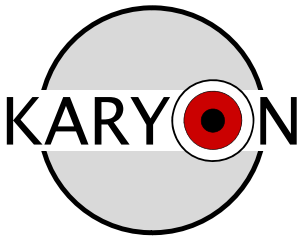

Kernel-based ARchitecture for safetY-critical cONtrol

KARYON

FP7-288195

D4.2 – First Report on Safety Kernel Definition

Work Package	WP4		
Due Date	M18	Submission Date	2013-06-25
Main Author(s)	Mário Calha (FFCUL), João Craveiro (FFCUL), Pedro Nóbrega (FFCUL), António Casimiro (FFCUL)		
Contributors	José Rufino (FFCUL)		
Version	1.0	Status	Final
Dissemination Level	Public	Nature	Report
Keywords	Safety management, Time and Space partitioning, Architecture, Interfaces		
Reviewers	Joerg Kaiser (OVGU), Kenneth Östberg (SP)		

Part of the Seventh

Framework Programme

Funded by the EC - DG INFSO

Version history

Rev	Date	Author	Comments
V0.1	2012-11-23	Mário Calha (FFCUL)	Initial Structure
V0.2	2013-03-29	Mário Calha (FFCUL)	Issues in the design and architecture of the Safety Kernel
V0.3	2013-04-06	João Craveiro (FFCUL) Pedro Nóbrega (FFCUL)	Issues in the design and architecture of the Safety Kernel (Adapting the level of service); Modules of the Safety Kernel; Scheduler support. Interfaces (figures). Minor formatting improvements.
V0.4	2013-04-26	Mário Calha (FFCUL) João Craveiro (FFCUL) Pedro Nóbrega (FFCUL)	Introduction and methodology. Scheduler support. Preliminary implementation based on TSP.
V0.5	2013-05-23	Mário Calha (FFCUL) João Craveiro (FFCUL) Pedro Nóbrega (FFCUL)	Executive summary and conclusions. Several small improvements throughout the text.
V0.6	2013-06-07	Mário Calha (FFCUL) Pedro Nóbrega (FFCUL) António Casimiro (FFCUL)	Revision based on reviewer comments.
V1.0	2013-06-22	António Casimiro (FFCUL)	Final revision and delivery.

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

This deliverable is the first report from the work task 4.2 on the Safety Kernel definition. The focus is on providing a first definition of the Safety Kernel architecture and its components.

As a basis for the design of the Safety Kernel, the relevant issues on its operation are presented and discussed, namely, the issues on collecting run time safety information, on assessing the safety requirements, and on making adjustments on the nominal system component according to the required level of service.

The Safety Kernel is a part of a KARYON system, which is composed by a few key components that realize functions that are necessary to manage the overall system behaviour and achieve functional safety objectives. Therefore, this deliverable describes the role of the Safety Kernel as a whole and of its individual components, their purpose and the functions they realise. The interactions between the Safety Kernel components and the functional components of the control system are also presented.

Finally, an example implementation approach is presented, to illustrate how the described Safety Kernel architecture can be instantiated and how its components may be realized in practice.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Purpose & Scope

KARYON focuses on the predictable and safe coordination of smart vehicles that autonomously cooperate and interact in an open and inherently uncertain environment. A fundamental idea underlying the KARYON approach is to consider a generic architectural pattern exploiting the concept of architectural hybridization. This is instrumental to allow dealing with the temporal uncertainties in the system operation, and thus deal with the main challenge to predictability and safety. In particular, part of the control system functions need not be proven timely in design time (which would be a problem due to the temporal uncertainties), provided that another part of the system always behaves timely with the required (proven in design time) probability. There are thus two distinct parts in the overall system, each with different properties with respect to synchrony, following the architectural hybridization concept. Given the uncertainties affecting the operation of part of the system and affecting the quality of the data used in control processes, the KARYON architecture defines the existence of a Safety Kernel, that is, a set of safety-related components that reside in the predictable part of the system and thus behave timely, as proven in design time. The set of components that constitute the Safety Kernel are responsible to perform monitoring and management tasks, according to rules defined in design time, which ultimately ensure the required functional safety goals. The purpose of this document is to identify the issues in the design of the Safety Kernel, specify its architecture and propose a high-level preliminary implementation.

Although the objective of KARYON is to deal with cooperative systems, ensuring functional safety of cooperative functionalities, the enforcement of some necessary behaviour (as necessary to secure safety rules) has to be decided at the vehicle level. This is because there is always the possibility that the ability of a vehicle to communicate with other vehicles or with the infrastructure is limited, preventing a centralized enforcement of the distributed behaviour to be performed. Therefore, a Safety Kernel will exist in each vehicle.

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 always the same algorithm, in the same manner, with a single performance level;

b) As a single component that may execute different algorithms or may execute an algorithm with different parameterizations, each corresponding to a different operation mode and 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 the employment of different algorithms.

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, can receive and produce information from/to other components and, finally, communication components are special in the sense that they can either receive and produce information from/to the system, but 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 thus can have a validity attribute.

Depending on the specific operation mode of each function, or on the components that are selected for realizing some function, the functionality that depends on these function will be performed with different Levels of Service (LoS). The overall goal of the Safety Kernel is precisely to manage the combination of operation modes or selected components, which is necessary to enforce the required LoS of each functionality, at each moment, in order to make sure that all 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. We note that the LoS is a measure of maximum performance level of the cooperative functionality (e.g., how close vehicles can be to each other, or how fast they can go), but does not necessarily imply a certain effective performance level (e.g., vehicles can stay far from each other even if the LoS of the functionality is high). This is because the actual control decisions depend on context data, like traffic density or weather conditions, which can be such that even if the LoS of the functionality is high, the actual performance may have to be low.

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

- The Safety Kernel will execute a set of functions on its own, which are required to support cooperative functionalities that may be executed with several Levels of Service (LoS);
- It will have to deal with timing failures of some functional components, namely complex components whose timeliness may not be easily verified 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 vary in run time;
- It will perform the management of operation modes and component configuration based on the observed timeliness of complex components and on the observed data validity, and considering pre defined safety rules associated to each LoS of each functionality.

This is the first report on this topic and its purpose is mainly to define the architecture of the Safety Kernel.

1.2 Relation to other work

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 about safety requirements and constraints. Namely, the specification of safety rules is an expected outcome of task 4.1.

The generic design specification for the Safety Kernel defined in this document will later be instantiated in the demonstrators in WP5.

Besides the tight cooperation with other work tasks in KARYON, the state of the art in the research field, and the state of practice in the industrial community, has been used as an input.

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 [14]. 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 [15] 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 [13] 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 [16] 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 [17] 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 [18] 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 sensor, 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 [19] 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 [20].

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

Chapter 2 covers the issues in the design of the Safety Kernel, from tracking and assessing the safety requirements to the adaptation of the LoS. The chapter ends with a discussion regarding the impact of the adjustment of the LoS of a cooperative functionality upon functional components that may be shared with other functionalities.

Chapter 3 presents the preliminary architecture of the Safety Kernel by detailing its components and their interfaces. The chapter 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. The chapter ends with an example scenario that illustrates the core operation of the Safety Kernel.

Chapter 4 introduces a possible high-level implementation of the Safety Kernel based on a Time and Space Partitioning system. This implementation is based on the AIR.

Finally, in chapter 5 the report is briefly summarized and the next steps are identified.

2. 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 chapter 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.

2.1 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.

2.1.1 Knowing 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 have to 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. The rules specification is an expected outcome of task 4.1.

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.

2.1.2 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 these 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.

2.2 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, which is an issue to be addressed in a future step within WP4.

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.

2.3 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. This adaptation is an outcome of the assessment of the safety requirements, as explained in section 2.2.

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 timing bounds.

2.4 A generic example of LoS management

Consider two cooperative functionalities, CF_A and CF_B, which are being executed among a set of cooperative vehicles. These functionalities are performed by combining the use of a number of functions available in each vehicle. In this illustrative example we consider that in order to provide these two functionalities it is necessary to use the following components in each vehicle: four sensors, S1 to S4, six functions, F1 to F6, and two actuators, A1 and A2, interconnected as depicted in Figure 2. For example, it can be seen that function F1 takes, as inputs, data from the four sensors and produces a result that will be consumed by both F2 and F3.

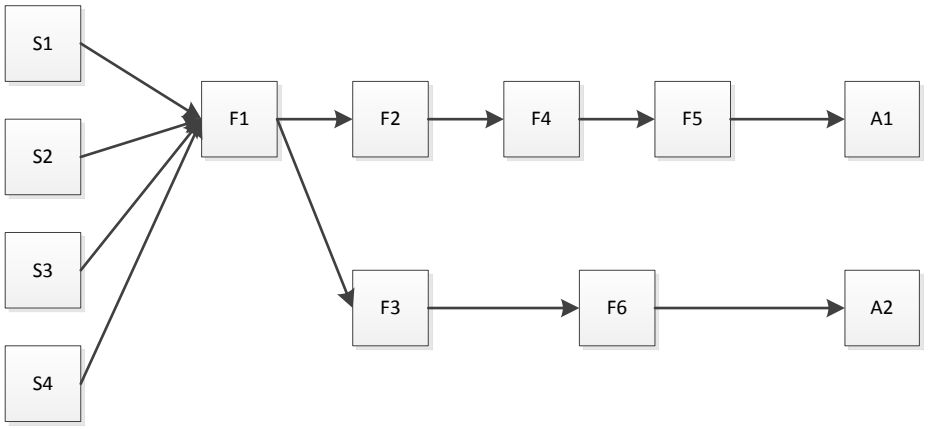

Figure 2: Example of component interconnection.

Each cooperative functionality uses the following functions:

- CF_A − F1, F2, F4 and F5;
- CF_B − F1, F3 and F6.

Given that the output of F5 is sent to A1, this means that functionality CF_A is realized by actuations on actuator A1. Similarly, CF_B is realized by actuating on A2.

The system functions have the following implementations:

- F1 is implemented as a component, C1, with three performance levels;
- F4 has two implementations, C4' and C4'', as shown in Figure 3;
- The remaining functions are implemented as a single component with a single performance level.

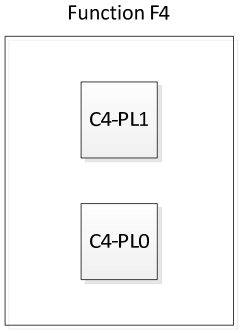

Figure 3: Example function with two implementations.

This means that in the case of F1, a change in its performance level only requires a reconfiguration, while for F4 it would require selecting the output of a different component, either C4-PL1 (performance level 1) or C4-PL0 (performance level 0).

From the point of view of the LoS for each cooperative functionality, both CF_A and CF_B have four LoS (from LoS0 to LoS3). The LoS of CF_A depends upon two monitored variables: 1) the validity of the output of sensor S1, designated as V1, and 2) the execution time of C4, designated as ET_{C4}. The LoS of CF_B depends upon two monitored variables: 1) the validity of the output of sensor S1 (already designated as V1) and 2) the validity of the output of sensor S2, designated as V2. So, in total, three variables are monitored at each vehicle where the two cooperative functionalities are defined. The values of V1 and V2 are given as floating-point

numbers in the interval [0.0, 1.0][1], while the value of ET_{C4} is checked against a given relative deadline D_{C4}.

More specifically, for each LoS, CF_A requires the following set of safety rules to be met:

- $CF_A(LoS3) \rightarrow V1 > 0.8 \ \wedge \ ET_{C4\text{-}PL1} < D_{C4}$
- $CF_A(LoS2) \rightarrow V1 > 0.6 \ \wedge \ ET_{C4\text{-}PL1} < D_{C4}$
- $CF_A(LoS1) \rightarrow V1 > 0.6$
- $CF_A(LoS0)$, *otherwise*

On the other hand, for each LoS, CF_B requires the following set of safety rules to be met:

- $CF_B(LoS3) \rightarrow V1 > 0.8 \ \wedge \ V2 > 0.7$
- $CF_B(LoS2) \rightarrow V1 > 0.8$
- $CF_B(LoS1) \rightarrow V1 > 0.6$
- $CF_B(LoS0)$, *otherwise*

The safety of CF_A and CF_B is guaranteed by the observation of these safety rules.

When a cooperative functionality has its LoS changed, this implies a change in the performance level of one or more components, depending on which safety rules are met.

In the case of this example, a table that defines the performance levels of the components in dependence of the LoS of all functionalities could be prepared. Table 1 presents the initial rows of the combinations between the LoS of both functionalities and the consequences in the performance level for each component. An invalid combination is represented as an action not being applicable, N.A., to any reconfigurable functional components.

CF_A	CF_B	F1	F2	F3	F4	F5	F6
LoS3	LoS3	PL2	-	-	PL1	-	-
LoS2	LoS3	N.A.	-	-	N.A.	-	-
LoS1	LoS3	PL2	-	-	PL0	-	-
LoS0	LoS3	N.A.	-	-	N.A.	-	-
LoS3	LoS2	PL1	-	-	PL1	-	-
LoS2	LoS2	N.A.	-	-	N.A.	-	-
LoS1	LoS2	N.A.	-	-	N.A.	-	-
…	…	…	…	…	…	…	…

Table 1: Effect of LoS combinations on the PL of components.

If a third functionality was added to the system, this would imply that the size of Table 2 would increase, in order to consider all possible combinations of functionalities and their LoS. In practice, several optimizations could be done in order to summarize the information in the table.

[1] We use floating point numbers just as an example. Preliminary approaches for modeling the validity of sensor data have been addressed in deliverable D2.2.

For instance, Table 2 could be simplified by removing the rows with impossible combinations, such as CF_A(LoS2) with CF_B(LoS3). The simplified table would become as follows:

CF_A	CF_B	C1	C2	C3	C4	C5	C6
LoS3	LoS3	PL2	-	-	PL1	-	-
LoS1	LoS3	PL2	-	-	PL0	-	-
LoS3	LoS2	PL1	-	-	PL1	-	-
…	…	…	…	…	…	…	…

Table 2: Simplified LoS combinations on the PL of components.

From this table it can be seen that, due to a LoS change, any functional component can have its performance level adjusted.

For example, a possible LoS change scenario happens when the best implementation of F4 misses its deadline (that is, when $ET_{C4\text{-}PL1} > D_{C4}$). In this case the LoS of CF_A could go from LoS3 to LoS1 (which can be seen by observing the stated safety rules). Since CF_B is in LoS3, the Safety Kernel would have to force C4 into PL0, while keeping C1 in PL2.

An interesting scenario happens when V2 becomes smaller or equal to 0.7 while V1 > 0.8 and $ET_{C4\text{-}PL1} < D_{C4}$. This leads to a reduction in the LoS of CF_B, which implies a reduction in the performance level of component C1. As this component is shared with CF_A, this cooperative functionality is affected by a reduction of the performance of C1. Despite all safety rules for CF_A to be provided in LoS3 are met, one of its functional components will execute in a lower performance level. This is not a problem for safety, just for performance of CF_A. However, if during design time this is considered to be unacceptable, then it is possible to define a new component similar to C1 to be used exclusively by CF_A, which would not be affected by a degradation of V2.

3. Architecture of the Safety Kernel

This chapter details a preliminary design specification for the Safety Kernel to manage the LoS of cooperative functionalities. The Safety Kernel includes the necessary components to perform the tasks identified in Chapter 2.

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.

3.1 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.

3.2 Components of the Safety Kernel

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 3.3.1.

All these components and their interactions are represented in Figure 4. 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.

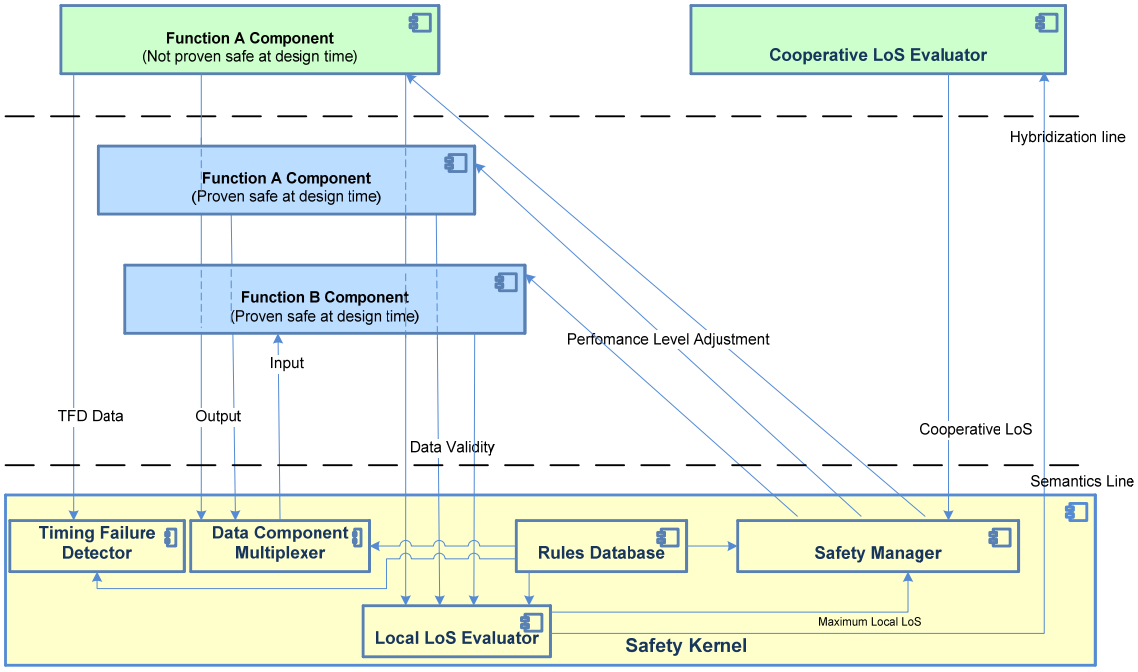

Figure 4: System components overview and interaction.

3.2.1 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. The rules specification is an expected outcome of task 4.1.

3.2.2 Data Component Multiplexer

As explained in Section 3.1, 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.

3.2.3 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.

3.2.4 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 3.2.5) and to the Cooperative LoS Evaluator (discussed on section 3.3.1).

3.2.5 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.

3.3 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.

3.3.1 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.

3.3.2 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 3.4) 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 3.5.

3.4 Interfaces

In order to build the components described in Section 3.2, 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 5.

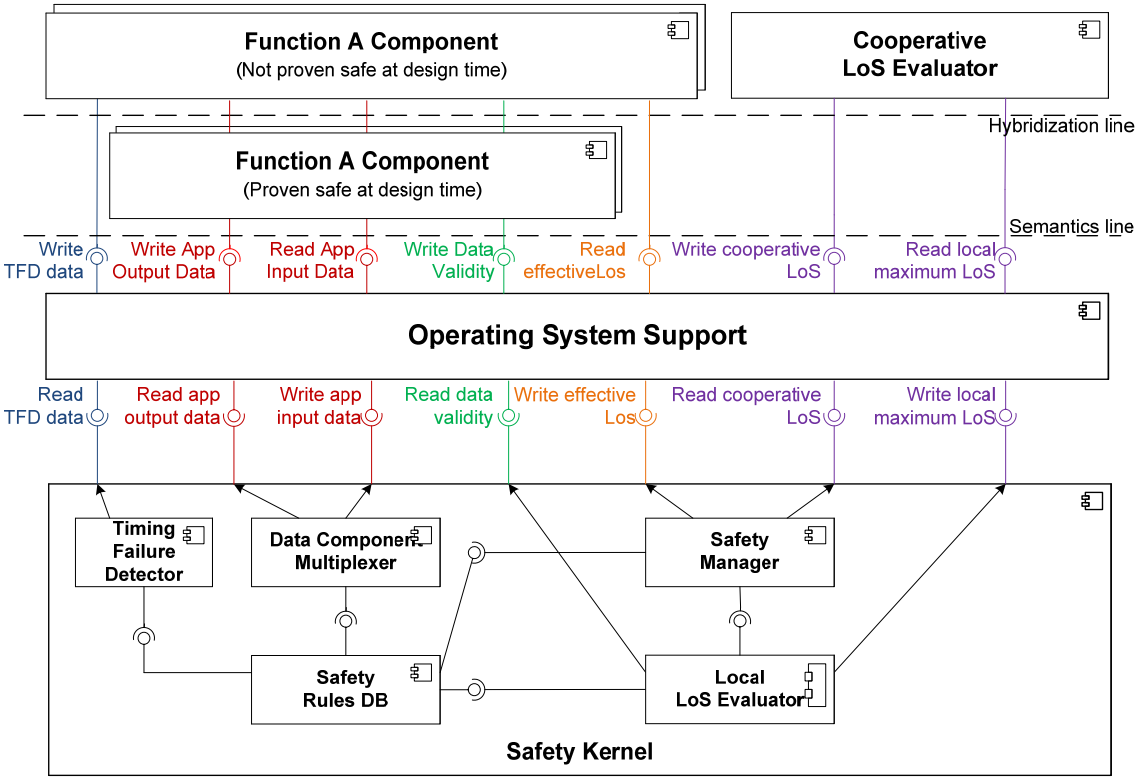

Figure 5: Interfaces between the Safety Kernel and external related components.

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 3: Interfaces of the Safety Kernel

3.4.1 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 6. 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 3.3.2.

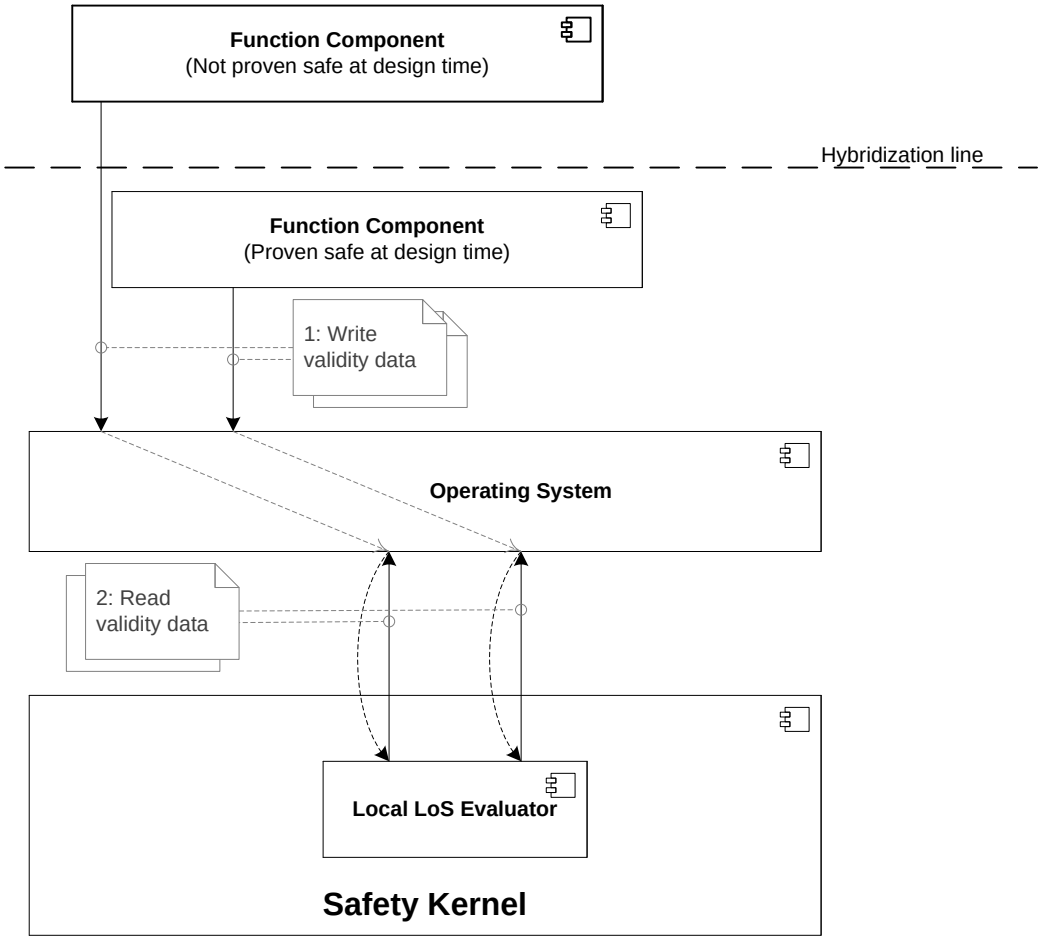

Figure 6: Interaction with the Data Validity Interface.

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.

3.4.2 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.

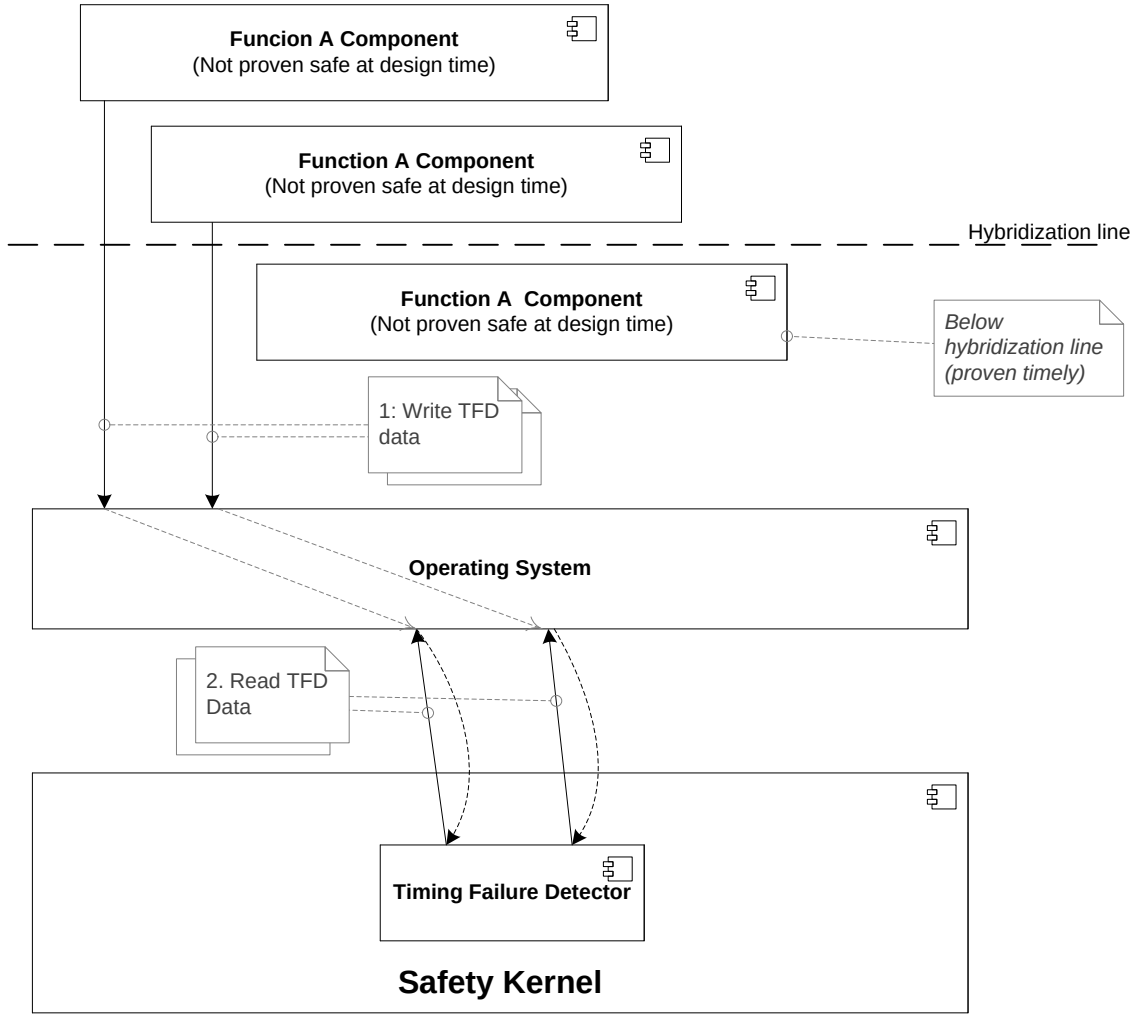

Figure 7: Interaction with the Timing Failure Detector Interface.

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.

3.4.3 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).

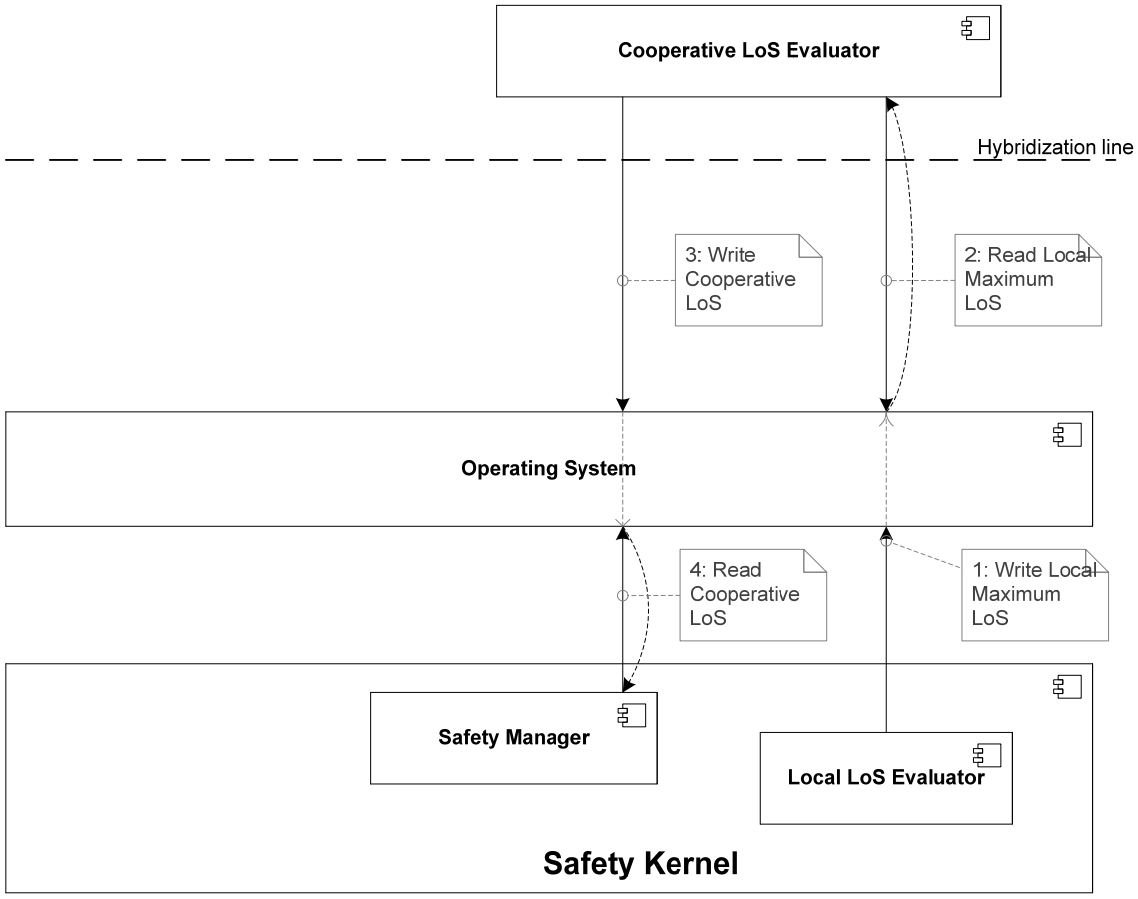

Figure 8: Interaction with the Cooperative LoS Interface.

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.

3.4.4 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.

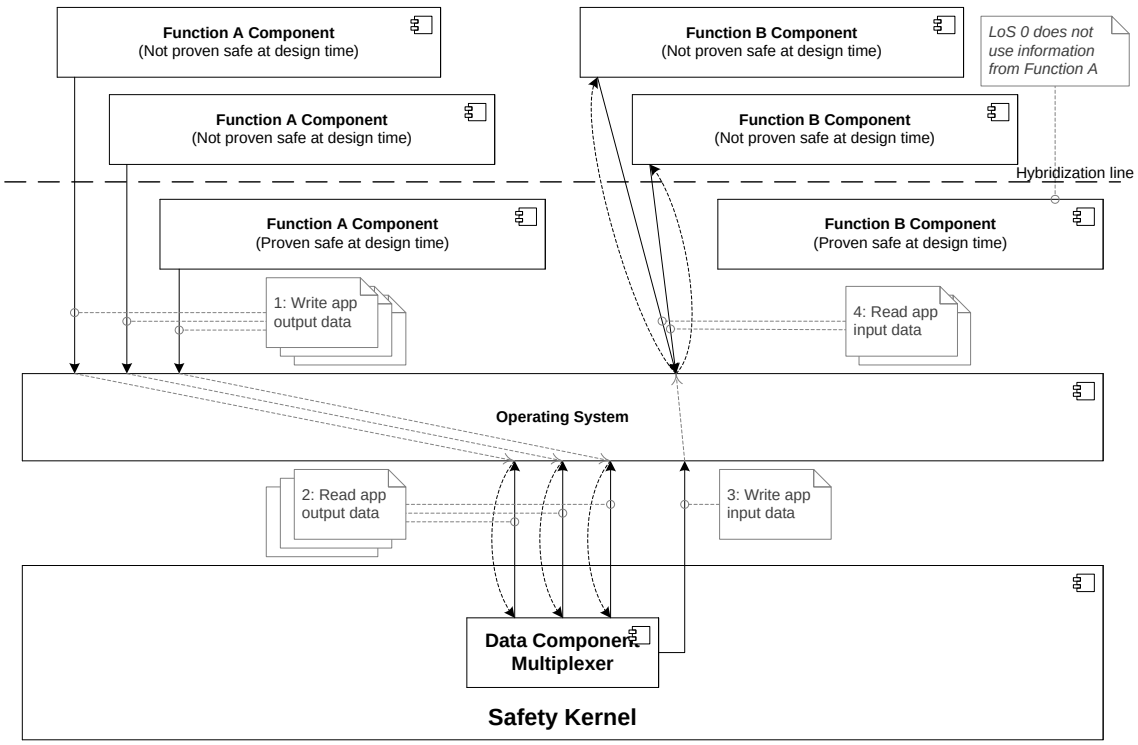

Figure 9: Interaction with the Data Component Multiplexing Interface.

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.

3.4.5 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.

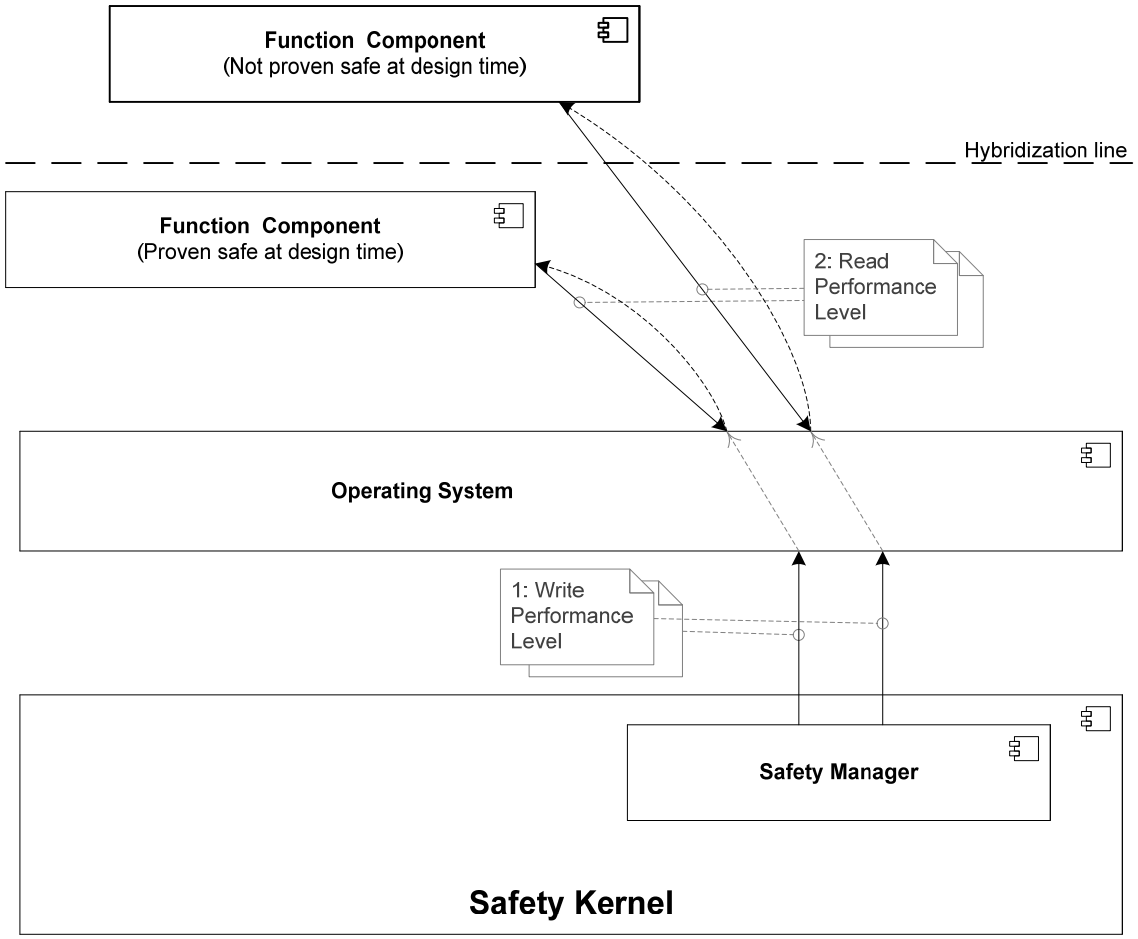

Figure 10: Interaction with the Mode Switch Interface.

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).

3.5 Scheduler support

In Section 3.3.2, we have 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 of 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.

4. Implementation

Based on the requirements described previously, this chapter details a preliminary implementation of a KARYON system based on the Time and Space Partitioning (TSP) concept [1, 2], and how this concept provides the mechanisms to ensure safety despite the presence of components with uncertain behaviour.

Finally, a practical example of implementation using a TSP architecture is shown, detailing how it will support scheduling and communication between the system components.

4.1 Time and Space Partitioning

As already explained, KARYON must ensure fault containment between different components, in order to guarantee that a fault in one component does not compromise another component's or system's safety.

Time and space-partitioned systems (TSP) is a concept for safety-critical systems in which applications with different criticalities and requirements may coexist in the same system and using the same hardware resources.

A prominent example of TSP system design is the adoption of the ARINC specifications 651 (Design Guidance for Integrated Modular Avionics [3]) and 653 (Avionics Application Software Interface [4]) in the civil aviation domain. The traditional approach, federated avionics, whereby each avionics function had its own dedicated (and sometimes physically apart) computer resources, suffered from potential inefficient resource utilization due to the inability to reallocate resources at runtime [9, 10]. Replacing federated avionics with ARINC 651 Integrated Modular Avionics (IMA) [3] allowed addressing the needs of modern systems, such as optimizing the allocation of computing resources, reducing size, weight and power consumption (a set of common needs in the area of avionics, which is commonly represented by the acronym SWaP), and consolidation development efforts (releasing the developer from focusing on the target platform, in favor of focusing on the software and easier development and certification processes) [11]. The ARINC 653 specification [4] is a fundamental block from the IMA definition.

The identification of similar requirements with the aviation industry led to the interest expressed from space industry partners in applying the time and space partitioning concepts of IMA and ARINC 653 to space missions onboard software [5, 6, 7]. In the European space industry domain, the TSP Working Group (incorporating space agencies and industrial partners) was established to cope with the issues of adopting TSP in space. The TSP Working Group analysed benefits and the remaining technology gap to the intended adoption, and found no technological feasibility impairments to the latter [5]. In 2003, the European Space Agency proposed [12] ensuring compatibility with ARINC 653/IMA as a future standardization action, so that the exchange of functional building blocks with the aeronautic industry (which had already adopted IMA) would be made possible. To manage the problem of how applications interface with the underlying operating system, ARINC 653 should be taken into account as an example, in order to define such an interface in a way that allows OS-independent software components. The European Space Agency is currently still active in funding and partaking studies for a software reference architecture based on time and space partitioning [6, 7].

The automotive industry is also active in the adoption of TSP system design. The top-level requirements for an AUTOSAR operating system include provisions that correspond, to some extent, to the notion of TSP – namely, requirements SRS_Os_11005 and SRS_Os_11008 [8].

TSP separates the system's software components into logical containers called partitions, ensuring fault containment between them (i.e. faults that occur in one partition do not affect other partitions), with respect to both time and space.

Time partitioning is the property that ensures that even in the presence of temporal faults in one partition, the timing guarantees of remaining partitions are not affected. Space partitioning ensures that one partition cannot access to zones of memory belonging to another. These two properties ensure that faults are contained to their domain of occurrence, preventing them from propagate to other partitions.

Partitions are managed by an underlying layer in charge of enforcing TSP properties, responsible for partition scheduling and dispatching, memory protection, and communication between partitions.

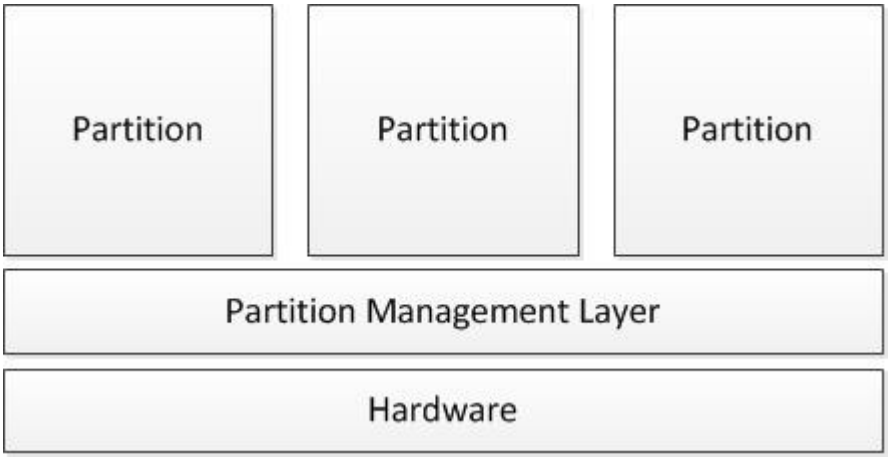

Figure 11: A TSP architecture.

Due to the nature and uncertainty present in a KARYON system, there is a clear mapping between the guarantees provided by TSP and KARYON requirements. By isolating non-safe components (i.e., above the hybridization line) in different partitions from other components, we can ensure that failures and delays that possibly occur in these components do not affect any other ones.

The next section will demonstrate an implementation of a KARYON system using a TSP architecture called AIR.

4.2 AIR – a TSP implementation

AIR [1] is a TSP architecture implementation, which uses an intermediary layer called Partition Management Kernel (PMK) to ensure TSP properties. Although inspired in the ARINC 653 specification [3], the design of the AIR architecture aims to improve upon such specification. More specifically, it deliberately diverts from ARINC 653 where the latter's limitations can be overcome to the benefit of additional functionality and flexibility without compromising safety.

As explained above, TSP provides fault containment between different partitions. KARYON must ensure fault containment between components above and under the hybridization line, so that the first group do not compromise the safety of the latter and of the overall system. As such, a KARYON implementation using TSP should isolate non-proven components in individual partitions to prevent errors from propagating to other components. Hence, components that compose the system should be divided in partitions according not only by the function they belong to but as well by the time guarantees they provide. One example of such division is pictured in Figure 12.

Figure 12: Partitions in a TSP system.

One partition may, however, host more than one component. The division in different partitions is only mandatory between components on which we want to ensure fault containment by enforcing temporal and spatial segregation.

Partitions are scheduled by the PMK according to a fixed schedule defined at design time, bounding the time assigned to each partition and ensuring that timing faults do not propagate from one partition to another. This schedule repeats itself over the time, over a time period called Major Time Frame (MTF).

The fact that a fixed cyclic schedule is used ensures that scheduling is completely predictable, and that the time assigned to each partition in known and bound. Each partition may then use a local policy to schedule its own processes within the time slice allocated to it. This two-level scheduling approach is the base mechanism used to ensure time partitioning guarantees. One example of this mechanism is picture in the Figure 13. PMK implements the first scheduling level, while the second level is managed by the partition itself.

This segregation in the time domain allows the use of different operating systems in different partitions, including non-RTOS (in which case, all the components hosted in the partition are considered to be above the hybridization line).

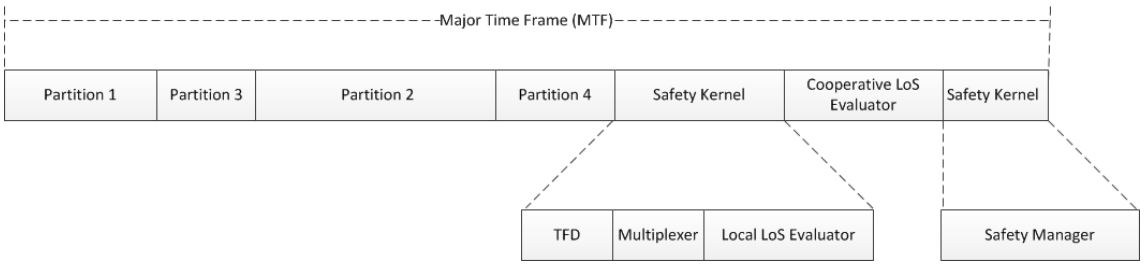

Figure 13: Schedule for a Major Time Frame.

AIR implements two communication mechanisms between different partitions: queues and sampling ports. Queues store all the values sent by one partition in a buffer until they are read. On the other hand, sampling ports only store one value, overwriting it every time a new value is sent, avoiding possible overflow problems.

The interfaces described in Section 0 required to provide communication between components and the Safety Kernel are implemented using sampling ports.

When a sampling port is created, a validity period must be defined. This period defines the rate at which a value should be refreshed by the sender. Upon reading from a sampling port, the reader knows whether the value is still valid or not. The value of this validity period must be calculated prior to the creation of the port, and it should reflect how long is the value valid to be used by other components after being written and the period of execution of the component that writes it, based on the scheduling performed by the PMK. This mechanism may be used by readers to detect delays and timing faults in components above the hybridization line. If a component is delayed and does not write some value that was supposed and the validity of the previous value expires, the delay will be, eventually, detected by the reading components.

The PMK layer is in charge of transferring the data between the sender and the receiver ports.

To implement all the required interfaces, sampling ports establishing communication channels between the components and the Safety Kernel must be defined. Depending on the component requirements, some ports may not be needed (e.g., one component under the hybridization line does not need TFD interface).

One example of the system partitions and the ports used in each partition to implement the required interfaces is picture in the Figure 14.

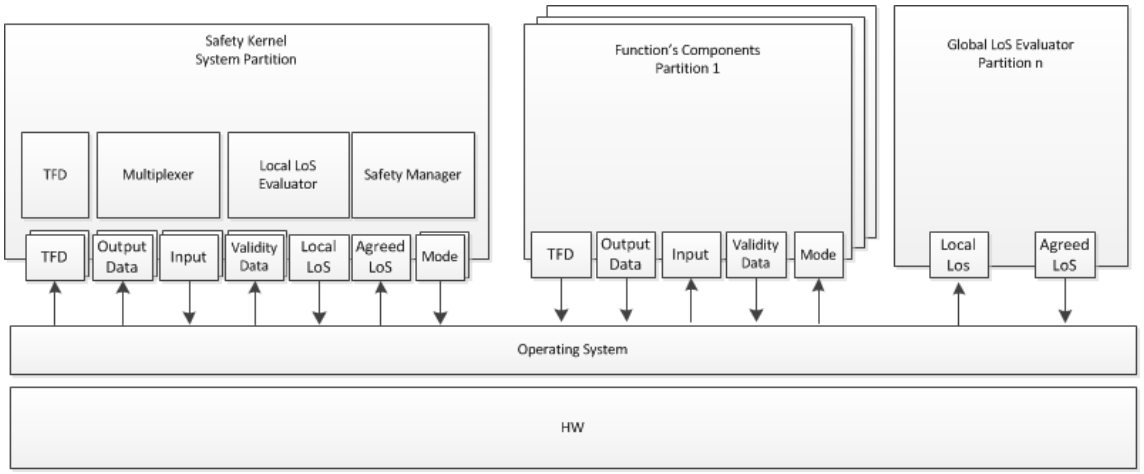

Figure 14: Communication through sampling ports.

This picture exemplifies the ports used to implement communication between the Safety Kernel and other components of the system using the interfaces described in 3.4.

As explained, the Safety Kernel and the Global LoS Evaluator are hosted in dedicated partitions, as well as each component of each function. As such, communication between these parties is done exclusively using sampling ports. In this document, the focus is on the communication between components and the Safety Kernel. However, if needed, components of one function in one partition may communicate with components in a different partition using the same mechanisms (whether sampling ports or queues) provided by the operating system.

The number of ports needed in each function component may vary. For instance, TFD ports are only created for components above the hybridization line. One component may also have more than one input or output, or may need to communicate more than one validity or TFD values, and may, therefore, contain more than one port for each of these purposes.

Each of the ports in each component links to a port in the Safety Kernel. Hence, Safety Kernel will have multiple TFD ports (i.e., one for each port in each component above the hybridization line), as well as multiple Output, Input and Mode ports.

Communication between the Safety Kernel and the Cooperative LoS Evaluator is done using two single ports; Local LoS, linking to the Local LoS Evaluator, and Agreed Mode, linking to the Safety Manager.

5. Conclusions and next steps

It is possible to define a Safety Kernel to manage the LoS of cooperative functions in order to secure functional safety requirement. The architecture of the Safety Kernel here presented fulfils three goals:

- Provides a set of components that are required to support cooperative functions that may be executed with several Levels of Service (LoS);
- Supports functional components whose timeliness does not need to be proven to hold in design time, in addition to components proven timely in design time; and
- Performs LoS management based on data validity information and on safety rules associating LoS with validity bounds, rather than detecting specific faults and failures in accordance to specific fault and failure models.

Additionally, a possible implementation of the Safety Kernel was presented. This implementation is based on a Time and Space Partitioning (TSP) system and it fulfills the goals of the Safety Kernel.

A key challenge now lies in the definition of the Rules Database in dependence of the expected output of task 4.1. If the rules only involve independent criteria then the time required for their evaluation can be determined with great precision. Otherwise, with interdependent criteria, the time for their evaluation can vary significantly and therefore an upper bound will have to be derived. These decisions will affect the Local LoS Evaluator and the Safety Manager.

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] AEEC. Design guidance for integrated modular avionics. ARINC Specification 651, Nov. 1991.

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

[5] J. Windsor and K. Hjortnaes, "Time and space partitioning in spacecraft avionics," in *3rd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009)*, July 2009.

[6] J. Windsor, M.-H. Deredempt, and R. De-Ferluc, "Integrated modular avionics for spacecraft — User requirements, architecture and role definition," in *30th IEEE/AIAA Digital Avionics Systems Conference (DASC 2011)*, Oct. 2011.

[7] J. Windsor, K. Eckstein, P. Mendham, and T. Pareaud, "Time and space partitioning security components for spacecraft flight software," in *30th IEEE/AIAA Digital Avionics Systems Conference (DASC 2011)*, Oct. 2011.

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

[9] N. Audsley and A. Wellings, "Analysing APEX applications," in *17th IEEE Real Time Systems Symposium (RTSS '96)*, Dec. 1996.

[10] M. A. Sánchez-Puebla and J. Carretero, "A new approach for distributed computing in avionics systems," in *1st International Symposium on Information and Communication Technologies (ISICT 2003)*, Dublin, Ireland, 2003.

[11] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics," in *26th IEEE/AIAA Digital Avionics Systems Conference (DASC 2007),* Dallas, TX, USA, 2007.

[12] P. Plancke and P. David, Technical note on on-board computer and data systems. European Space Technology Harmonisation, Technical Dossier on Mapping, Technical Note TOS-ES/651.03/PP, ESA. Xii, 2003.

[13] 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

[14] 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.

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

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

[17] 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.

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

[19] 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.

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

