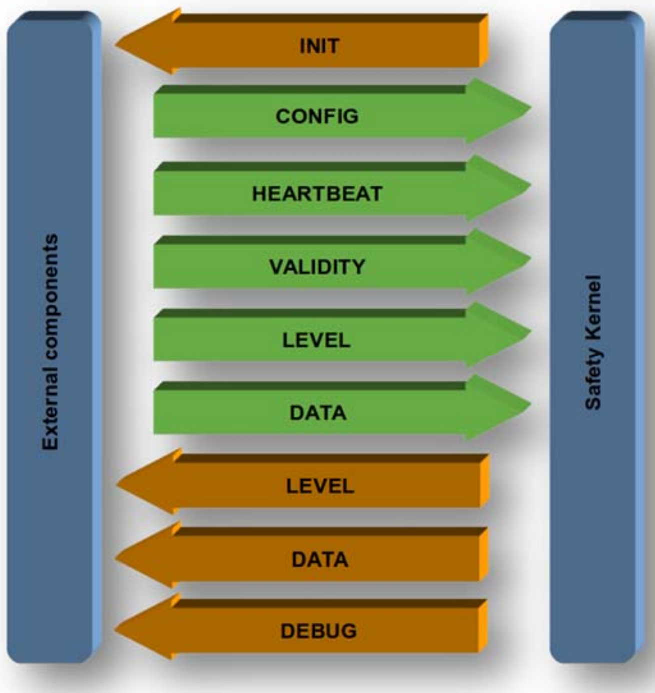


Figure 18: Message types.

Input messages:

- **CONFIG**: Used to remotely send a configuration content to the Safety Kernel at runtime. This might be the startup configuration if no local file was previously found. Note that this message is intended just for testing purposes. In run-time, a production system will never change the Safety Kernel configuration.
- **HEARTBEAT**: Used by a component above the hybridization line to signal the Safety Kernel that it is alive.
- **VALIDITY**: Used by a component above or below the hybridization line to provide a data validity value.

- **LEVEL**: Exclusively used by the *Cooperative LoS Evaluator,* which is above the hybridization line, as a means to indicate the agreed cooperative LoS. If this is not done within a defined amount of time, the Safety Kernel will consider that there was a failure of the Cooperative LoS Evaluator, and will always decide to execute all functionalities in the lowest LoS.
- **DATA**: Used by a multi-component source to send a data value to the Safety Kernel.

Output messages:

- **INIT**: Used to inform the ECI that the Safety Kernel has just started but no local configuration has been found. This message will be periodically sent (every second by default) until a remote configuration is received through the *CONFIG* message. This message is only relevant when testing the Safety Kernel.
- **LEVEL**: Used to indicate the required performance level of a nominal system component situated below the hybridization line. This kind of packet is also used to let the Cooperativel LoS Evaluator know about the level of service that was calculated inside the Safety Kernel. This LoS is usually broadcast to the other cooperative vehicles (through wireless network), and received by the respective Cooperative LoS Evaluators, which will decide the cooperative LoS value.
- **DATA**: Used to send back to the ECI the data issued by the selected component source.
- **DEBUG**: Used by the Safety Kernel to send a runtime warning message to the ECI. This message type can be used as well to provide the execution times at the end of the benchmarking phase. As the name indicates, the message is only relevant for testing purposes.

6.2.4 Timing Failure Detector

We can distinguish two types of input units according to their position in relation to the hybridization line. The ones below the line are enjoy real-time properties and thus always produce their output in a timely way. On the other hand, the execution time of components above the hybridization line will not necessarily satisfy timing requirements. For each of these latter units, the Safety Kernel keeps track of the inter-arrival time between consecutive input values. When an input value is received by the kernel, it is automatically time-stamped in the *RSI* array. A timeout will be associated to each of these units, so that a timing failure will be detected when the inter-arrival time exceeds this timeout.

At every Safety Kernel cycle, the RSI array is scanned and for each timeout-defined unit. The *Timing Failure Detector* checks whether the value was received within this timeout. If not, the input unit is set to “non-timely”. At the next cycle, the *TFD* will repeat the same verification and may change the input unit status to “timely” if a value was received before the timeout exceeded.

If the component producing the input value is very unstable, the corresponding unit status may continually change over time. In order to avoid this “flapping” phenomena, the Safety Kernel introduced two configuration attributes: the minimum number of required successes and the maximum number of tolerated failures, which have to be observed in a row, and which necessary to respectively declare the input unit as “timely” or “non-timely”. Both attributes can be globally defined for all timeout-defined values or separately defined for each unit.

6.2.5 Safety Manager

At run-time the *Safety Manager* will periodically scan the RSI array. For each unit with at least one defined rule (some units, like units 0 and 1 from the example, do not have any associated rule), the *Safety Manager* evaluates them starting with the rule with the highest level. The rationale is to first evaluate if the conditions to perform some function at the highest level of service are satisfied. When they are not, then other safety rules will be checked. Therefore, the evaluation stops when a rule is satisfied or when the end of the rule list is reached. In the latter case, this means that the function has to be executed at the lowest LoS (level 0). At the end of the process, the Safety Manager updates the level field of internal units (those holding the acceptable LoS for some function) and of output units (holding the performance level of specific components). The rule evaluation functions are the following:

```
 1: function LEVEL(rule_list)
 2:   for all rule in rule_list do
 3:     node_list ← rule.root
 4:     if AND(node_list) then
 5:       return rule.level
 6:     end if
 7:   end for
 8:   return 0
 9: end function
10:
11: function AND(node_list)
12:   for all node in node_list do
13:     if ¬EVAL(node) then
14:       return false
15:     end if
16:   end for
17:   return true
18: end function
19:
20: function EVAL(node)
21:   switch node.type do
22:     case test
23:       return TEST(node)
24:     end case
25:     case unit
26:       return UNIT(node)
27:     end case
28:     case value
29:       return true
30:     end case
31:   end switch
32: end function
33:
34: function TEST(node)
35:   node_list ← node.test.childs
36:   switch node.test.type do
37:     case sup
38:       if ¬AND(node_list) then
39:         return false
40:       end if
41:       return COMPARE(node) > 0
42:     end case
43:     …
44:   end switch
45: end function
46:
47: function UNIT(node)
48:   id ← node.unit.id
49:   unit ← unit_array[id]
50:   return unit.status
51: end function
```

The *level* function (line 1) evaluates the unit rule list. The *and* function is first called, as the top-level node is always an *AND* in any rule tree. This first node gathers all conditions required for the rule to be satisfied. The *eval* function (line 20) evaluates a node according to its type. In the *test* function (line 34) we only show the SUP operator (line 37). First we check the timeliness status of both operands by recursively calling the *and* function. If the evaluation returns true, we compare the values of both operands (line 41). The *unit* function (line 47) is called to evaluate a unit and returns its timeliness status.

6.2.6 Data Multiplexer Component

As explained in section 4, the *DCM* is a special component aimed to select and forward data from an input multi-component to an output component. The selected data source is timely received and is issued from the component implementation having the greatest performance level.

Among the available implementations, only one is proven-safe (below the hybridization line) and has the weakest performance level. Other implementations with higher performance levels are all located above the hybridization line and consequently might suffer failure or latency.

In the example shown in Figure 19, *C1* is a multi-component with three implementations *C1*, *C1'* and *C1''*. *C1* is proven-safe while *C1'* and *C1''* may provide data in a non-timely way. Let's assume that the performance level of *C1''* is greater than *C1'*. Finally *C2* is the output component where data must be forwarded.

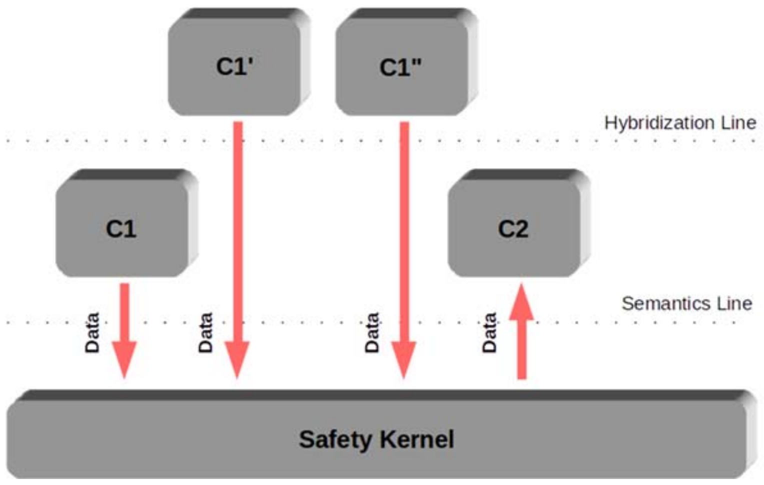


Figure 19: Example with multi-components.

In a concrete way, *C1*, *C1'*, *C1''* and *C2* are all defined as units. In the Safety Kernel configuration, we define timeouts for *C1'* and *C1''* (values can be different) and we configure *C2* as an output unit from a multi-component depending of three sources, *C1*, *C1'* and *C1''* with respectively 1, 2 and 3 as performance levels. These performance levels are arbitrary set and only aim to assign a priority to each unit source.

At each Safety Kernel cycle, *the Data Component Multiplexer* is started and scans the RSI array. For each output multi-component unit (i.e. *C2* in our example), the DCM logs the timely input unit with the greatest performance level. When all units have been scanned, the DCM returns the data value from the selected unit to the ECI. Consequently, the *Safety Manager* forwards at each cycle as many data values as output multi-component units.

In this example, the performance levels of each input units *C1*, *C1'* and *C1''* are static and hardcoded in the configuration. Consequently the selection can only change according to the timeliness of each input unit. In some situation, it could be interesting to update at runtime the level of service of the input units and that way, give priority to a component implementation. The *Data Component Multiplexer* provides such a functionality through the *Safety Manager*. Based on safety rules, the *SM* can update the performance level of the input unit in the same manner it updates any performance level or level of service. When the DCM scans the RSI array, it will use this new performance values to make the selection.

Let's come back to our example and now imagine that the performance of *C1'* depends on a sensor validity *V1*. If this validity is greater than 0.7, we give priority to *C1'* rather to *C1''*. Otherwise *C1''* must be used. The solution consists in building the following rule:

$$C1'(PL4) \rightarrow V1 > 70$$
$$C1'(PL2)\ otherwise$$

We set the performance level 2 as the default value for *C1''*. The performance level of *C1''* is 3 and remains static over time. So if both data values from *C1'* and *C1''* are timely received, the data source is selected by the Safety Kernel as follows:

$$V1 > 70 \Rightarrow C1'(PL4) \wedge C1''(PL3) \Rightarrow Data\ from\ C1'is\ selected$$
$$V1 \leq 70 \Rightarrow C1'(PL2) \wedge C1''(PL3) \Rightarrow Data\ from\ C1''is\ selected$$

6.3 Safety Kernel deployment

Two versions of the Safety Kernel are currently available: a non-real-time version for Unix/Linux environments and a real-time version built over the RTEMS operating system. On both versions, the Safety Kernel communicates with an *External Component Interface* (ECI) through UDP and RAW sockets, on dedicated physical links.

The ECI is a C library that may be used in the implementation of nominal system components, which facilitates the use of native primitives from the Safety Kernel API to send/receive messages to/from the Safety Kernel. In the default configuration, there is a single ECI.

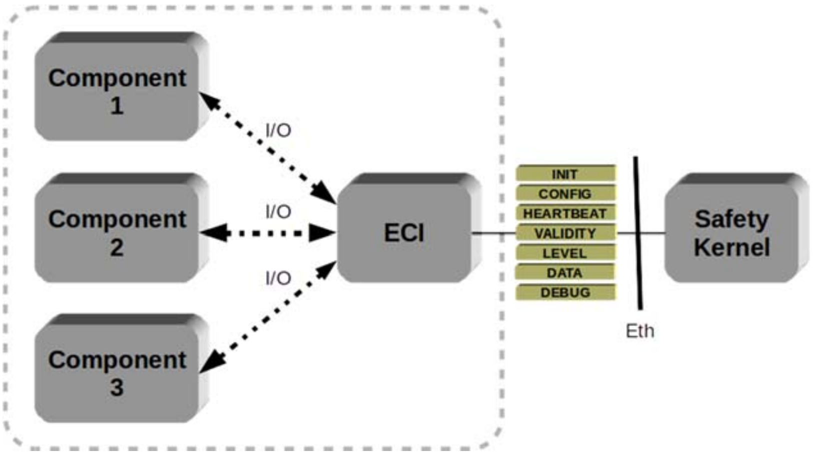


Figure 20: Single External Component Interface.

In some situations, it could be interesting to have several ECIs all connected through sockets with the Safety Kernel. Every ECI is able to send any kind of message to the Safety Kernel. The routing between the Safety Kernel and the right ECI is statically defined in the configuration.

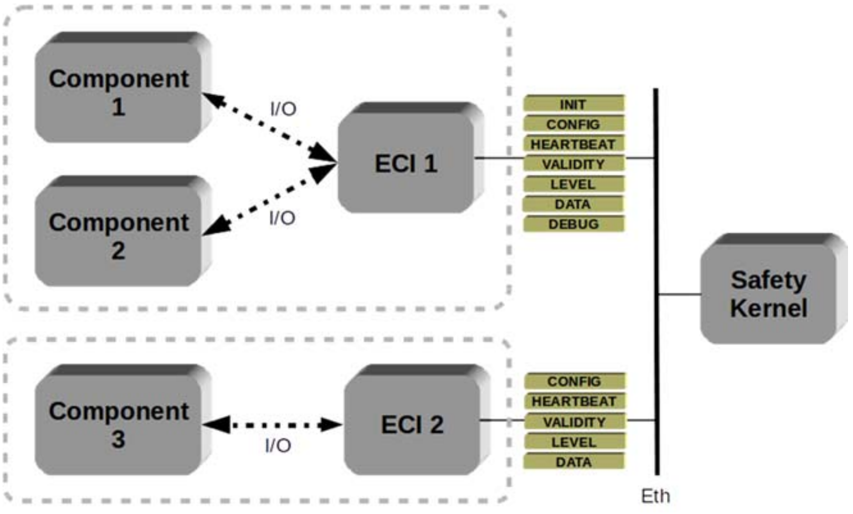


Figure 21: Multiple External Component Interfaces.

An ECI is associated with a set of components so that each LoS or *DATA* message to these components will be automatically routed to the associated ECI. Special messages as *INIT* or *DEBUG* are always routed to the default ECI.

6.4 Configuring the Safety Kernel

The Safety Kernel uses a configuration file in XML format. The file is loaded at kernel start-up. The running configuration can be updated in runtime using a special API primitive but, as explained, this is only possible for testing purposes. The XML main structure is the following:

```xml
<?xml version="1.0"?>
<config>
  <!-- System definition -->
  <system>
    ...
  </system>
  <!-- Interface definition -->
  <interface id="0">
    ...
  </interface>
  <!-- Unit definition -->
  <unit id="0">
    ...
  </unit>
  ...
</config>
```

The *system* and *interface* sections are both optional and are used respectively to define the main system attributes and configure the Safety Kernel interfaces. The unit section allows to set up the component units and the safety rules. Each unit has an ID and refers to an input, output or any locally calculated value (a local LoS for instance).

6.4.1 System definition

This section contains four optional system attributes: *FAILURE*, *SUCCESS*, *PERIOD* and *BENCHMARK*. *SUCCESS* and *FAILURE* correspond, respectively, to the minimum number of required successes and the maximum number of tolerated failures in a row, as explained in Section 6.2.5. As both attributes are declared in the *system* section, they affect all timeout-defined units.

```xml
<system>
  <failure>2</failure>
  <success>3</success>
  <period>200</period>
  <benchmark>100<benchmark>
</system>
```

The *PERIOD* attribute defines the TFD period and consequently the output sending frequency. This period must be large enough to not overload the kernel and make the ECI handle properly the kernel outputs. On the other hand, it must be small enough so that the Safety Kernel could produce a quick response to any input change (default period is 100ms).

The *BENCHMARK* attribute defines the number of benchmarking cycles to be run when a new configuration is sent to the Safety Kernel or read locally. If the attribute is 0 (default value), no benchmarking is performed. When the benchmarking cycles are achieved, the Safety Kernel runs back in a normal way.

6.4.2 Benchmarking

During a benchmarking cycle, the Safety Kernel measures the execution time of the *Time Failure Detector* (TFD), the *Safety Manager* (SM), which evaluates the rules and, finally, the *Data Component Multiplexer* (DCM), which selects the data source from the multi-components. The measures are taken regardless of the unit values (validity or LoS). All rules of all units are evaluated since the Safety Manager does not stop the evaluation when a first rule is evaluated to true. So the results are a good estimate of the worst case scenario execution.

The benchmarking returns the mean and deviation values calculated over the cycles. A cycle corresponds to the Safety Kernel period. Therefore, a benchmarking phase of 100 cycles will last about 10 seconds if the Safety Kernel period is 100ms.

The mean value is useful if a large configuration file, with a big number of rules, is used and if it is necessary to adjust the Safety Kernel period in the most accurate way. For instance, if the mean value is about 12ms and the Safety Kernel period is set to 10ms, there is a risk of overloading the Safety Kernel and loose synchronization. In this case, the period should be increased to a higher value (20ms or 30ms).

The deviation value provides an idea about how stable is the system. This value should be greater if the Safety Kernel is run on a non-real-time operating system. The smallest values should be obtained on real-time architectures (RTEMS).

6.4.3 Interface definition

Using this section of the configuration file, IP and port of the default interface (ID 0) can be changed or new interfaces (with ID > 0) can be added. These new interfaces are aimed to make communication with the Safety Kernel directly with some remote components having their own interface.

```
<!-- Default interface -->
<interface id="0">
  <port>6000</port>
</interface>
...
<!-- New interface -->
<interface id="1">
  <ip>192.168.128.200</ip>
  <port>6000</port>
</interface>
```

In the configuration above, the default interface has IP 192.168.128.100 and UDP port 6000 while a new interface is created with IP 192.168.128.200 and port 6000.

6.4.4 Unit definition

A basic unit definition can contain three different parts, each one optional: first the definition of the unit attributes, secondly the safety rules and thirdly the multi-components sources. Each unit is associated with an ID. For input and output units, this ID is written into the ID field of the packet header exchanged with the ECI.

```
<unit id="2">
  <!-- Unit attributes -->
  <mode>update</mode>
  ...
  <!-- Rule definition -->
  <rule level="4">
    ...
  </rule>
  ...
  <!-- Multi-component definition -->
  <from id="4" level="2"/>
  ...
</unit>
```

Some units correspond to components above the hybridization line. These components are not proven safe so they are required to periodically send *HEARTBEAT*, *VALIDITY*, *LEVEL* or *DATA* packets to the Safety Kernel to prove they behave properly.

HEARTBEAT or *VALIDITY* messages are used by components above the hybridization line sending a data validity value while *HEARTBEAT* or *LEVEL* are used in the same way by components sending a level of service (e.g. Cooperative LoS Evaluator). Finally, implementations of a multi- component function will use DATA packets to send back the selected output.

6.4.5 Non proven safe components

For such components, a *TIMEOUT* attribute defines the maximum period of time in milliseconds between the receptions of two consecutive packets. If not specified, the time-out is equal to 0 by default and the component is considered as proven safe.

The configured timeout must be a bit greater than the packet sending period in order to tolerate slight transmission delays. For instance, a timeout of 220ms should be set for an external component designed to send a heartbeat every 200ms.

```
<unit id="0">
  <timeout>400</timeout>
  <failure>2</failure>
  <success>3</success>
</unit>
```

The meaning of *FAILURE* and *SUCCESS* is similar to the attributes defined in the *system* section except that they have priority over the previous ones and are only valid for the associated unit.

6.4.6 Sending mode

For every unit which is a proven safe component, the Safety Kernel will update the units' level of service or performance level. By default, this level is not sent to the ECI as an output. The MODE attribute allows to force the sending. There are three available modes:

- **REGULAR**: The level is periodically sent to the ECI. The sending period is the same as the Safety Kernel one.
- **UPDATE**: The level is only sent to the ECI if it is different from the previous value.
- **SILENT**: No output (default value). The result is only computed for a local usage and remain available in another unit rule.

6.4.7 Rule definition

A rule is identified by a unit ID and a level. This level value becomes the unit level of service or performance level if the rule is evaluated to true. The rules are evaluated from the highest to the lowest LoS but it is not necessary to declare them in this order, given that rules are automatically sorted in memory.

As soon as a rule is evaluated to true, the unit is set to the corresponding level and the next rules with lower levels are not evaluated. If neither of the rules is true, the unit level is set to 0. A rule contains an evaluation tree with four different node types: *TEST*, *VALIDITY*, *LEVEL* and *VALUE*.

- **TEST node:** This node runs a Boolean test between different sub-node operands. Each operand might be a test node or a terminal node. Below the different test operations:
    - *AND*: All sub-nodes must be evaluated to true. This operator is used as default one between the first-level nodes.
    - *OR*: At least one sub-node must be evaluated to true.
    - *SUP*: True if the first operand is greater than the second one.
    - *SUPE*: True if the first operand is greater or equal to the second one.
    - *INF*: True if the first operand is lower than the second one.
    - *INFE*: True if the first operand is lower of equal to the second one.
    - *EQUAL*: True if the first operand is equal to the second one.
    - *DIFF*: True if the first operand is different from the second one.
- **VALIDITY node**: The evaluation of this node returns the latest data validity timely received by this unit through the *VALIDITY* packet.
- **LEVEL node**: The evaluation of this node returns the current level of the unit. This value comes from three different sources:
    - The latest level of service timely received by this unit through the *LEVEL* packet.
    - The latest rule evaluated to true for this unit. In this case, the level can be either a level of service or a performance level.
    - The level of service of the latest selected output from a multi-component function.
- **VALUE node**: Evaluation returns the static value of the node.

To illustrate the rule creation, we show below a basic LoS rule definition for a functionality. The level of service of the functionality is 1 if the validity of the component 0 is greater than 50. Otherwise the level of service is 0 (default value).

```
<unit id="1">
  <rule level="1">
    <test type="sup">
      <validity id="0"\>
      <value>50</value>
    </test>
  </rule>
</unit>
```

6.4.8 Multi-component definition

A multi-component definition is a list of component sources introduced by the *FROM* attribute. Each source has an associated performance level and is supposed to periodically send to the Safety

Kernel a data input. Among all sources which sent timely these data, the Safety Kernel will select the one with the highest PL and will send it back to the *ECI*.

At least one source must be timely received by the Safety Kernel otherwise a warning message will be sent to the *ECI* saying that none of the sources is timely.

```
<system>
  <period>200</period>
</system>
<unit id="2">
  <from id="3" level="0"/>
  <from id="4" level="1"/>
  <mode>regular</mode>
</unit>
<unit id="4">
  <timeout>200</timeout>
</unit>
```

In the above example, the multi-component function is composed by two sources: component 3 with performance level 0 and component 4 with performance level 1. Component 4 is not proven safe and must send its result with a maximum period of 200ms. The Safety Kernel will forward the selected data to the ECI (ID 2) every 200ms (Safety Kernel period).

Every 200ms, the *PL* of component 2 will be updated with the *PL* of the selected source (0 or 1). As the *REGULAR* mode is enabled for component 2, this *PL* will be periodically sent to the *ECI* (*ID* 2).

There is a special case in which both safety rules and multi-component sources are defined in a unit. The rules have always priority to update the unit level, which means the selected data will be periodically sent to the *ECI* but the component *PL* will not be updated with the source *PL*.

6.4.9 Interface attribute

In the unit section, the interface attribute defines the remote interface to be used by some units to send their LoS updates or selected data.

```
<unit id="3">
  <interface>2</interface>
  ...
</unit>
```

Even if all units are configured to communicate with a new interface with *ID* > 0, *INIT* and *DEBUG* messages will be still sent to the default interface (*ID* 0). So creating the default interface within the *ECI* code is not mandatory but in this case *INIT* and *DEBUG* messages might not be handled properly.

Moreover the interface attribute is only used to define the output interface for an unit but this latter unit can receive any message (*CONFIG*, *HEARTBEAT*, *VALIDITY*, *LEVEL* or *DATA*) from any interface (default or not).

6.5 Use case example

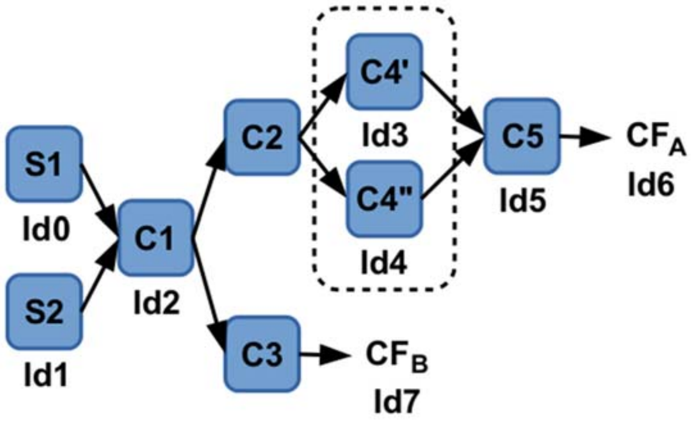


Figure 22: Example functions.

We consider an application in which two cooperative functions are implemented. These functions use two sensors, *S1* and *S2*, and five functional components, from *C1* to *C5*. Both sensors provide a data validity value associated to the sensor data they produce, which is sent to the Safety Kernel (*V1* and *V2*). *C4* is a multi-component with two implementations, *C4'* and *C4"*, corresponding, respectively, to performance levels *PL1* and *PL0*.

According to the execution timeliness of *C4'*, called ET_{C4-PL1}, the *Data Component Multiplexer* will forward the selected value from *C4* to *C5*. Finally, *C1* is a component below the hybridization line able to execute with three different performance levels (from PL2 to PL0). We also consider that the safety rules for both functions are the following (the bounds have to be defined at design-time, and it must be proven that the functions will be safely performed in each LoS when the safety rules are met):

$$CF_A(LoS3) \rightarrow V1 > 80 \wedge ET_{C4\_PL1} < D_{C4} \qquad CF_B(LoS3) \rightarrow V1 > 80 \wedge V2 > 70$$
$$CF_A(LoS2) \rightarrow V1 > 60 \wedge ET_{C4\_PL1} < D_{C4} \qquad CF_B(LoS2) \rightarrow V1 > 80$$
$$CF_A(LoS1) \rightarrow V1 > 60 \qquad CF_B(LoS1) \rightarrow V1 > 60$$
$$CF_A(LoS0)\ otherwise \qquad CF_B(LoS0)\ otherwise$$

Table 7 shows the performance levels of *C1* and *C4* in dependence of the *LoS* of both functions. All invalid combinations have been removed.

Table 7: Performance levels for each combination of LoS.

CF$_A$	CF$_B$	*C1*	*C4*
LoS3	LoS3	PL2	PL1
LoS1	LoS3	PL2	PL0
LoS3	LoS2	PL1	PL1
LoS2	LoS1	PL1	PL1
LoS1	LoS1	PL1	PL0
LoS0	LoS0	PL0	PL0

Given the information provided in Table 7, it is possible to define a set of expressions that can be used to calculate these performance levels. The expressions are the following:

$$C1(PL2) \rightarrow CF_B\_LoS = 3$$
$$C1(PL1) \rightarrow CF_B\_LoS > 0$$
$$C1(PL0)\ otherwise$$

Given all the above expressions, required to determine the feasible LoS for each function and the corresponding component performance levels, it is possible to create an XML configuration file. An identifier must be first assigned to each unit (e.g., ID0 for *S1*, ID1 for *S2*, ...), and this allows the proper references to be made in the configuration file. Note that there are no IDs for *C2* ad *C3* as they are not involved in any expressions. A subset of the resulting configuration file is presented below.

```
<?xml version="1.0"?>
<config>
  <!-- C1 component -->
  <unit id="2">
    <mode>update</mode>
    <rule level="2">
      <test type="equal">
        <level id="7"/>
        <value>3</value>
      </test>
    </rule>
    <rule level="1">
      <test type="sup">
        <level id="7"/>
        <value>0</value>
      </test>
    </rule>
  </unit>
  ...
  <!-- Function A -->
  <unit id="6">
    <rule level="3">
      <test type="sup">
        <validity id="0"/>
        <value>80</value>
      </test>
      <test type="equal">
        <level id="5"/>
        <value>1</value>
      </test>
    </rule>
    <rule level="2">
      <test type="sup">
        <validity id="0"/>
        <value>60</value>
      </test>
      <test type="equal">
        <level id="5"/>
        <value>1</value>
      </test>
    </rule>
    <rule level="1">
      <test type="sup">
        <validity id="0"/>
        <value>60</value>
      </test>
    </rule>
  </unit>
  <!-- Function B -->
  <unit id="7">
    <rule level="3">
      <test type="sup">
        <validity id="0"/>
        <value>80</value>
      </test>
      <test type="sup">
        <validity id="1"/>
        <value>70</value>
      </test>
    </rule>
    <rule level="2">
      <test type="sup">
        <validity id="0"/>
        <value>80</value>
      </test>
    </rule>
    <rule level="1">
      <test type="sup">
        <validity id="0"/>
        <value>60</value>
      </test>
    </rule>
  </unit>
</config>
```

7. Performance analysis and evaluation

The objective of the performance analysis provided in this section is to determine the ability of the Safety Kernel to scale with the system dimension. In particular, we will study how the number of inputs/outputs and the complexity of the safety rules could affect the execution time of the different modules of the Safety Kernel. To achieve this goal, we follow two approaches: a formal analysis and a benchmarking test. The former allows deriving a complexity measure for the Safety Kernel execution time in the worst case scenario where all rules are evaluated. The latter is aimed at obtaining execution time measures from a running Safety Kernel.

7.1 Formal analysis

We define T_{SK} as the maximum execution time of the Safety Kernel process. This process is powered by two threads: a *Listener Thread* is activated for every incoming packet and a *Periodic Thread* runs once at every kernel period. Given that $T_{listener\ thread}$ and $T_{periodic\ thread}$ represent, respectively, the maximum execution time assumed for the listener thread and the periodic thread, and that $N_{packets}$ represents the total number of input messages received during one execution period, T_{SK} will be given by:

$$T_{SK} = N_{packets} \times T_{listener\ thread} + T_{periodic\ thread}$$

There are different types of incoming packets: heartbeat, data validity, multicomponent data or cooperative level of service. Therefore, the worst case execution time for the listener thread corresponds to the longest processing time, out of the processing times for each message type. We also take into consideration the time necessary to read a packet from the network, represented by $T_{network\ reading}$. $T_{listener\ thread}$ can thus be expressed as:

$$\begin{aligned} T_{listener\ thread} = {} & T_{network\ reading} \\ & + max(T_{heartbeat\ handling}, T_{validity\ handling}, T_{data\ handling}, T_{LoS\ handling}) \end{aligned}$$

All handling times are fixed so the complexity of $T_{listener\ thread}$ is $O(1)$.

The periodic thread runs 3 functions in sequence: the *Timing Failure Detector*, the *Safety Manager* and the *Data Component Multiplexer*. Therefore, we have:

$$T_{periodic\ thread} = T_{TFD} + T_{SM} + T_{DCM}$$

TFD and *DCM* functions scan the unit array to respectively find out the update values untimely received and the component data value to be forwarded. Complexities of T_{TFD} and T_{CM} are both $O(N_{units})$.

The *Safety Manager* is the more complex process as it evaluates for each unit the safety rules and determines the new level of service or performance level.

$$T_{SM} = N_{units} \times N_{rules\ per\ unit} \times N_{nodes\ per\ rule} \times T_{node\ eval}$$

The time to evaluate one rule node ($T_{node\ eval}$) is steady whatever the node type. So the complexity of T_{SM} is $O(N_{nodes})$.

Let's come back to the T_{SK} formula. In normal situations, $N_{packets}$ corresponds to the number of input units, since each unit is supposed to send one update value during the Safety Kernel period.

$$T_{SK} = N_{packets} \times T_{listener\ thread} + T_{periodic\ thread}$$

Given the previous results, the complexity of T_{SK} is $O(N_{units}, N_{nodes})$.

7.2 Benchmarking test

One of the objectives of performing a real benchmarking test is to get some insight on which of number of units or number of nodes has more weight on the Safety Kernel execution time.

We choose to measure two execution times: first T_{config}, which is to the necessary time to parse a XML configuration and create the safety rules, and secondly $T_{periodic\ thread}$, which corresponds to the execution of the three main Safety Kernel modules: the *Timing Failure Detector*, the *Safety Manager* and the *Data Component Multiplexer*.

The measurement system is intrusive as we use timer start/stop functions embedded in the kernel code. However, this intrusiveness is deemed as insignificant. All measures are taken while running the Safety Kernel on a FPGA board, on top of RTEMS. The board is the same that is used in the KARYON demonstrators, implemented in the scope of WP5.

We built three types of configurations as depicted in Figure 23.

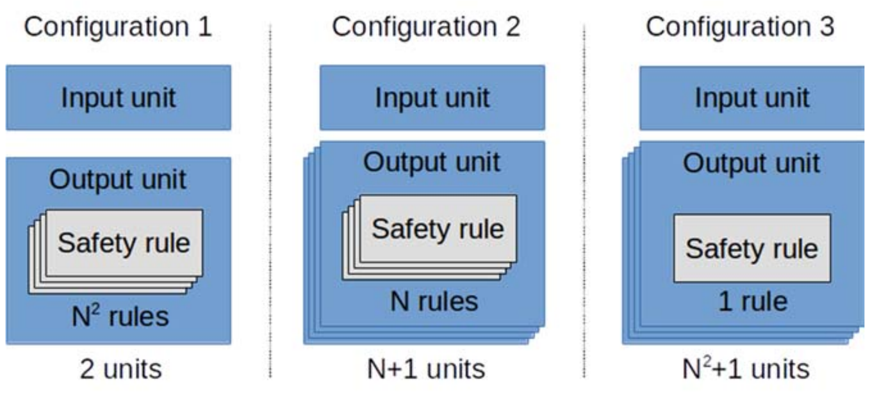
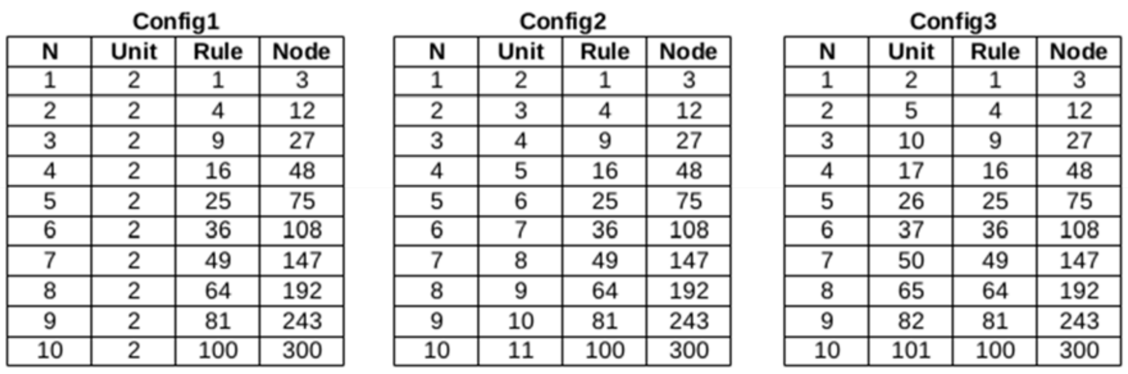


Figure 23: Benchmarking configurations.

Each configuration involves a single input unit and one or more output units. The number of units, rules and nodes is a function of *N*, which goes from 1 to 10. Configuration 1 implements one output unit with N^2 rules, configuration 2 implements *N* output units with *N* rules in each unit and configuration 3 implements N^2 output units with only one rule in each unit. Whatever the configuration, each rule is always built with three nodes. As a result, the number of rules and nodes is the same in each configuration for a given value of N. The tables presented in Figure 24 provide all the values.



Config1

N	Unit	Rule	Node
1	2	1	3
2	2	4	12
3	2	9	27
4	2	16	48
5	2	25	75
6	2	36	108
7	2	49	147
8	2	64	192
9	2	81	243
10	2	100	300

Config2

N	Unit	Rule	Node
1	2	1	3
2	3	4	12
3	4	9	27
4	5	16	48
5	6	25	75
6	7	36	108
7	8	49	147
8	9	64	192
9	10	81	243
10	11	100	300

Config3

N	Unit	Rule	Node
1	2	1	3
2	5	4	12
3	10	9	27
4	17	16	48
5	26	25	75
6	37	36	108
7	50	49	147
8	65	64	192
9	82	81	243
10	101	100	300

Figure 24: Number of units, rules and nodes in each configuration.

7.2.1 Configuration parsing time

The results presented in Figure 25 illustrate the time to parse the XML file for each of the three considered cases. The presented time values correspond to the time taken to build all the safety rules in the safety rules repository. The configuration loading time (file access time or network transmission time) is not considered.

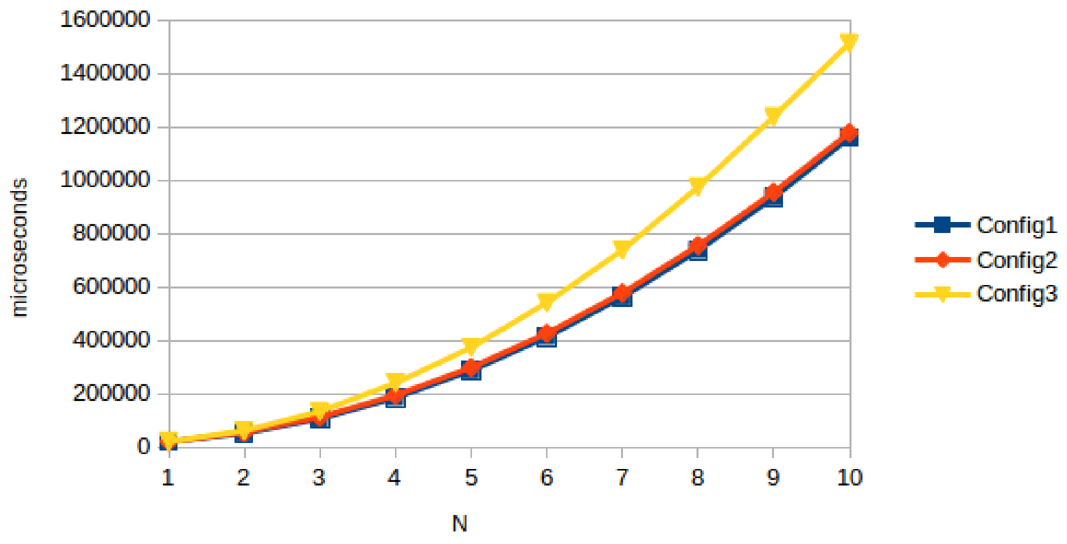


Figure 25: Configuration parsing time.

The configuration parsing is time consuming. In configuration 3, the parsing time reaches 1.5s for 101 units and 300 nodes. The difference between configuration 3 and the two other configurations can be explained by the greater number of units. Despite being a time consuming process, it is executed only once, at bootstrap, and is thus not significant from the perspective of concluding about scalability or ability of the Safety Kernel to perform timely and efficiently.

7.2.2 Periodic execution time

Execution times are mean values calculated from 100 experiments. Each time, we consider the worst case scenario where all rules of all units are evaluated. The time values for TFD, SM and DCM are separately measured. In Figure 26, Figure 27 and Figure 28 we show the obtained results. Each figure corresponds to one of the configurations.

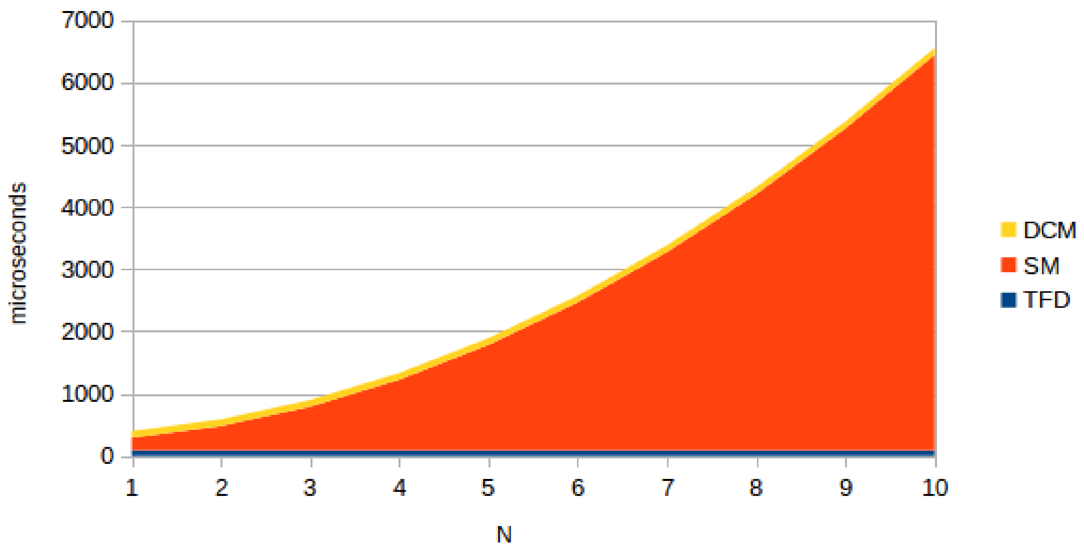


Figure 26: Execution time (configuration 1).

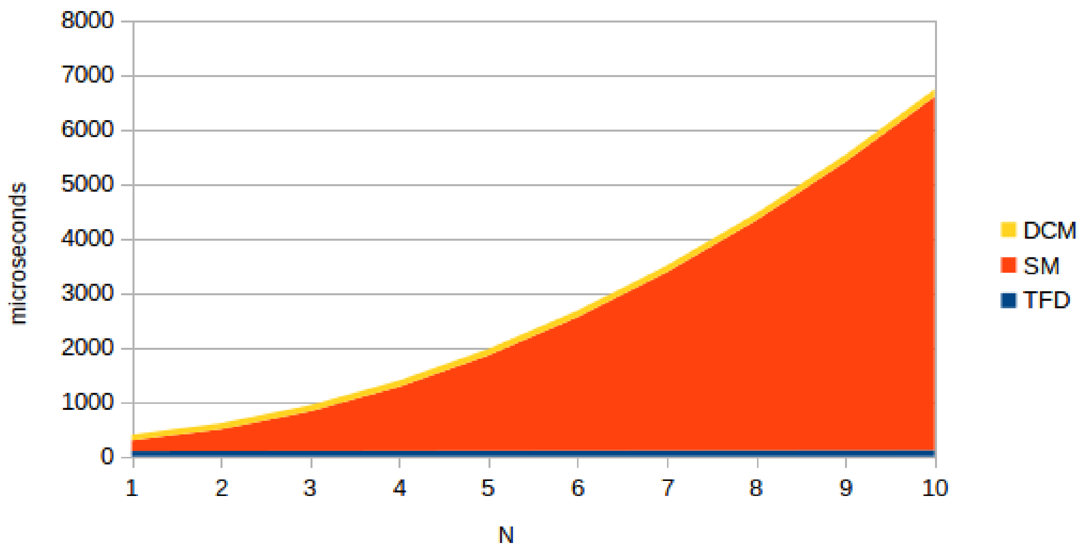


Figure 27: Execution time (configuration 2).

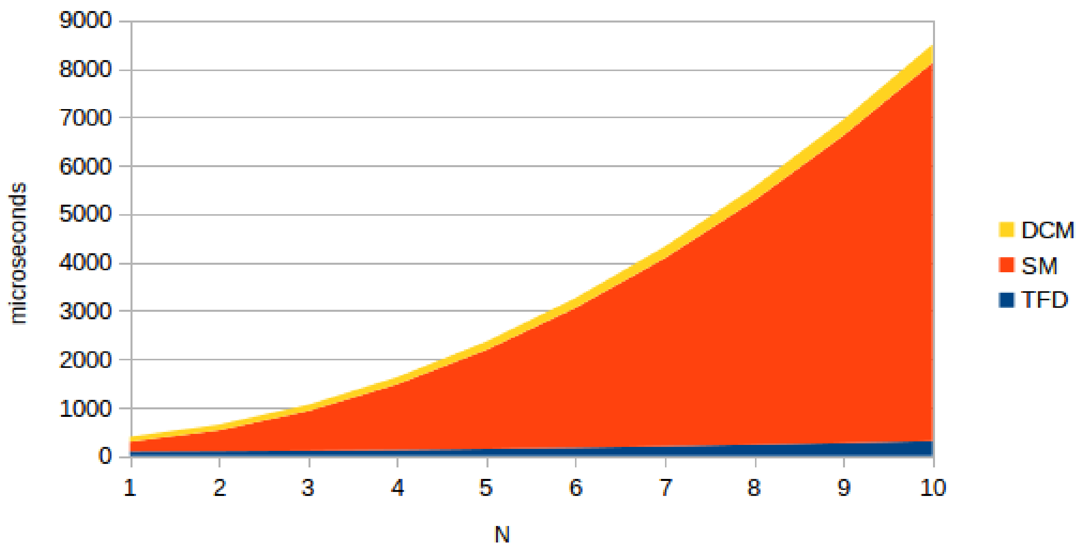


Figure 28: Execution time (configuration 3).

TFD and DCM execution times only depend on the number of units and are marginal compared to the SM execution time. This is quite normal because the SM handles the evaluation of the safety rules, which is the most complex task in the Safety Kernel.

The number of units has a weak influence on the complexity of T_{SK}, which can be approached by $\boldsymbol{O(N_{nodes})}$**.** For instance, the evaluation time for 100 rules and 300 nodes is lower than 10ms. Given that this time corresponds to the worst case scenario, it remains below the default Safety Kernel period (100ms or 200ms), as we are considering in the KARYON demonstrations.

To conclude, this analysis showed the strong correlation between the Safety Kernel execution time and the total number of nodes created in the safety rules. It also showed that the overall execution time is quite small, and sufficiently small to allow this particular instantiation of the Safety Kernel to be integrated and used in demonstrations.

It is important to note that the implementation makes use of an FPGA board, in which a LEON3 soft-processor (SPARC v8 arch) is programmed and is running the RTEMS as the operating system and the Safety Kernel as the application. Therefore, the obtained results could be very significantly improved (possible 2 or 3 orders of magnitude) if using a physical (not soft) processor. Nevertheless, the intended objectives were achieved, as it is possible to use the Safety Kernel with this implementation, with the advantage that a development board was used and could have served to test different solutions (e.g., using different soft-processors).

8. Conclusions

In this deliverable we provided a comprehensive collection of information that is relevant to understand the need for a Safety Kernel, to understand its role and the proposed architecture. The deliverable goes into details about the definition of this Safety Kernel, in terms that should allow others to continue from here, implementing and improve the ideas, possibly, and hopefully, to realise new cooperative systems. It also provides details on a concrete implementation of the ideas, explaining how the implementation can be used and, in this way, illustrating the feasibility of the concept.

Despite being, in its essence, a simple component, the Safety Kernel plays a fundamental role in the KARYON architectural pattern. From the initial rough idea expressed in the DoW of what a Safety Kernel could be, a simple component responsible for managing the system safety, to what we have today, a well-defined part of the system whose role and interactions with the rest of the system are well-understood, many concepts had to be clarified and many issues had to be addressed. The work on the definition of the Safety Kernel was not isolated. On the contrary, it was done in a strictly closed connection with the work on the definition of the generic architectural pattern, with the work on the definition of an abstract sensor model, with the work on the definition of what a safety analysis means for cooperative systems with multiple levels of service, with the work on the definition of requirements for use cases in the automotive and avionic domains and, finally, with the work on the development of the use case demonstrators.

We believe that the results we achieved, not only the Safety Kernel definition, but the whole KARYON approach, are valuable as enablers for future research and development initiatives focused on autonomous and cooperative applications. Our focus was on the automotive and avionic domains, but the approach as a much larger reach. We believe that it may be applied for the development of any control system in which it is necessary to balance the achievable performance with the required safety, while using cost-effective components and dependability solutions.

References

[1] J. Rufino, J. Craveiro, and P. Veríssimo, “Architecting Robustness and Timeliness in a New Generation of Aerospace Systems,” in *Architecting Dependable Systems VII*, LNCS 6420, A. Casimiro, R. de Lemos, and C. Gacek (Eds.), Berlin Heidelberg: Springer-Verlag, 2010, pp. 146-170.

[2] J. Rushby, “Partitioning in avionics architectures requirements mechanisms and assurance” SRI International, California, USA, Tech Rep. NASA CR-1999-209347 Jun. 1999.

[3] GLib Project: Rsimple xml subset parser, version 2.37 (2014).

[4] AEEC. Avionics application software standard interface. ARINC Specification 653 Supplement 1, Jan. 1997.

[5] AUTOSAR. Requirements on operating system, V3.1.0, R4.1 Rev 1, Jan. 2013.

[6] J. Barhorst, T. Belote, P. Binns, P. Hoffman, J. Paunicka, P. Sarathy, J.S.P. Stanfill, J.S.P. Stuart, and R. Urzi, “A research agenda for mixed-criticality systems (2009)”, http://www.cse.wustl.edu/˜cdgill/CPSWEEK09_MCAR/RBO-09-130%20Joint%20MCAR%20White%20Paper%20PA%20approved.pdf, white paper

[7] P. Veríssimo, “Uncertainty and predictability: Can they be reconciled?”, In *Future Directions in Distributed Computing*, Schiper, A., Shvartsman, A.,Weatherspoon, H., Zhao, B. (eds.), Lecture Notes in Computer Science, vol. 2584, pp. 108–113. Springer Berlin Heidelberg, 2003.

[8] P. Veríssimo, “Travelling through wormholes: a new look at distributed systems models”. SIGACT News 37(1), 66–81, Mar 2006.

[9] R. Obermaisser, H. Kopetz, (eds.): “GENESYS: A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems”, Sep 2009.

[10] K.H. Kim, “The distributed recovery block scheme,” in *Software Fault Tolerance*, Lyu, M.R. (ed.), chap. 8, pp. 189–209. John Wiley Sons, 1995.

[11] L. Sha, “Using simplicity to control complexity,” *IEEE Software,* 18(4), 20–28, Jul/Aug 2001.

[12] G. Heiser, “The role of virtualization in embedded systems,” in *First workshop on Isolation and integration in embedded systems (IIES’08).* Glasgow, Scotland, Apr 2008.

[13] AUTOSAR: Specification of operating system, v5.1.0, release 4.1, revision 1, Feb 2013.

[14] ISO 26262: Road Vehicles – Functional Safety – Part 10: Guideline on ISO 26262.

Annex A Functional Safety for Cooperative Systems

Josef Nilsson, Carl Bergenhem, Jan Jacobson, Rolf Johansson and Jonny Vinter, “Functional Safety for Cooperative Systems”. In proceedings of SAE International Congress 2013.

This page is intentionally left blank.

SAE International

Functional Safety for Cooperative Systems

2013-01-0197
Published
04/08/2013

Josef Nilsson
SP Technical Research Inst of Sweden

Carl Bergenhem
Qamcom

Jan Jacobson, Rolf Johansson and Jonny Vinter
SP Technical Research Inst of Sweden

Copyright © 2013 SAE International
doi:10.4271/2013-01-0197

ABSTRACT

This paper investigates what challenges arise when extending the scope of functional safety for road vehicles to also include cooperative systems. Two generic alternatives are presented and compared with one another. The first alternative is to use a vehicle centric perspective as is the case in the traditional interpretation of ISO 26262 today. Here, an “item” (the top level system or systems for which functional safety is to be assured) is assumed to be confined to one vehicle. In the vehicle centric perspective inter-vehicle communication is not an architectural element and is therefore not a candidate for redundancy as part of the functional safety concept. The second alternative is to regard a cooperative system from a cooperative perspective. This implies that one item may span over several vehicles. The choice of perspective has implications in several ways. We investigate the implications for the cooperative item and in what ways the results may differ when going through the reference life cycle of ISO 26262. In particular we look at classification of hazardous events where severity is significantly higher since the cooperative system involves multiple rather than one single vehicle. We therefore suggest an additional severity class and as a consequence introduce a new automotive safety integrity level, ASIL E. The cooperative perspective includes the inter-vehicle communication as a candidate for redundancy. ASIL E can therefore be achieved using ASIL decomposition and the currently recommended product development phases for ASIL A to ASIL D. As an example for illustrating we use platooning.

INTRODUCTION

The current standard for functional safety for road vehicles, ISO 26262 [1], was released late 2011. It is currently being introduced and applied at a number of companies including car manufacturers and suppliers of different tiers. Meanwhile there is a growing interest in cooperative functions that rely on vehicle-to-vehicle and vehicle-to-infrastructure communication. Safety-related cooperative functionalities are still not commercially available. There are however research projects already showing working demonstrators. One such example is the SARTRE project [2] where platooning has been successfully shown to operate with a combination of trucks and passenger cars on public roads. A relevant question is therefore: Can ISO 26262 be efficiently applied to cooperative functions?

In ISO 26262, the procedure is to look at one “item” at a time and assess that this item will be safe. The user of ISO 26262 confines what is to be one item, but it cannot be narrower than a function including some sensing and some actuating. Even if it is not explicitly mentioned, it is obvious that what is to be regarded as one item is confined to a functionality performed by one vehicle. Furthermore the standard makes the assumption that all items can be analyzed independently of each other. For cooperative functions this can be an issue as functionalities extends beyond the boundaries of individual vehicles. As an example the function of platooning involves all vehicles that take part in the road train.

An explicit issue with the single vehicle perspective of ISO 26262 regards hazard analysis and risk assessment (HA&RA). With the item confined within the boundaries of one vehicle, a hazard is caused by malfunctioning behavior of a single vehicle. However, for cooperative functions a single failure may affect multiple vehicles and potentially cause hazardous situations that are significantly more severe than the consequences of a single vehicle malfunctioning. An example is a brake commission failure (unintended full brake application) of a vehicle driving at high velocity in a platoon with short inter-vehicle distances. Because platooning involves several vehicles driving at short distances from each other, hazardous situations are potentially more severe. As a result, risks identified during HA&RA may be misjudged and underestimated in the single vehicle perspective. Furthermore, the single vehicle perspective limits the design options for dealing with hazards, as each vehicle is addressed individually. A solution where cooperation between vehicles is used as a means to deal with hazards is not supported.

To address these issues we propose a *cooperative perspective* where the item may encompass multiple vehicles and include communication links that connect vehicles with each other or with infrastructure. This perspective broadens the scope of what can be included in the definition of an item. When applied to a HA&RA it requires the analysis of hazards that simultaneously affect several vehicles. The cooperative perspective also supports the use of inter-vehicle communication to deal with hazards as suggested in e.g., [3].

The remainder of the paper is structured as follows. The next section contains a definition and description of cooperative systems. This is followed by an overview of ISO 26262, describing best-practice in functional safety for road vehicles. In the next section we present platooning as an example of a cooperative functionality. This example is used in the subsequent section to illustrate and motivate the use of a cooperative perspective in the development stages outlined by ISO 26262. The paper ends with a summary and conclusions.

COOPERATIVE SYSTEMS

Communication between vehicles and between vehicles and infrastructure adds new capabilities to the traffic system [4]. It provides a means for distributing information (e.g., sensor readings, intentions, and properties of vehicles) and allows negotiations among road users to reach a common goal (e.g., optimize flow in intersections, join a vehicle platoon, or control hazardous situations). A taxonomy of the types of applications that can be realized using wireless inter-vehicle communication is presented in [5]. The taxonomy divides applications into four types. Type 1 applications use communicated information that may be delayed or lost without compromising safety. Examples of such information are weather reports or entertainment feeds. Type 2 applications provide services that may compromise safety if communication fails. This type includes but is not limited to applications that use information about hazardous road conditions or abnormal vehicle behavior to issue safety alerts. Applications of Type 3 rely on communicated information from other vehicles about motion and actuator states in order to ensure safe and/or efficient operation. Collision avoidance based on communicated information is an example of an application of Type 3. Type 4 applications use inter-vehicle communication to reach a common goal. The previously mentioned platooning functionality falls into this type of application.

The taxonomy shows that inter-vehicle communication can be used to broadcast information between neighboring vehicles. This provides in-vehicle systems with an improved view of the traffic environment and has enabled the development of cooperative adaptive cruise control, emergency vehicle warning, intersection management and other applications [6, 7]. A more complex use of inter-vehicle cooperation is to use it for negotiations and to perform coordinated actions. This type of cooperation extends the capabilities further and allows for applications such as optimized path planning and platooning [5]. As a consequence of negotiation and coordination, inter-vehicle dependencies are strengthened and boundaries erased. This has implications for functional safety. The significance of vehicle sharing a consistent view of the traffic environment and reaching consensus before taking action become increasingly important for safety. As previously mentioned, hazardous events can become more severe when more vehicles are involved. On the other hand, cooperation enables new schemas for controlling hazards. Neighboring road users may assist a malfunctioning vehicle as suggested in [3] or cooperating vehicles may adapt their behavior based on information about the integrity of system components and confidence in perception as outlined in [8].

This paper targets Type 2, 3, and 4 applications with emphasize on Type 4 where inter-vehicle communication is used for complex cooperation that affects functional safety. In addition to the previously mentioned challenges concerning consistency and consensus, cooperative applications often introduce high levels of automation and reduce safety margins, changes that also impact functional safety. For many Type 3 and 4 applications, automation replaces the human driver who then cannot be relied upon to control hazards. Furthermore, inter-vehicle distance is in many cases reduced to improve efficiency. The required level of safety integrity of cooperative applications is therefore high and requires attention.

FUNCTIONAL SAFETY FOR ROAD VEHICLES

The international standard ISO 26262 [1] addresses functional safety in road vehicles. ISO 26262 is intended to be applied to safety-related systems that include one or more electrical/electronic (E/E) systems and that are installed in series production passenger cars with a maximum gross weight up to 3 500 kg. The standard ISO 26262 addresses possible hazards caused by malfunctioning behavior of E/E safety-related systems including interaction of these systems. An example of malfunction is if an obstacle detection system gives an alarm even if an obstacle is not present. The standard does not address hazards as electric shock, fire, smoke, heat, radiation, toxicity, flammability, reactivity, corrosion, release of energy, and similar hazards.

Functional safety concerns are not new in the automotive industry. A low complexity hydraulic braking system has requirements for functional safety since a brake response of the vehicle is expected every time the brake pedal is pushed. The introduction of high complexity E/E systems has called for a standard to handle functional safety. The ISO 26262 provides an automotive safety lifecycle, an automotive specific risk-based approach for determining risk classes (Automotive safety Integrity levels, ASILs) and requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety is achieved.

The safety lifecycle as depicted in Figure 1 starts with the definition of the item and ends with decommissioning. The basic idea of the safety lifecycle is to consider functional safety aspects all through the lifetime of the vehicle. The ISO 26262 describes all phases of the safety lifecycle, the activities for each cycle and the necessary work products.

During the concept phase; the item shall be defined, the safety lifecycle shall be initiated, a HA&RA shall be performed and the functional safety concept shall be defined. The item definition has the objective to define and describe the item, its dependencies on, and interaction with, the environment and other items. It is also intended to support an adequate understanding of the item so that the activities in subsequent phases can be performed. An item is understood as a system or array of systems to implement a function at the vehicle level. The HA&RA aims to identify and to categorize the hazards that malfunctions in the item can trigger. It shall also formulate the safety goals related to the prevention or mitigation of the hazardous events, in order to avoid unreasonable risk. The functional safety concept shall derive the functional safety requirements, from the safety goals, and allocate them to the preliminary architectural elements of the item, or to external measures. Fault detection, failure mitigation, transition to a safe state, fault tolerance mechanisms and fault detection are examples of what the functional safety concept should contain. A well performed concept phase of the overall safety lifecycle sets the frame for the detailed development of the system, the hardware and the software.

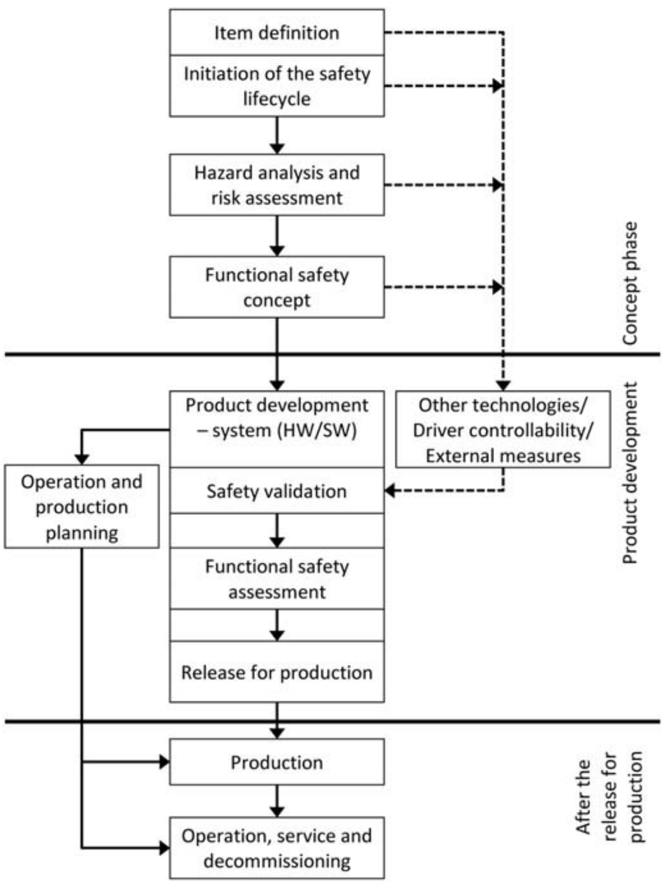

Item definition
Initiation of the safety lifecycle
Hazard analysis and risk assessment
Functional safety concept
Concept phase
Product development – system (HW/SW)
Other technologies/ Driver controllability/ External measures
Operation and production planning
Safety validation
Functional safety assessment
Release for production
Product development
Production
Operation, service and decommissioning
After the release for production

Figure 1. Simplified model of the overall safety lifecycle adopted from ISO 26262 [1].

An ASIL shall be determined for each hazardous event using the estimation parameters severity (S), probability of exposure (E) and controllability (C). The severity parameter S is a measure of the extent of harm to the persons involved in a specific situation. It can be classified as no injuries, light and moderate injuries, severe and life-threatening injuries (survival probable), life-threatening injuries (survival uncertain) or fatal injuries. The E parameter of each operational situation shall be estimated. The probability of E shall be described as incredible, very low probability, low probability, medium probability or high probability. The parameter C describes avoidance of the specified harm or damage through the timely reactions of the persons involved. The potential for C shall be assigned to controllable in general, simply controllable, normally controllable or difficult to control (uncontrollable). The ASIL is stated as one of four levels to specify the necessary requirements of the item or the element and safety measures for avoiding an unreasonable residual risk. ASIL D represents the most stringent and ASIL A the least stringent level. Quality management (QM) may be applied when the risk is so low that no requirement to comply with ISO 26262 is given.

It is sometimes regarded feasible to divide a system into redundant elements. The safety requirement for the function may then be distributed on the redundant elements. The requirements for techniques and measures will then be less demanding than they would have been on the single element. The decomposition of safety requirements can be made with respect to ASIL tailoring. This procedure is called "ASIL decomposition" in ISO 26262 (1-1.7). The intention is to appoint safety requirements redundantly to sufficiently independent elements with the objective of reducing the ASIL of the redundant safety requirements, that are allocated to the corresponding elements. During the allocation process, benefits can be obtained from architectural decisions and the existence of independent architectural elements. ASIL decomposition offers the opportunity to implement safety requirements redundantly by these independent architectural elements and potentially assign a lower ASIL to those redundant requirements.

The standard is applied by several car manufacturers and by their suppliers. It is evident that different phases will be addressed depending on the perspective of the user. A software supplier will apply other techniques and measures than an original equipment manufacturer. Experience in the application of the standard creates a common understanding of how functional safety should be achieved. It can also be expected that the principles will be applied also for trucks and buses, and for cooperative functionality where several vehicles interact.

EXAMPLE - PLATOONING

The basic definition of vehicle platooning is a number of vehicles that are intentionally grouped together in a tight formation and travel in a coordinated manner. There can be different variations of the concept of platooning such as the degree of automation, the type(s) of vehicles that are included, the motivation for doing it, types of road, requirements on infrastructure etc. Two different concepts of platooning are researched in the PATH [11] and GCDC [12] projects. PATH automates both lateral and longitudinal control in platoons with one vehicle type while GCDC automates only longitudinal control in platoons with mixed vehicle types. Both have requirements on the infrastructure. PATH requires dedicated lanes and reference markers in the road surface and GCDC requires augmented GPS positioning. An overview of platooning systems is given in [9].

Another variant of platooning is SARTRE [2] - a European Commission funded FP7 project. It seeks to support a step change in transport utilization by developing and integrating solutions that allow vehicles to drive in platoons. SARTRE defines a platoon (or road train) as a collection of vehicles led by a manually driven designated heavy lead vehicle (e.g. truck or bus). ADAS (Advanced driver Assistance Systems) are assumed to be used in the lead vehicle. Examples are driver monitoring to detect drowsiness and inattentiveness of the lead vehicle driver. Also lane departure warning and lane keeping systems are assumed in the lead vehicle. An illustration of a SARTRE platoon is presented in Figure 2. The following vehicles (trucks and passenger cars) follow the lead vehicle automatically; both laterally and longitudinally. Vehicles may join or leave the platoon dynamically, e.g., join a desired platoon at a known rendezvous-point or leave the platoon on arrival at the desired destination. SARTRE aims to explore technology for platooning on motorways, without changes to the infrastructure, that it is safe enough to allow mixing with other users on public roads. Expected advantages of platooning include a reduction in fuel consumption, increased safety and increased driver convenience and comfort. The concept has been demonstrated on public roads with a truck as lead vehicle. There was another truck and three cars as following vehicles; hence a total of five vehicles.

The SARTRE platooning application uses V2V (vehicle-to-vehicle) communication in addition to local sensors in each vehicle. Using V2V communication implies that data from local sensors, such as its speed and direction of the own vehicle, can be broadcast directly from the source vehicle to other vehicles in the platoon. This implies that all vehicles should have the same data available via V2V communication. Lead vehicle commands, such as the trajectory to follow, is also broadcasted to following vehicles. The local sensors in a vehicle can also be used to indirectly measure the intentions of the platoon, e.g., lead vehicle command or emergency braking by a following vehicle, based on the movements of the surrounding vehicles. However, used in this indirect way, the measurement, e.g. of lead vehicle acceleration, is prone to delay and to accumulated error. This is because local sensor measurements are only based on the adjacent vehicle and cannot detect non-adjacent vehicles. Put another way, there is no "look ahead" of movements in vehicles that are beyond adjacent vehicles. For example, the lead vehicle can directly send requested acceleration data with V2V communication rather than having a following vehicle, that is several vehicles behind the lead vehicle, measure the change in velocity of the preceding vehicle with its local sensors. With local vehicle sensors a change in acceleration has to "propagate" through the platoon from the lead vehicle to each of the following vehicles and be detected. This affects, for example, the minimum gap size that can be safely achieved. Without V2V data the gap has to be larger to allow for the slower response. Using only local sensors can also lead to lateral and longitudinal instability, increasing oscillations, and unsafe behavior of the platoon. The two sources for sensor data are combined using sensor fusion where V2V data is preferred.

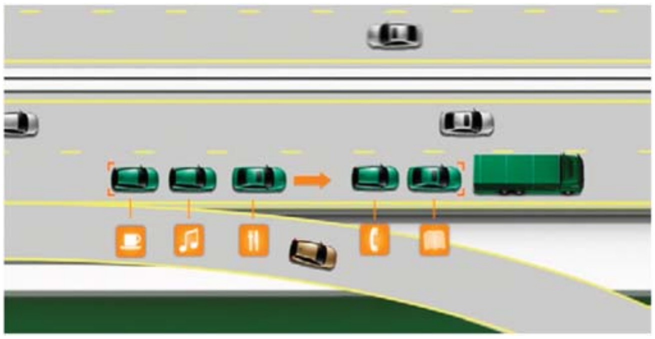


Figure 2. A SARTRE platoon. One heavy lead vehicle and five following cars.

The goal of longitudinal control, e.g., the speed of the following vehicles, is to make coordinated movements that are accurate and adequately safe. Two examples are, keeping a set gap between vehicles and being able to perform an evasive maneuver such as emergency brake. Lateral control, such as steering and lane change, has a similar goal, solution, and also faces similar challenges.

According to [5], SARTRE platooning is a Type 4 application (see section Cooperative Systems). There are several motivations for this: The platooning function is highly automated - relieving the driver of control for an extended period of time. In SARTRE, the target inter-vehicle gaps are small (approx. 5m). This gap is unsafe for manual take over, hence the system must be dependable. The communications protocol is an integral part of the function. If it fails, coordination of the vehicles is lost. The back-up strategy is to rely on local sensor in each vehicle. However, the data quality of the sensors is lower hence the inter vehicle gap must be greater to maintain safety. In the next section we discuss how to assess functional safety for cooperative systems.

DISCUSSION WITH REFERENCE TO EXAMPLE (CONCEPT PHASES OF THE SAFETY LIFECYCLE)

Item Definition

Traditionally an item is defined such that it is fully confined to the functionality within one vehicle. This implies that if the vehicle is to take part in a cooperation, like driving in a platoon, whatever functionality realized outside the vehicle has to be considered as given, and is hence outside the scope of the analysis. An alternative is to also allow items spanning over several vehicles, thus being able to fully cover a cooperative functionality. A first question is to what extent this is just a matter of how to divide the total amount of analyses that have to be done, and to what extent it really matters what problems that are identified at all. Let us discuss this based on our example.

Consider our platooning functionality where several vehicles cooperate to perform a road train. With the traditional vehicle perspective, the item is confined to the functionality of fulfilling what is expected from one single vehicle to perform the agreed platooning functionality. With a cooperative perspective we instead allow the entire realization of the platooning functionality to become captured within the same safety case. As we will show in the following sections this difference may affect both what problems that are analyzed and what solutions that may be considered.

Hazard Analysis and Risk Assessment

In the phase of HA&RA, the safety goals are identified including their ASIL attributes. For this phase we identify two new implications of introducing cooperative systems. The first one is that an accident may cause much more harm, as a consequence of having several vehicles tightly coupled. The second implication comes from the fact that hazards may arise from failures in the cooperation itself, not due to the malfunctioning in a single vehicle.

The ASIL determination scheme of ISO 26262 combines severity (S), exposure (E), and controllability (C), in such a way that a decrease in one level of any of these parameters implies a decrease of one level of the ASIL attribute. When introducing cooperative functionalities it is mainly the severity factor that becomes a subject for discussion. For exposure there is nothing changed; it is still relevant how often a certain failure may become safety-related. Regarding controllability, one can note that almost per definition many hazardous events of cooperative functionalities will have the controllability class C3. This is because a high degree of cooperation between vehicles implies a high degree of autonomy, which in turn means that the driver is to a larger degree kept out of the loop. This does not affect the risk analysis because the C3 attribute is still relevant. The difference for the ASIL determination shows up when we look on how severe the consequence of an accident might become. For vehicles in a tight cooperation, one could set up scenarios where it is likely that more than 10 persons will get life-threatening injuries. We argue that this is one order of magnitude more severe than what is the case for S3 of today, and hence there is a need for a new severity class: S4. ISO 26262 is not restricted to injuries of single persons or to injuries sustained by the occupants of a single vehicle. However, we do not consider it appropriate to have a single severity class for hazardous events that threatens the lives of one person as well as many persons. Having the breakpoint between S3 and S4 at 10 persons is motivated by an order of magnitude difference. Similar breakpoints based on order of magnitude are proposed by ISO 26262 for distinguishing between classes of exposure and by the generic safety standard IEC 61508 [13] to distinguish between classes of severity.

A fourth severity class that accounts for hazardous events that involve the potential death of multiple persons have previously been used in the risk assessment methods of [14] and [15]. The two publications explicitly address cooperative functions. The scope of [14] encompasses all programmable electronic systems in vehicles, including cooperative functions. In [15] the risk assessment method includes safety aspects of security threats for cooperative functions.

In our example we may consider the hazard of: "unintended full braking in platooning at cruise speed". Having several cars performing the platooning, it is reasonable to assume that a sudden full braking of one vehicle in the middle may cause a lot of harm to the passengers in several cars behind. In order to indicate such a significant increase of severity, we propose the introduction of a severity class S4. The definition of S4 could be:

Def: S4 - More than 10 persons getting life-threatening injuries (survival uncertain) or fatal injuries.

Furthermore we propose to extend the already existing severity classes with definitions including more than 10 persons affected.

Def: S3 - Life-threatening injuries (survival uncertain), fatal injuries; OR more than 10 persons getting Severe and life-threatening injuries (survival probable).

Def: S2 - Severe and life-threatening injuries (survival probable); OR more than 10 persons getting Light and moderate injuries.

Following the pattern for ASIL determination, the introducing of S4 implies a higher level of safety integrity: ASIL E, according to Table 1.

Table 1. ASIL determination extended with severity class 4. The original table including severity class 1 to severity class 3 was adopted from ISO 26262 [1].

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
S4	E1	QM	A	B
	E2	A	B	C
	E3	B	C	D
	E4	C	D	**E**

Introducing ASIL E as an attribute on a safety goal does not necessarily imply that all method tables applicable for the product development phases of ISO 26262 have to be extended with recommendations for how to achieve ASIL E integrity. An alternative is to at least once apply ASIL decomposition so that no higher integrity than the today specified ASIL D will be required for product development. This would also be fully aligned with IEC 61508 where SIL 4 systems have to be built without any single point of failure. Giving the possibility to determine an ASIL E attribute to a safety goal, which has to be realized of elements fulfilling safety requirements of maximum integrity ASIL D, would also imply no single point of failure in the design.

Besides this first implication of cooperative functions to introduce S4, we also identify a second implication emanating from malfunctions in the cooperation itself, rather than from failures within a single vehicle.

Let us go back to the HA&RA of our platooning example. Having a traditional vehicle centric perspective would allow us to find safety goals (SGs) like:

- SG1: No unintended full braking in platooning at cruise speed, ASIL E (E4, C3, S4).

- SG2: No sudden unintended full acceleration in platooning at low speed, ASIL C (E4, C3, S2).

These safety goals also will be the found when having a cooperative perspective. However, from the cooperative perspective will also be identified another safety goal:

- SG3: No inconsistency in system state perception among platoon members, leading to full braking of some vehicles and cruise speed of others, ASIL E (E4, C3, S4).

This is a kind of safety goal that hardly can be identified from the vehicle centric perspective. It does not point out any particular failure in one single vehicle, but rather in the cooperation itself. We argue that such safety goals hardly may become identified when just considering the responsibility of how one single car fulfills its part of a cooperative functionality. In order to identify safety goals about the cooperation itself, the scope of the HA&RA have to be the cooperative function and not just the part of it confined to one vehicle.

Another way to describe this kind of safety goals on the cooperation itself is to describe them as restricting distributed hazards. It will be clearer what the distribution means, when braking down such safety goals to safety requirements restricting distributed failures. The way to model such distributed failures have been investigated in, e.g., [10].

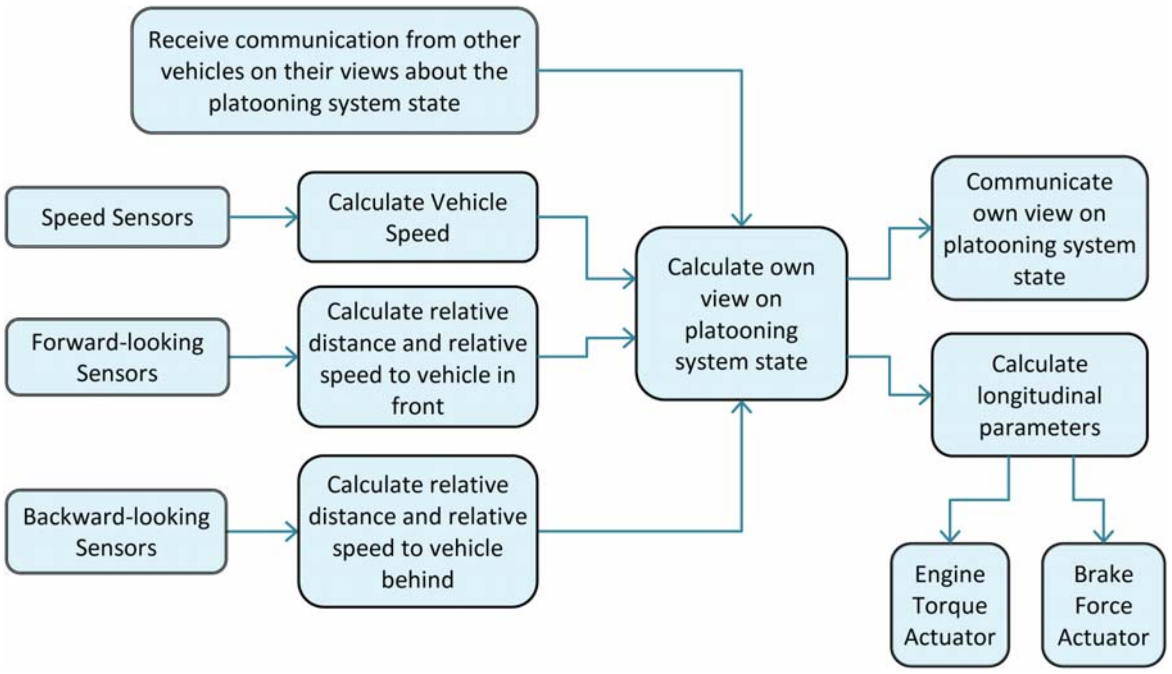

Receive communication from other vehicles on their views about the platooning system state
Speed Sensors
Calculate Vehicle Speed
Forward-looking Sensors
Calculate relative distance and relative speed to vehicle in front
Backward-looking Sensors
Calculate relative distance and relative speed to vehicle behind
Calculate own view on platooning system state
Communicate own view on platooning system state
Calculate longitudinal parameters
Engine Torque Actuator
Brake Force Actuator

Figure 3. Preliminary functional architecture in vehicle centric perspective.

To conclude the HA&RA phase we claim that there are two important implications from cooperative functions. The first is the need for a higher severity class (S4) leading to a higher safety integrity level (ASIL E). The second is the need to allow a cooperative perspective when identifying hazards and safety goals, i.e., having items spanning over several vehicles.

Functional Safety Concept

In the Functional safety concept, the safety goals are broken down to functional safety requirements. The context for these requirements is a preliminary functional architecture. Section 6 of part 8 of ISO 26262 lists a number of requirements that are applicable to safety requirements on any phase of the reference life cycle. For the discussion in this paper, some of the most important requirements are:

• ASIL attribute existence (8-6.4.2.5c)

• ASIL attribute inherited unless ASIL decomposition (8-6.4.2.2)

• Allocate onto architectural elements (8-6.4.2.3)

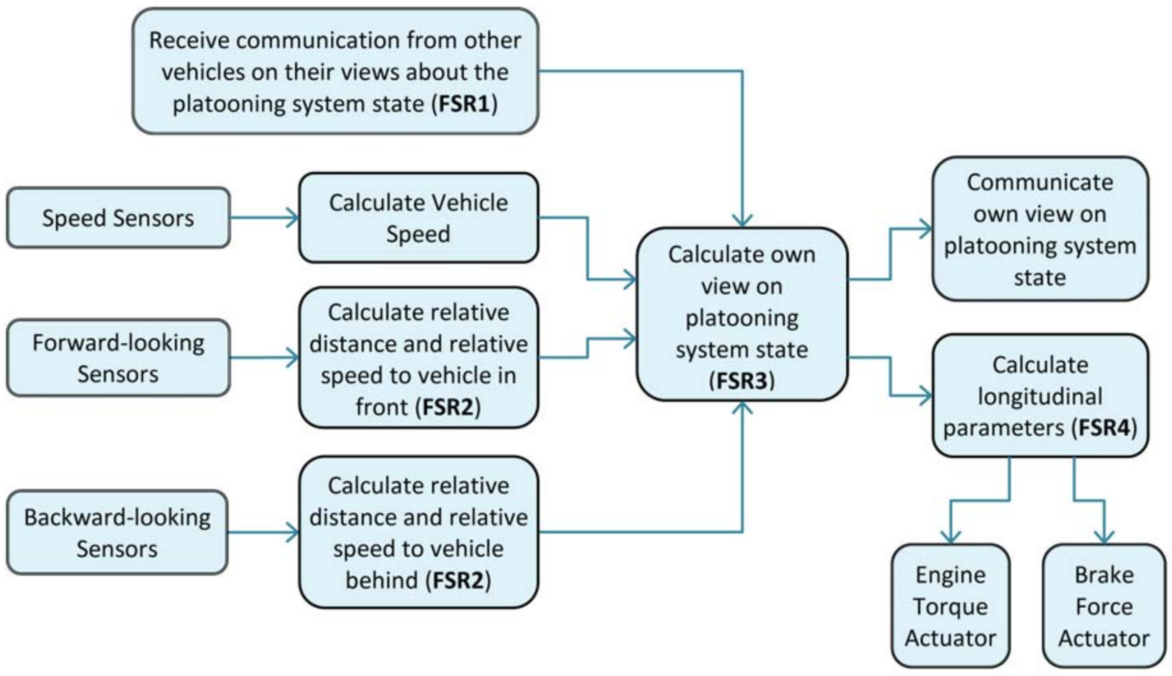
• Completeness w.r.t. the set of safety requirements (8-6.4.3.1c)

These four requirements on safety requirements imply the following. The ASIL attribute of a safety requirement indicates the requested level of confidence that an unwanted event will not happen, i.e., violation of that safety requirement. The allocation points to the architectural element for which this unwanted event shall not happen. Hence, each safety requirement implies a guaranteed absence of a certain failure, on an architectural element, with a certain confidence. Then in order to show completeness, you have to know that you have considered all relevant failures. This implies that there has to be a failure model telling about the set of relevant candidate failures to consider. A modeling pattern for establishing a failure model is given in [10]. If the required safety integrity levels are found too high (too expensive), ASIL decomposition is a way to decrease the required level of safety integrity by means of applying redundancy.

Let us use the example of platooning to elaborate the important things to consider for the functional safety concept of cooperative functions. Firstly, if we use a traditional vehicle centric view, the preliminary functional architecture could be depicted as in Figure 3.

This architecture is a pure functional one, which implies that there is no discussion on, e.g., topology or allocation of functions onto computational nodes. The sensors and the actuators are functional placeholders rather than modeling some physical ones. Note that the V2V communication here is modeled as remote sensors and actuators, respectively.

Furthermore, the architecture is confined to the item, i.e., to the fulfillment of one car performing its part of a cooperative functionality. We focus on the first safety goal (SG1). When breaking this down we may identify a number of functional safety requirements and allocate them as depicted in Figure 4.


Receive communication from other vehicles on their views about the platooning system state (**FSR1**)
Speed Sensors
Calculate Vehicle Speed
Forward-looking Sensors
Calculate relative distance and relative speed to vehicle in front (**FSR2**)
Backward-looking Sensors
Calculate relative distance and relative speed to vehicle behind (**FSR2**)
Calculate own view on platooning system state (**FSR3**)
Communicate own view on platooning system state
Calculate longitudinal parameters (**FSR4**)
Engine Torque Actuator
Brake Force Actuator

FSR1: No misinterpretation of received communication as request for full Brake force, ASIL E
FSR2: No underestimation of relative distance or of relative speed leading to full Brake force request at high speed, ASIL E
FSR3: No mistake in system state calculation leading to full Brake force request at high speed, ASIL E
FSR4: No unintended full Brake force request at high speed, ASIL E

Figure 4. Allocation of functional safety requirements to the preliminary functional architecture in vehicle centric perspective.

Note that the picture is not showing a complete list of safety requirements that together can be shown to fulfill the safety goal SG1. As the functional safety requirements each inherits an ASIL E attribute, all these architectural elements are candidates for redundancy patterns, i.e., ASIL decomposition.

In order to argue for completeness, we need firstly to show that we have considered all failures of all architectural elements as candidates as allocation target for a safety requirement. Secondly we need to show that we have considered every possible kind of failure, i.e., that we are complete with the respect to any reasonable failure model. Depending on how we confine our item, what is considered as complete will differ. Let us therefore instead look at a preliminary architecture from a cooperative perspective as depicted in Figure 5.

As a start we can look at the same example safety goal, SG1. In order to achieve completeness, we now also need to list a functional safety requirement on the V2V communication between the cars. This means that even if we aggregate all functional safety requirements of the different cars, the vehicle centric perspective will miss the functional safety requirements on the V2V communication between the cars. By using a cooperative perspective we will also catch the need for such requirements in order to show completeness with respect to the safety goals. In the vehicle centric perspective we must assume that whatever is specified for V2V communication will be sufficient for achieving safety. However, there is no way to let the specification of the V2V communication (including choice of protocols for assuring consensus and consistency) become a part of a safety case showing that reasonable failures in the V2V communication will not violate any safety goal. On the other hand, in the cooperative perspective, the safety requirements allocated on the V2V communication gives a reason to investigate what kind of redundancy that is needed to enable safety arguing.

As a consequence of the differences in the HA&RA, the functional safety concept in the vehicle centric perspective will not analyze the breakdown of safety goals addressing cooperative issues. This means that the breakdown of SG3 will not be part of any functional safety concept if a vehicle centric perspective is chosen.

An implication of the conclusion that the vehicle centric perspective is insufficient, is that parts of 26262 life cycle activities have to be agreed among the companies that implement vehicles that are specified to cooperate in applications of Type 3 and 4. This means that when defining the functional specification for a cooperative application, HA&RA has also to be done from a cooperative perspective

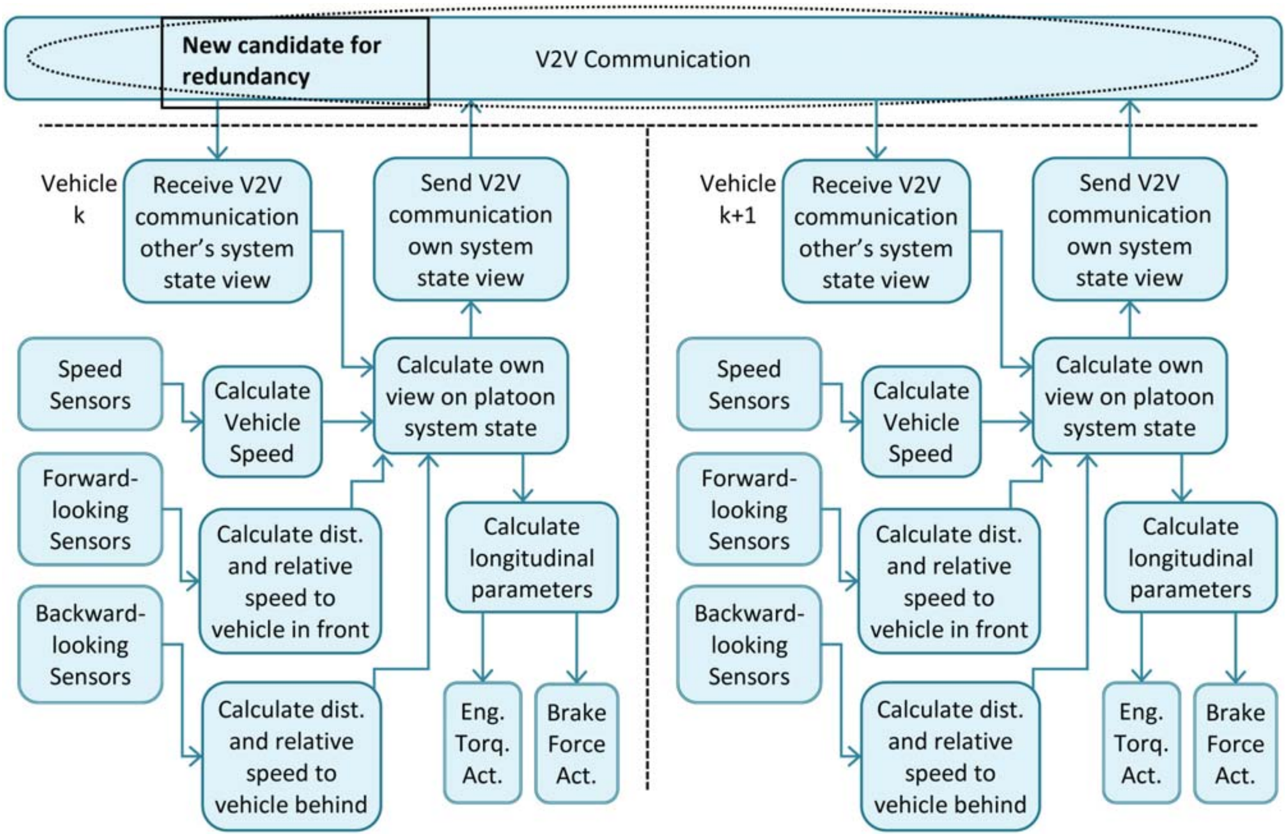

New candidate for redundancy
V2V Communication
Vehicle k
Receive V2V communication other's system state view
Send V2V communication own system state view
Speed Sensors
Calculate Vehicle Speed
Calculate own view on platoon system state
Forward-looking Sensors
Calculate dist. and relative speed to vehicle in front
Calculate longitudinal parameters
Backward-looking Sensors
Calculate dist. and relative speed to vehicle behind
Eng. Torq. Act.
Brake Force Act.
Vehicle k+1
Receive V2V communication other's system state view
Send V2V communication own system state view
Speed Sensors
Calculate Vehicle Speed
Calculate own view on platoon system state
Forward-looking Sensors
Calculate dist. and relative speed to vehicle in front
Calculate longitudinal parameters
Backward-looking Sensors
Calculate dist. and relative speed to vehicle behind
Eng. Torq. Act.
Brake Force Act.

Figure 5. Preliminary functional architecture in cooperative perspective.

together with a certain amount of the life cycle phases that affects the cooperative application specification. As mentioned before this may include ASIL decomposition of the safety requirements on the cooperation, by redundancy techniques in V2V, potentially implemented in different protocol levels. Examples of such techniques are membership protocols and protocols for assuring timely consistency of critical global state variables.

Given that this is done, each company can then go on with a life cycle based on the traditional vehicle centric perspective. However, it is important to capture both hazards and possible risk reductions related to the cooperation itself, from a cooperative perspective.

As an example of ASIL decomposition for a safety goal with an ASIL E attribute we use SG3. With the cooperative perspective we can allocate functional safety requirements to the V2V communication. One of the redundant requirements is therefore: Vehicles in the platoon must monitor each other and in case a vehicle detects unintentional braking at high speed of the vehicle ahead it shall communicate this to the other vehicles that then initiate braking. This requirement is given an ASIL B attribute. The second redundant requirement shall be independent of V2V communication and implemented in each vehicle: No unintentional full brake application of a vehicle at high speed. This requirement is assigned an ASIL C attribute. Together, these two redundant requirements achieve a combined safety integrity level of ASIL E and fulfill SG3. It can be noted that the second requirement most likely also appears as a result of safety goals related to non-cooperative items such as the base brake.

To conclude the Functional safety concept, we note that the preliminary architecture is confined by the item definition. Having a vehicle centric perspective excludes those architectural elements that do not belong to any single vehicle. This implies that in order to identify functional safety requirements on the communication between vehicles, we need a cooperative perspective. If the V2V communication is outside the scope of the functional safety concept, there will be no way to reason about what is a sufficient and still efficient redundancy strategy to achieve ASIL decomposition down to a level that can be proven fulfilled in the product level phase activities.

SUMMARY/CONCLUSIONS

In this paper we have investigated how two alternative perspectives (a single vehicle perspective and a cooperative perspective) on cooperative functionalities influence design and assessment of functional safety. First we identify that ISO 26262 takes a single vehicle perspective where an item is confined by the boundaries of a single vehicle. Next we show, using a platooning example, that this perspective is not suitable for cooperative functionalities because of issues with HA&RA and functional safety concept. As a solution to the issue we propose a cooperative perspective, where boundaries of the item extend beyond the individual vehicles and encompass the complete cooperative functionality. In the cooperative perspective all vehicles that are part of the cooperative functionality are also included in the safety lifecycle. We have shown that this prevents the underestimation of risks, identifies the need for additional safety goals, and facilitates the use of inter-vehicle communication as a means of redundancy. A new severity class (S4) and an additional ASIL (ASIL E) is introduced to accommodate for those hazardous events with the highest risk, involving multiple vehicles and a potentially large number of fatal injuries.

We argue that ASIL E can and should be achieved using ASIL decomposition and the currently recommended product development phases for ASIL A to ASIL D. This is supported by the cooperative perspective, where inter-vehicle communication is an architectural element and a candidate for redundancy. The benefits of ASIL decomposition may also motivate the use of redundancy in inter-vehicle communication to reach lower levels than ASIL E (i.e., ASIL A-D).

If redundancy in the communication between vehicles is used in ASIL decomposition, then functional safety will depend on the cooperation of several vehicles (vehicles depend on other vehicles to reach safety goals). We assume that an agreement needs to be established between the developers (vehicle manufacturers and subcontractors) of cooperative systems in order to define responsibilities for safety. This would support functional safety assessment and be used in the safety case for a cooperative system. Future work is necessary to establish a suitable procedure for specifying such an agreement.

The scope of this work has been limited to state-of-the-art of functional safety for road vehicles. The challenges with respect to cooperative systems have similarities with aircraft collision avoidance system and future air traffic management approaches. Future work should therefore investigate these similarities further.

REFERENCES

1. ISO International Standard, "Road vehicles - Functional safety," ISO Standard 26262, Rev. Nov. 2011.

2. Bergenhem, C., Huang, Q., Benmimoun, A., and Robinson, T., "Challenges of platooning on public motorways," presented at 17th World Congress on Intelligent Transport Systems, South Korea, Oct. 25-29, 2010.

3. Lygeros, J., Godbole, D. N., and Broucke, M., "A Fault Tolerant Control Architecture for Automated Highway Systems," *IEEE Transactions on Control Systems Technology*, 8(2):205-219, 2000, doi: 10.1109/87.826792.

4. Papadimitratos, P., La Fortelle, A., Evenssen, K., Brignolo, R., and Cosenza, S., "Vehicular communication systems: Enabling technologies, applications, and future outlook on intelligent transportation," *IEEE Communications Magazine*, 47(11): 84-95, 2009, doi: 10.1109/MCOM.2009.5307471.

5. Willke, T., Tientrakool, P., and Maxemchuk, N., "A Survey of Inter-Vehicle Communication Protocols and Their Applications," *IEEE Communications Surveys & Tutorials*, 11(2):3-20, 2009, doi: 10.1109/SURV.2009.090202.

6. Kianfar, R., Augusto, B., Ebadighajari, A., Hakeem, U. et al., "Design and Experimental Validation of a Cooperative Driving System in the Grand Cooperative Driving Challenge," *IEEE Transactions on Intelligent Transportation Systems*, 13(3):994-1007, 2012, doi: 10.1109/TITS.2012.2186513.

7. ETSI Technical Committee Intelligent Transport System (ITS), "Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Definitions," ETSI TR 102 638 (V1.1.1), June 2009.

8. Casimiro, A., Kaiser, J., Karlsson, J., Schiller, E. M. et al., "KARYON: Towards Safety Kernels for Cooperative Vehicular Systems," presented at 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2012), Canada, Oct. 1-4, 2012.

9. Bergenhem, C., Pettersson, H., Coelingh, E., Englund, C. et al., "Overview of Platooning Systems," presented at ITS World Congress, Austria, Oct. 22-26, 2012.

10. Bergenhem, C., Johansson, R., and Lönn, H., "A Novel Modelling Pattern for Establishing Failure Models and Assisting Architectural Exploration in an Automotive Context," Proceedings 31st International Conference SAFECOMP, Germany, Sept. 25-28, 2012, doi: 10.1007/978-3-642-33678-2_21.

11. Michael, J. B., Godbole, D. N., Lygeros, J., and Sengupta, R., "Capacity Analysis of Traffic Flow over a Single-Lane Automated Highway System," *ITS Journal*, 4(1-2):49-80, 1998.

12. Grand Cooperative Driving Challenge, "GCDC homepage," http://www.gcdc.net, Oct. 2012.

13. IEC International Standard, "Functional safety of electrical/electronic/programmable electronic safety-related systems," IEC Standard 61508, Edition 2.0, Rev. Apr. 2010.

14. MISRA, "Guidelines for safety analysis of vehicle based programmable systems," MIRA Limited, Warwickshire, ISBN 978-0-9524156-7-1, 2007.

15. Henniger, O., Apvrille, L., Fuchs, A., Roudier, Y. et al., "Security requirements for automotive on-board networks," presented at 9th International Conference on Intelligent

Transport Systems Telecommunications, (ITST), France, 2009, doi: 10.1109/ITST.2009.5399279.

CONTACT INFORMATION

Josef Nilsson
SP Technical Research Institute of Sweden
Box 857
SE-50115 Borås
Sweden Tel: +46-10-5165317
josef.nilsson@sp.se

ACKNOWLEDGMENTS

This paper was prepared, in part, within the projects SARTRE (reference no.: 233683) and KARYON (reference no.: 288195), both funded by Seventh Framework Programme (FP7/2007-2013) of the European Commission. This publication reflects the authors' views. The European Community is not liable for any use that may be made of the information contained herein.

This page is intentionally left blank.

Annex B An Architecture Pattern enabling safety at Lower Cost and with Higher Performance

Rolf Johansson, Jörg Kaiser, António Casimiro, Renato Librino, Kenneth Östberg, José Rufino, and Pedro Costa, "An Architecture Pattern enabling safety at Lower Cost and with Higher Performance". In proceedings of Embedded Real Time Software and Systems (ERTS2) 2014.

This page is intentionally left blank.

An Architecture Pattern Enabling Safety at Lower Cost and with Higher Performance

Rolf Johansson, Jörg Kaiser, António Casimiro, Renato Librino, Kenneth Östberg, José Rufino, and Pedro Costa

Abstract— **In both avionic and automotive systems, it might become very costly and/or restricting the functional performance, to prove functions safe in all operational conditions and for 100% of the mission time. This is especially true if the quality of sensor data and of communication data may vary very much. One way to solve this trade-off paradox is to leave part of the safety assessment from design-time to run-time. This paper proposes a general architectural pattern for this, and also how to instantiate this pattern in Integrated Modular Avionics (IMA) for the avionic domain, and in AUTOSAR for the automotive domain. The solutions imply some extensions of ARINC 653 and of AUTOSAR respectively, but they are not in conflict with the existing concepts. The proposed solutions are also fully in-line what is prescribed by the standards for functional safety of the two domains.**

***Index Terms*—Safety Integrity, IMA, AUTOSAR,**

I. Introduction

In both the industry fields of Avionics and of Automotive, there has been established a norm of having integrated, rather than federated, Electrical/Electronic (E/E) architectures. In the avionic field this is called Integrated Modular Avionics (IMA) [6] and is today often following the ARINC 653 specifications [7], and in the automotive field it is the AUTOSAR specifications constituting the state-of-practice.

One advantage with an integrated architecture is that it is possible to increase the number of vehicle functions and still decrease the number of computing nodes, often called LRUs (Line Replaceable Units) or ECUs (Electronic Control Units). Different functions may be realized by architectural elements (sensors, actuators, computing components, communication components, etc.) that may be shared among several functions. However, in the transition from a federated to an integrated pattern, the way to ensure functional safety has become more complex. Instead of directly transfer the safety arguing responsibility to a node supplier, the integrated pattern calls for a component-based approach in both the design itself and in the safety case generation. Furthermore, achieving functional safety is a goal that often is in conflict with high performance and low cost.

In this paper we outline an architectural pattern that is possible to apply for both IMA and AUTOSAR, and that helps to resolve the paradox of getting functional safety together with low cost and high functional performance.

II. Arguing safety

The problem how to prove that a solution is functionally safe, is defined specifically by applicable standards in each domain. For road vehicles, functional safety is defined by ISO 26262 [10], and the instruction how to apply component-based safety arguing is so far limited to what is stated as Safety Element out of Context (SEooC) in the informative part 10 of the standard. For the avionic domain there are several applicable standards. The IMA perspective is found in RTCA/D0-297: "Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations". This standard refers to other standards such as SAE International's Aerospace Recommended Practice (ARP) 4754 on "Certification Considerations for Highly-Integrated or Complex Aircraft Systems", and SAE ARP 4761 on "Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment". These in turn refer to RTCA/DO-178: "Software Considerations in Airborne Systems and Equipment Certification".

On a high level, one can say that arguing safety is similar in the two domains. They both rely on a Hazard Analysis & and Risk Assessment (HARA) of the vehicle functions, resulting in required risk reduction by means of safety integrity levels for the realizing architectural components. In the automotive domain the safety integrity levels are called Automotive Safety Integrity Level (ASIL) [10], and in the avionic domain they are called Development Assurance Level (DAL) [12]. The default alternative is to use a fault avoidance argumentation based on the fact that all used components, each conform to the required safety integrity level. This can be combined with a fault-tolerance argumentation based on introducing redundancy and hence lower the required integrity level of the components. To integrate components having different integrity levels on the same platform, freedom of

This work was supported by the project KARYON (reference no.: 288195), funded by Seventh Framework Program (FP7/2007-2013) of the European Commission.

Rolf Johansson and Kenneth Östberg are with SP Technical Research Institute of Sweden (e-mail: rolf.johansson@sp.se, kenneth.ostberg@sp.se).
Jörg Kaiser is with Otto von Guericke Universität Magdeburg (email: kaiser@ivs.cs.uni-magdeburg.de)
António Casimiro and José Rufino are with Faculdade de Ciências da Univesidade de Lisboa (email: casim@di.fc.ul.pt, ruf@di.fc.ul.pt)
Renato Librino is with 4SGroup (email: renato.librino@4sgroup.it)
Pedro Costa is with GMV Skysoft (email: pedro.costa@gmv.com)

interference has to be shown. This is a major concern in RTCA/DO-297.

III. Determine the Required Risk Reduction (Safety Integrity Level safety)

In both domains, the determination of safety integrity level is based on a severity classification of the possible failures of the vehicle function under consideration. In the avionic domain this directly leads to one of the Development Assurance Levels in the interval between A (required by a catastrophic failure) and E (when there is no safety effect required). In the automotive domain the severity classification of the vehicle function has to be evaluated together with assumptions on how often such a failure is critical among the driving scenarios, and with an assumption how well the driver can compensate for the failure. These three factors together, lead to an Automotive Safety Integrity Level in the range between ASIL D (highest requirement on risk reduction) and QM (no requirement on risk reduction).

In both domains, a high requirement on risk reduction implies a high safety integrity level requirement (formulated by DAL or by ASIL) on the used components. It is needed to show that the vehicle and its components fulfill all requirements on safety integrity levels. This assessment is completely done in design-time, and shows that the vehicle always is functional safe with respect to the complete scope of the functions considered in the Hazard Analysis.

IV. Extending the architectural pattern

Building vehicles where the complete scope of all functions are proven safe in all operational conditions and for 100% of the mission time, might become very costly and/or restricting the functional performance. One way to solve this trade-off paradox is to leave part of the safety assessment from design-time to run-time. By letting the architecture itself in run-time measure the provided safety integrity levels from the components, it can enable adjustment of the different functions such that their required safety integrity levels are met. Let each vehicle function have a number of predefined levels of performance (levels of service), for which the resulting hazard analyses are different. If we then in run-time can measure for which levels of service/performance the safety integrity requirements are fulfilled, the vehicle can be proven safe once we can guarantee that all functions always are forced to a level of service/performance that is considered safe. What we need for this is an architectural pattern enabling the complete measuring of the safety integrity levels of the architectural elements, and a way to ensure that all functions operates on a level of service/performance that can be proven safe.

This paper shows the general architectural pattern, and also outlines how it can be instantiated into the state-of-practice in the integrated architecture of the domains of automotive (AUTOSAR) and of avionics (IMA realized by ARINC 653), respectively.

A. Design-Time vs. Run-Time

Even if the strategy proposed in this paper is that there is a run-time check of what safety integrity levels that are met in order to match with the appropriate performance level, all possible results have to be assessed in design-time. In both the avionic and the automotive domain, a complete functional assessment has to be done in design-time according to the respective reference life-cycles. The solution presented in this paper is fully aligned with this. When we say that we leave part of the safety assessment to run-time, this implies that all the possible alternatives for performance levels and also the mechanisms to determine the right levels, all are assessed in design time.

B. Relation to functional safety standards of today

In both DO178 and in ISO26262, there is a concept of integrity levels (DAL / ASIL), to allocate requirements on the reference life cycle in order to argue sufficiently absence of systematic design faults. When we suggest having several levels of performance implying different required safety integrity levels on the output of some components, this implies that the DAL/ASIL applicable for the design of each component will be the highest safety integrity level among the possible ones. The concept of adjusting the levels of performance to the run-time available safety integrity levels is hence not primarily applicable to handle systematic design faults, but to take care of the varying quality of data due to the varying amount of redundancy sources and of varying quality of sensors and communication links.

Especially from the software design point of view, we assume that what is prescribed as needed to argue for the highest applicable safety integrity level, still must be implemented as well in the application components as in the platform software (ARINC653/AUTOSAR). This requirement of design according to the highest safety integrity level is of course applicable to the redundancy mechanisms checking the quality of data, and to the platform mechanisms determining the appropriate level of service/performance. This is further elaborated in the section below: A general pattern.

The consequence of having a number of different levels of performance is that the Hazard Analysis and Risk Assessment has to be done completely for each considered level. The stages prescribed in ARP4754 or ISO26262 part 3 thus have to be done not only for the functions, but for each level of performance for each function. Furthermore, each transition between two levels of performance has to be considered, as elaborated below in the section: Scalability and timing.

Regarding the safety standards applicable today in the domains of avionic and automotive, we conclude that they are fully in-line with both the concept of several levels of performance, and with the architectural pattern we outline in this paper.

C. *Relation to existing patterns for mixed criticality*

Mixed criticality [5] is the concept of allowing applications with different levels of criticality (safety integrity) to co-exist on the same system. In both ARINC 653 and in AUTOSAR there are mechanisms to deal with this problem. The assumed problem to deal with is to guarantee freedom of interference between application components designed to different levels of safety integrity. The solutions in both ARINC 653 and in AUTOSAR are to provide mechanisms for handling time and space partitioning.

However, when introducing several levels of performance, each implying different requirements on safety integrity levels of the output of some components, this is not the classical problem of mixed criticality. Even if we have a mixture of different levels of safety integrity among the components co-existing on the same platform, their requirements on the safety integrity of the absence of systematic design faults are all on the same level (the highest among alternatives). Hence we do not need to prove freedom of interference between components designed according to different DAL or ASIL, because that is not the case. All components used for a certain function are designed according to the safety integrity level that is applicable for the highest level of performance.

The mechanisms for enabling time and space partitioning are of course still important when arguing safety, even in our proposed pattern, but they are not affected by the introduction of several levels of performance with different safety integrity level requirements on some data signals.

D. *Scalability and timing*

Given the fundamental idea of leaving part of the safety assessment for the run-time, raises potential issues of scalability and timing.

Scalability issues stem from the fact that some system resources will be required for performing the run-time safety assessment. For instance, it will be necessary to collect measurements of available integrity levels, and it will be necessary to store information (defined in design time) concerning the safety integrity requirements for each level of service. This has essentially practical implications (availability of enough memory and computing power), which could limit the applicability of the approach. Fortunately, the effective requirements grow linearly with the number of components for which integrity has to be assessed, which is also limited by the available resources. In fact, the additional resources required for assessing safety in run-time are necessarily a very small fraction of the resources required by the components itself.

Timing issues are more important in this context because it is necessary to argue about functional safety, for which they have to be considered. As mentioned before, in design time it is necessary to assess, for each function, that it will perform safe in each possible performance level. But this is not enough. Given that, for each function, there will be run-time changes of the performance level, the analysis must take into account the time that it takes to complete these changes. For instance, if in run-time it is detected that some component is not performing with the required integrity level, then it will be necessary to change the performance level of all the functions that are affected by this integrity degradation, and this has to be done within some limited amount of time. Otherwise, the functions would continue to perform in some inadequate level for an uncertain amount of time, clearly outside the safety analysis performed in design time.

The general pattern must hence provide the means to address these timeliness requirements. In addition, it is necessary to discuss the implications of the (bounded) amount of time that is necessary to change the performance level of some function. A sufficient condition for ensuring timely detection of changes in the safety integrity levels, and consequent timely change of performance level, is that it is possible to perform a timing analysis of all the involved system components, deriving upper execution bounds. In particular, this includes the component responsible for performing the safety assessment, which is always involved in the process. If some function has to be performed with some minimal performance level, then it must also be possible to perform such timing analysis for all the implied components. The time that will be necessary to switch among different performance levels will typically be close the execution periods of the system components. In comparison with the typical latency of physical processes (such as braking, deviating from an obstacle, etc), these periods are much smaller. This means that the safety analysis will be, for the relevant part, still valid. In any case, the time that it takes to perform a change in the performance level can also be considered in the design of the functions, so that this is accounted in safety margins.

V. A General Pattern

Each vehicle function is realized by a set of interconnected sensors, actuators and software components. The software components have an interface making them possible to be allocated on any platform node. The outputs of every sensor, and of every software component, are duplets consisting of both the nominal output and of an attribute stating the estimation of the corresponding safety integrity level. All redundancy mechanisms in the architecture such as: sensor fusion, voting, consistency checking, etc., are evaluating the consistency of the nominal values and calculates a resulting determination of the safety integrity level value. All safety integrity level values of a computing node are checked by a safety manager that is part of the platform specification. The safety manager compares in run-time that each provided safety integrity level of every output value, is high enough for the current scope of vehicle functions. If some of the provided safety integrity levels are too low, the safety manager tells the application mode managers to change to a level where the respective functions are considered safe. If all provided safety integrity levels are high enough for a higher level of service/performance than the actual for some functions, the safety manager tells the application mode managers to change accordingly.

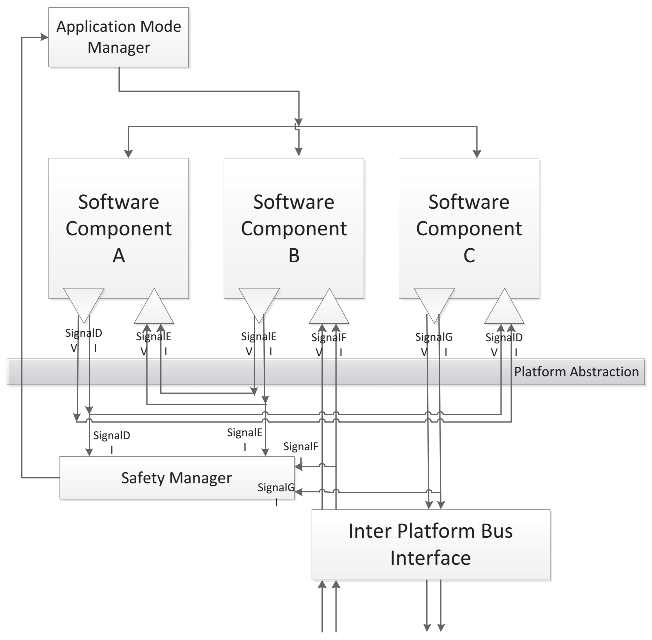


Fig. 1. The figure of a general architecture pattern. Each software component communicates via ports with signals. Each signal is a duplet with a nominal value (V) and a safety integrity level attribute (I). The safety manager checks all safety integrity level values, and decides what function modes that are safe. The mode manager tells each software component about the mode decisions

A. Derivation of the Safety Integrity Attributes

As argued before, the extension of safety assessment from design-time to run-time allows for relaxing overly restrictive assumptions about the required integrity status of the overall complex control systems. However, it requires the continuous monitoring and evaluation of the integrity attribute during run-time. Fig.1 shows that for every signal generated by a component, there is a complementary output that provides an integrity level attribute. Roughly, this attribute represents a measure for the integrity of the respective signal produced by a component. When checking against allocated safety requirements, this estimation needs to be done against the defined discrete levels of the applicable standard. For both the standards we have five levels to consider. However, when deriving the safety integrity attributes we may use more fine-grained estimations. In this paper we call such a more fine-grained value, the Validity of a signal.

The assumed structure of a complex component comprises the acquisition and computational components from a sensor that captures a real-world entity to the component that outputs an application relevant data element (signal). The architectural element outputs the nominal signal (V) together with an integrity attribute (I) that enables the safety assessment at run time. To generate the integrity attribute, the component needs a self-assessment mechanism. Further, a nominal value affected by a failure may pass a filter to mitigate or mask the effect. We therefore distinguish between a detection mechanism and a filter mechanism. The checking mechanism detects a failure without affecting the respective signal. It modifies the integrity attribute only. The filter is a general abstraction of a component that mitigates or eliminates the effect of a failure. Typically, detectors and filters are integrated in a fault-tolerance mechanism. We separate the aspects of awareness and treatment because these are different concerns and the separation allows for more freedom of design. e.g. omitting a filter completely in the component for handling the failure in a subsequent stage. Fig. 2 depicts this general structure of such an element.

The run-time assessment mechanisms are based on the specification and quantification of design-time assumptions. During design-time an engineer has to answer questions like "which failure types are affecting the components and what is their impact?", "How are these failures detected and how good the detection mechanisms need to be?" and " How is the data conditioned and filtered to compensate the effects of failures?". Based on these assumptions the engineer adjusts the quality of the component's outgoing data at run-time to the integrity requirements. In our approach, these engineering assumptions are quantified to allow a comprehensible assessment. Assumptions are quantified in a failure model, a quantification of the detection capabilities and the filter characteristics. This is particularly needed when such a component is used in a larger setting or will be reused in another design. As an example we examine a typical component where the input data is provided by a sensor.

We distinguish two flows of information in Fig.3. The lower part generates the nominal data output while the upper part is devoted to the calculation of the integrity attribute. In this paper we will focus on the definition and transformation of the parameters defining the integrity attribute.

We are considering a data centric failure model [2] which specifies failures in terms of how they affect the data e.g. according to an anticipated signal behaviour. The starting point is the identification of relevant sensor failures. Sensors deliver continuous values that may be affected by e.g. noise, offsets, spikes and outliers. A detailed discussion about sensor failure modes is given in [3]. An assessment vector quantifies for each failure type a number that characterizes the anticipated severity and the occurrence probability of this failure. This is comparable to a risk priority number in the FMEA (Failure Mode and Effects Analysis) scheme [4]. Different from FMEA, we provide a scheme that allows combining multiple failure types and using this for run time assessment, while FMEA maps multiple failure types to the static worst case risk priority number.

For describing the detector and filter characteristics we provide transformation matrices that modify the assessment vector for the respective stages. In case of a detector, the matrix specifies the ratio of correctly detected failures versus the wrongly and not detected failures. This statically sets the upper and lower bounds for the integrity attribute and modifies the assessment vector accordingly. The filter matrix defines the impact that the filter may have on the signal, i.e. the degree of suppressing a faulty signal. A filter that operates as a failure masking mechanism will raise the lower bound for the validity because it eliminates faulty values. Applied to the assessment vector it modifies the respective elements related to the affected failure types. The assessment vector finally holds for each failure type an element that describes which effect this specific failure will have on the final integrity attribute. It should be noted that this number includes the capabilities of the detector and filter with respect to the particular failure. The integration stage collapses the vector representation to a single scalar integrity attribute. This stage uses a selection vector that holds weights for each failure type and therefore allows a further restriction to relevant failure types. E.g. for long term navigation, single outliers of a localization sensor may not be as relevant as a constant offset failure. However, outliers may have a high impact on the validity because of their amplitude. Thus, an outlier would decrease the integrity attribute to an inacceptable low level although it would not be relevant. Another application may need a high validity of differential positions. Here, constant offsets would not play a major role. The selection vector can adjust these different application needs. In the end, we obtain what we call the system validity, a measure of how good the detection and filter mechanisms will deal with failures.

So far, all the information that is captured in the assessment vector, the transformation matrices and the selection vector is available at design time and quantifies of the engineering assumptions about relevant failure types, their impact, the quality of the detection mechanism and the power of the filter in suppressing the effect of failures.

In the conventional approach, the outcome of this analysis would be compared to a required integrity and in the case of a match, the design would be accepted. This implies that the assumptions about the failures, the detection and filter capabilities are worst-case assumptions and require a substantial amount of resources to keep the bounds. If the design does not fulfill the static worst-case requirements, the design needs to be changed. This may require substantial redundant resources or more expensive components even though the actual operation would not even come close to the worst-case bounds in by far the most cases. In the proposed architectural pattern, the numerical representation of the integrity is exploited to detect at run time whether the component will meet the integrity requirements. If the dynamically derived integrity attribute falls below a certain threshold, we are able provide the countermeasures on a higher level, i.e. on the level of provided services that may need to be degraded (see also the example in the next section B).

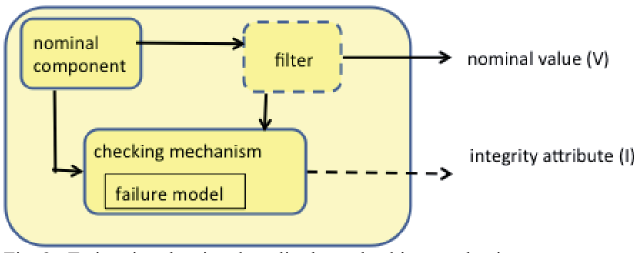


Fig. 2. Estimating the signal quality by a checking mechanism.

At run time, the detector will provide a result for each failure that it is able to detect. These outcomes are stored in a vector with the same dimension as the assessment vector. The elements of the assessment vector are applied as weights to this vector to form the validity vector. The validity vector represents the actual estimated validity as a result of the detector stage. It is transferred to the filter stage where a similar calculation is performed. The final validity vector holds a dynamic estimate about the validity of the generated nominal data item with respect to each failure type. Finally, this vector is converted into a single scalar number which represents the integrity attribute. The details of the assignment of values and the operations defined by the failure algebra are beyond the scope of this paper. They are provided in [5].

B. Relating the validity estimation to the Integrity Attributes

In a safety-critical system, the hazard analysis will result in a set of functional failures that should not occur. In the coming phases these restrictions are broken down to what failures should not occur for the elements in the chosen architecture. Still, all these failures are restricted by the safety integrity level attributes telling how sure one have to be that these failures will not occur. The question is how sure we can be at run-time about the absence of these failures, if we cannot guarantee at design-time that this will always hold. We now will show, how the dynamic validity can be exploited to quantify these requirements. Let us first use a simple data centric model of failure. Failures may be specified in terms of its amplitude, e.g. the error exceeds ±5% of the true value or in duration e.g. that they will not last for more than 10ms. According to the goal of our architectural pattern, we enable the dynamic change of the Level of Service (LoS) to maintain

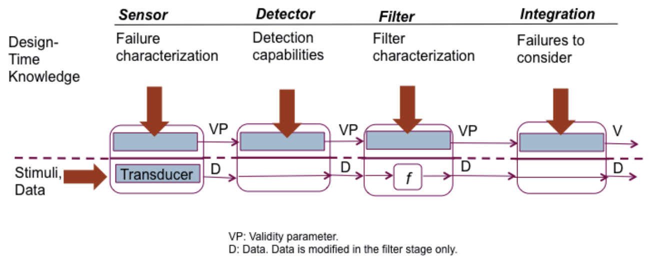
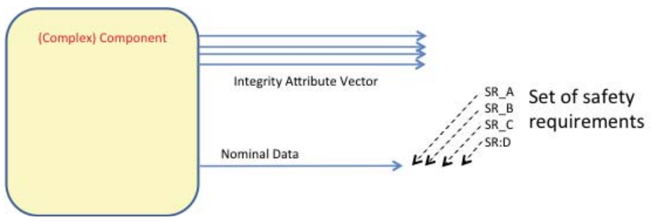
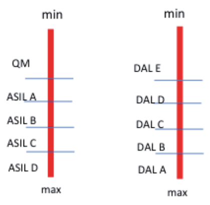


Fig. 3. Computational chain for a sensor processing component.

a required level of safety. For the information generated by a component that contributes to a function we need specifying different levels of confidence that the data value is within certain bounds. Fig. 4 illustrates this.



a.) A complex component has to meet several safety requirements



b.) Example of defining 4 thresholds for different safety integrity levels in the automotive and avionics terminology

Fig. 4 Example of a complex component used in applications running in multiple integrity levels

The complex component provides a nominal output that is subject of multiple safety requirements ranging from the highest ASIL D to QM. Integrity attributes are provided at separate outputs. They are represented as an integrity attribute vector that holds for each failure the confidence that it does not exceed the predefined bounds. Considering the thresholds in Fig. 4b, we model failure classes that e.g. do not deviate more than ±2%, ±5%, ±10%, ±20% and ±30% from the true value for ASIL D, C, B, A, and QM respectively. The integrity attribute vector holds the respective confidence values that these values are indeed within the specified bounds. If the confidence drops below a certain value in the most demanding safety class, this confidence value may well be within the bounds of a lower one. This would be reflected by the integrity attribute of the respective failure class. In this case, that the integrity attribute is too low, it may be necessary to switch to another level of service.

The notion of validity that was introduced before can be mapped to this more safety centric perspective. Given that the requirements of the safety integrity level concerning the error of the nominal value are specified, the validity calculus allows defining the failure characteristics through the computational chain. The failure model will be adapted to the thresholds of the amplitudes. Consider that the failure may not exceed ±2% of the true value and we want to be sure about this at a certain confidence level. This affects the choice of the sensor, the detector and the filter. The validity calculus allows to quantify these effects. At the sensor, an assessment vector represents the sensor characteristics for each failure type in terms of amplitude and of occurrence probability. If the requirements specify a very low margin on the amplitude of failure, noise, small offsets and drift have to be considered. We have to consult the failure model to see which failures are effective in such tight bounds. This will be analysed statically for each deviation addressed in the example above. In the end, we will have the design-time assurance that at run-time, under all anticipated conditions, the output of the component will have the integrity level indicated in the respective integrity attribute vector. If the safety requirements are very high, as it is e.g. required for ASIL D, the quality of the sensor and the mechanisms to mask a failure to always be within the bounds must be very high. I.e. we may need massive redundancy of expensive components. The problem is, that even a less expensive sensor will deliver results within tight bounds most of the time with sufficient probability. Detection and filters will detect or suppress failures sufficiently most of the time with sufficient probability. Thus the probability of a failure at run time will meet the ASIL D requirements most of the time but violations may happen. Conventional systems usually do not use the mechanisms to assess the integrity level at run-time. Instead they guarantee by design time analysis that such violations will never happen.

In contrast, our system explores the safety-cost-performance trade-off by offering multiple levels of service. Without sacrificing safety, we may run a function with a lower integrity attribute at a lower level of service. Prerequisite for this is the run-time assessment. The run-time integrity attribute output will give us, for each of the bounds specified in the example above, the confidence that the actual nominal value really is within the respective bounds. Thus, the design-time analysis is the basis to derive the run time values. The quality of detection and filter mechanisms are included in this assessment. A detector, for example, is characterized by the probability to deliver false {positives, negatives} and true {positives, negatives}. It is clear that for guaranteeing a very high confidence in the checked data, these values must be adequate. An ideal detector with no false positives and negatives will detect each failure correctly and therefore will transform the integrity attribute of a nominal value to "0" in case of a failure detected and "1" if no failure is detected. Real detectors, of course, will have weaker bounds. For a filter we similarly specify the ability to mask a failure. Details of these calculations are presented in [5]. As a consequence of the quantitative representation of engineering assumptions and the tight interplay between design time assurance and run time assessment, we can adjust the needed confidence according to the needs of the safety requirements. If such a dynamic value, represented in the integrity attribute, is not sufficient for a function running in a certain assurance level we have to switch to a lower level of service.

VI. Instantiation in AUTOSAR

AUTOSAR is a de facto standard in the automotive domain enabling an integrated architecture. It defines how to specify application software components (SWC) that are reallocatable. All software component inputs and outputs are as data elements through ports. Our suggested pattern implies that each data element should consist of a duplet: the existing nominal data element, complemented by an ASIL attribute. It is the responsibility of each SWC to compute the corresponding ASIL values. If there is no redundancy implemented, this means that the input ASIL values are inherited for the output values. Otherwise, ASIL decomposition is applied according to the algebra as defined in the ISO26262 standard. We furthermore propose a new basic software module (BSW): Safety Manager. This is in line with existing managers among the BSWs today. This new safety manager will compare the computed ASIL value of each signal with the required ones, that are stored for each function level of performance/service. In AUTOSAR today, all data elements are already connected to the run-time environment (RTE), constituting the interface for the BSW. Our pattern hence implies that the RTE will be extended with the requirement to connect all ASIL values to the Safety Manager.

The concept of modes is on three hierarchical levels in AUTOSAR: Vehicle Modes, Application Modes and BSW Modes, see Fig. 5. It's only the BSW modes that are standardized in the AUTOSAR set of specifications. It is assumed that different applications have implemented modes in their definitions. The control of these modes is within the application implementation, i.e. it is implemented by application software components (not in the platform). On an even higher level than application modes, there is Vehicles modes, which are global for the entire vehicle. As seen in the Fig. 5 there might be influences between modes on all these levels.

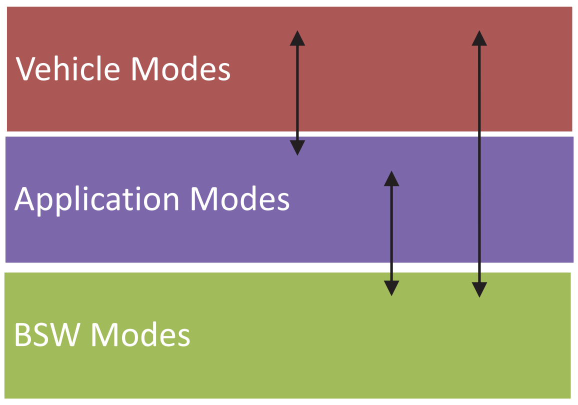


Fig. 5 How the different kinds of modes in AUTOSAR may influence each other.

We assume that the existing concept of application modes can be used for forcing all applicable SWCs to a mode corresponding to the level of service/performance as considered the highest and still safe by the Safety Manager. This is one extra connection between a SWC and the RTE, but it is in-line with the existing pattern and hence considered as an attractive extension of AUTOSAR.

As we don't consider checking spatial and temporal interference between SWCs, the normal BSW mode managers as specified by the AUTOSAR standards such as: ECU State Manager, BSW Mode Manager, Communication Manager and Watchdog Manager are not of importance here. The application mode managers are implemented as ordinary software components and communicate with other software components via RTE.

VII. Instantiation in IMA

Currently, in the avionic domain IMA is considered to be implemented by means of the ARINC 653 specification [7], which determines that applications are functionally separated in logical containers, called partitions. One goal of partitioning is to ensure the containment of faults in the domain in which they occur. Partitioning in logical containers implies non-interference of applications' execution in the time domain and the usage of separated memory and input/output addressing spaces [8].

Application software components are hosted in partitions. All software components inputs and outputs are as data elements, using the inter-partition communication services provided by the ARINC 653 Application Executive (APEX) application programming interface primitives. In particular, we propose to take advantage of the *sampling ports* services to extend the software component port values with a DAL attribute, as illustrated in Fig. 6.

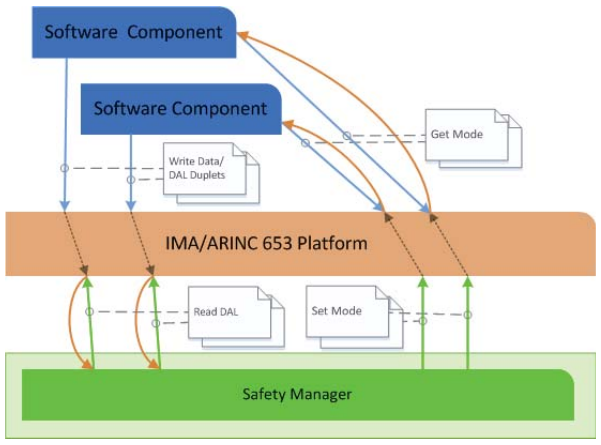


Fig. 6 Instantiation in IMA/ARINC 653 platforms

The Safety Manager will also be hosted in a partition and it will monitor the DAL attributes of all software component signals [9]. This can be easily achieved by platform configuration, allowing inter-partition communication services to deliver the components' nominal output and DAL duplets both to the destination components and to the Safety Manager, which for will read the DAL attribute, as illustrated in Fig 6.

A similar approach can be followed when the Safety Manager needs to change the performance/service level and set a new application mode. This action, which may also take advantage of inter-partition communication services, is also illustrated in Fig. 6. The Safety Manager sets the highest possible safe level of service/performance and the components change to the corresponding mode.

With this approach the instantiation of the general architecture pattern in IMA/ARINC 653 platforms does not require any modification or extension of the platform itself; only a platform configuration action is needed.

VIII. Conclusion

This paper summarizes how to apply a new architectural pattern as an extension to existing state-of-practice in the domains of avionics (IMA) and of automotive (AUTOSAR). The pattern is applicable to solve the problem when it is hard to show in design-time that a high safety integrity is met under all circumstances and for 100% of the mission. By introducing different levels of service/performance each having different implications on needed safety integrity, high performance most of the time can be combined with guaranteed functional safety all of the time.

The pattern constitutes that every signal value that is the candidate for different requirements on safety integrity levels, should be evaluated by a redundancy mechanism capable to calculate a run-time estimation of the actual provided safety integrity level.

Furthermore, the pattern implies the introduction of a Safety manager taking care of checking the safety integrity level attribute of every system signal. The safety manager compares during run-time all currently provided safety integrity levels with those derived from the break-down of the hazard analyses of the different levels of performance/service of the vehicle functions. As a result of the comparison, the safety manager enforces all functions to operate in the highest performance/service level that is still safe, by means of application mode managers.

References

[1] J. Barhorst, T. Belote, P. Binns, P. Hoffman, J. Paunicka, P. Sarathy, J.S.P. Stanfill, J.S.P. Stuart, and R. Urzi, “A research agenda for mixed-criticality systems (2009)”, http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/RBO-09-130%20Joint%20MCAR%20White%20Paper%20PA%20approved.pdf, white paper.

[2] K. Ni, N.Ramanathan, M. Chehade, L. Balzano, S. Nair, S. Zahedi, E. Kohler, G. Pottie, M. Hansen und M. Srivastava, “Sensor network data fault types” in *ACM Transactions on Sensor Networks (TOSN*), Bd. 5, Nr. 3, pp. 1-29, 2009.

[3] Zug, Sebastian; Dietrich, André; Kaiser, Jörg, “Fault-Handling in Networked Sensor Systems” (book chapter), Rigatos, Gerasimos (Hrsg.): Fault Diagnosis in Robotic and Industrial Systems, Concept Press Ltd., St. Franklin, Australia, 2012.

[4] D. H. Stamatis, “Failure Mode and Effect Analysis: FMEA from Theory to Execution”, 2 Hrsg., {ASQ} Quality Press, 2003.

[5] Brade, Tino; Zug, Sebastian; Kaiser, Jörg: “Validity-based failure algebra for distributed sensor systems” in Proc. of the IEEE 32st Symposium on Reliable Distributed Systems, Braga, Portugal, 2013.

[6] AEEC: Design guidance for Integrated Modular Avionics. ARINC Report 651-1 (November 1997)

[7] AEEC, Avionics application software standard interface, part 1 – required services, ARINC Spec. 653P1-2 (Mar. 2006).

[8] J. Rufino, J. Craveiro, P. Verissimo, Architecting robustness and timeliness in a new generation of aerospace systems, in: A. Casimiro, R. de Lemos, C. Gacek (Eds.), Architecting Dependable Systems VII, Vol. 6420 of LNCS, Springer, 2010, pp. 146–170. doi:10.1007/978-3-642-17245-8_7.

[9] P. Nóbrega da Costa, J. P. Craveiro, A. Casimiro, and J. Rufino, “Safety Kernel for Cooperative Sensor-Based Systems,” in *Safecomp 2013 Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS)*, Toulouse, France, Sep. 2013.

[10] ISO: Road vehicles — Functional safety — International Standard ISO 26262, part 1-9 (Nov. 2011).

[11] The AUTOSAR development partnership: www.autosar.org

[12] SAE: Guidelines for Development of Civil Aircraft and Systems, SAE ARP 4754A (Dec. 2010).