Kernel-based ARchitecture for safetY-critical cONtrol

KARYON

FP7-288195

D2.3.1 - KARYON architecture (Public version)

Work Package	WP2		
Due Date	M18	Submission Date	2013-07-03
Main Author(s)	António Casimiro (FFCUL)		
Contributors	Rolf Johansson (SP), Kenneth Östberg (SP), Renato Librino (4SG), Siavash Aslani (4SG), Jörg Kaiser (OVGU), Elad M. Schiller (CTHA)		
Version	V1.0	Status	Final
Dissemination Level	Public	Nature	Report
Keywords	Functional architecture, system architecture, system model		

Part of the Seventh

Framework Programme

Funded by the EC – DG INFSO

Version history

Rev	Date	Author	Comments
V0.1	2013-02-4	A. Casimiro (FFCUL)	Table of Contents
V0.2	2013-06-30	A. Casimiro (FFCUL)	Draft version
V0.3	2013-07-02	A. Casimiro (FFCUL)	Final version
V1.0	2013-07-03	A. Casimiro (FFCUL)	Final review and delivery

Glossary of Acronyms

4DT	4 Dimensions Trajectory
ABS	Anti-lock Braking System
ADAS	Advanced Driver Assist Systems
ADS-B	Automatic Dependent Surveillance-Broadcast
ASIL	Automotive Safety Integrity Level
ATM	Air Traffic Management
CAM	Co-operative Awareness Messages
COTS	Commercial Off-The-Shelf
DAL	Design Assurance Level
DB	Database
DoW	Description of Work
Dx.y	Deliverable belonging to work package x, with serial number y
ETSI	European Telecommunications Standards Institute
HMI	Human Machine Interface
I2V	Infrastructure to Vehicle
ISO	International Organization for Standardization
ITS	Intelligent Transport Systems
KARYON	Kernel-based ARchitecture for safetY-critical cONtrol
LDM	Local Dynamic Map
LIDAR	Light Detection And Ranging
LoS	Level of Service
N.A.	Not Applicable
OWL	Web Ontology Language
PICS	Protocol Implementation Conformance Statement
RADAR	Radio Detection And Ranging
RDF	Resource Description Framework
RDF(-S)	Resource Description Framework Schema
RPV	Remote Piloted Vehicle
RSU	Road Side Unit
Rx	Receive
SIL	Safety Integrity Level
SK	Safety Kernel
SLAM	Simultaneous Localization And Mapping
TCB	Timely computing Base

TPM	Trusted Platform Module
TTCB	Trusted Timely Computing Base
TVRA	Threat, Vulnerability and Risk Analysis
Tx	Transmit
Tx.y	Task belonging to work package x, with serial number y
V2I	Vehicle to Infrastructure
V2V	Vehicle to Vehicle
V2X	Vehicle to Vehicle or to Infrastructure
WP	Work Package
WPx	Work Package with serial number x

Executive Summary

The main objectives of WP2 are to the definition the KARYON safety architecture, providing the guiding principles on how to structure a safe system in relation to assumed system and fault models. This will be done while taking into account that systems can be built from heterogeneous application components, where some components will realize their functions in a very predictable way, as necessary to meet the safety integrity levels that are assigned to them, but other components may also be included to achieve improved functionalities, although at the cost of being more complex, less dependable and possibly untimely. The goal is to define a hybrid system architecture that integrates all these components in a way that makes it possible to secure functional safety requirements while achieving, whenever possible, improvements in the way the functionality is provided, in comparison to solutions where all system components and interactions have to be predictable in order to prove that the functionality is safe at design time. This deliverable builds upon the first report on the KARYON architecture, improving it and extending it with new material to describe in a complete way the KARYON generic architectural pattern.

The specific contributions that are provided in this deliverable are threefold.

Firstly, the deliverable discusses some notions that are important to set the stage for understanding the problems under consideration. These include, for instance, a discussion on the nature of cooperative systems and what this implies for the development of safe cooperative functionality, a discussion of functional safety concepts and their importance in KARYON, and a discussion of the underlying system model solution that is adopted in KARYON.

Secondly, the deliverable presents the generic KARYON architectural pattern for developing safety critical cooperative and autonomous systems. The architectural pattern is at a sufficiently high level of abstraction to allow varied instantiations of the architecture, depending on the concrete system that is to be developed. On the other hand, it is sufficiently concrete to be usable in guiding system designers in the right direction, by explaining how to structure the system and by defining the fundamental architectural components that must be present in any concrete system. At this level of abstraction, the details on components such as sensors or the Safety Kernel are not important and are not provided. Instead, what is important is to clearly explain the role of all these components or architectural blocks, their semantics, the connections that exist between them, the properties that are expected from the underlying infrastructure where these components are implemented and the properties they exhibit. This is what the reader will find in this deliverable.

Last, but not the least, the deliverable provides examples on how the generic architectural pattern would be instantiated to develop system solutions in the avionics and automotive domains. The examples are necessarily constrained because KARYON is not focused on the development of concrete functions, but are intended to be sufficiently rich to illustrate the application of the KARYON architectural solution.

Table of Contents

1. Introduction

1.1 Purpose & Scope

The main objective of WP2, as stated in the KARYON Description of Work, consists in "*the definition of the KARYON safety architecture, providing the guiding principles on how to structure a safe system in relation to assumed system and fault models*".

KARYON focuses on the predictable and safe coordination of smart vehicles that autonomously cooperate and interact in an open and inherently uncertain environment. Although it is possible to exploit the cooperative functionality for the benefit of each vehicle's behaviour, with implicit gains to vehicles as a whole and to traffic, it becomes necessary to deal with the possible negative impact of the uncertainties affecting communication and ultimately creating safety problems.

The purpose of this deliverable is to describe the KARYON architecture, which must be understood as a general pattern to be applied in the development of concrete systems and cooperative functionalities. The cooperative nature of the considered functionalities and the need to deal with functional safety requirements are thus concerns that have been considered in the definition of this architectural pattern. One of the fundamental challenges that we address in the definition of the KARYON architecture is, in fact, the need to accommodate uncertainties in the temporal and value domains, while also providing the conditions for safety requirements to be satisfied and functional safety argumentations to be developed in design time. Although the focus of the deliverable is on the description of the KARYON architecture, the deliverable aims at providing the necessary background and concepts that are important for a good understanding of the architectural options. To complement the description, the deliverable also focuses on some illustrative examples, in which the architectural pattern is instantiated considering concrete use cases.

1.2 Approach

There is a whole body of knowledge on how to achieve safe systems, but in general the existing solutions and approaches are restrictive regarding the considered operational environments, excluding the sources of uncertainty or unpredictability right from the start and thus limiting the contexts in which the resulting systems can be used. Uncertainty can also be dealt with by making pessimistic worst case assumptions on bounds for the relevant variables. The consequence, in this case, is that system resources are over-dimensioned and hence the resulting systems are less efficient than what they could, in principle, be.

In KARYON we consider hybrid distributed system models [22] and we explore the concept of architectural hybridization as a baseline design principle [24]. This allows us to define a generic architecture that accommodates both complex functions that might be subject to temporal uncertainties, and simple functions, with well-defined behaviour, which are fundamental to deal with safety concerns. The architecture includes these two different realms of operation, as needed to explore the potential benefits of complex components being used as part of the cooperative control system, while ensuring that safety is preserved by means of a well-defined set of components, in which a Safety Kernel is included. In essence, hybrid distributed system models assume that different parts of the system are characterized by different properties (for instance, each part having different timeliness properties or different integrity levels with respect to some assumed failure modes), and architectural hybridization explicitly separates different functions or components of the system into these different parts, to ensure that the

assumed properties are indeed satisfied in the real system. Using a system model that, on one hand, allows heterogeneous properties to be achieved for different parts of the system (as needed when considering operational environments where uncertainty is intrinsic but where some predictability is required) and, on the other hand, provides the adequate formal framework for the design of protocols and system solutions, is fundamental to meet KARYON objectives.

The idea of structuring the system in terms of complex control functions and simple control functions has also been proposed for achieving control systems with improved performance and without compromising control stability, and was called the Simplex approach [19]. In this approach, the idea is to have two alternative control functions that might be used for controlling some system, where one is designed to achieve improved control at the expense of an increased complexity of the control algorithm, while the other uses a simple control algorithm that is not designed for ultimate performance. The trade-off for the improved performance of the complex control algorithm is that it might not always behave correctly because it will be more exposed to errors, and therefore this may bring the controlled system to an unwanted state. The simple algorithm provides the necessary redundancy to compensate for problems in the execution of the complex algorithm. It will necessary to know the control set points and define allowed deviations to determine when it becomes necessary to switch over from the complex algorithm to the simple one. The key issue in this Simplex approach is to use simplicity to control complexity, which is better than using other fault-tolerance approaches like N-version programming [2] and recovery blocks [18], when considering that resources are limited.

Although the basic concept we follow in KARYON for the definition of the generic architecture is the same that is underlying the Simplex [19] and the Wormholes [24] approaches, we exploit it in a different manner, which is better suited for the autonomous and cooperative systems under consideration.

For instance, differently from the Wormholes approach, we consider that the simple part of the system includes functions that pertain to the application (like in Simplex), not just generic services that may be used, whenever necessary, by the applications executing in the complex part of the system. But we still define our Safety Kernel, which provides generic services, to be implemented in the simple and predictable part of the system, thus exploiting the same benefits from such design that exploited in the Wormholes model.

From the Simplex approach we inherit the idea of implementing control functions redundantly, as explained above. It will thus be possible that in a KARYON system some functions: a) might be complex, as necessary to achieve the desired performance improvements, b) might not always perform predictably, which can happen when considering open environments with uncertain sensory information being collected, c) but will have simpler redundant counterparts that will execute in a predictable way and will ultimately fulfil the needed safety requirements. But differently from Simplex, in KARYON we do not assume that sensors are reliable. Therefore, we cannot use sensor information to determine how well the system is being controlled in order to decide when to switch to a safe control algorithm. Moreover, in KARYON we consider different settings that go beyond a well defined control problem with well defined control objectives that might be predefined and checked in run time. In KARYON the context is changing and the control objectives are changing as well. They are defined by the engineers developing the cooperative functions and are typically dependent on the context. We thus define a fundamentally new way of dealing with the management of the system configuration, and we introduce the notion of Level of Service (LoS) for that. In addition, given that in KARYON we are concerned with faults affecting sensor data, we introduce the notion of a continuous characterization of the quality of sensor data, which we translate into a validity attribute that is used to manage the system configuration. Finally, and in order to achieve a separation of concerns between the development of the nominal control system and the development of the mechanisms to manage the system configuration, we say that the management is performed by a Safety Kernel, and the definition of this Safety Kernel is an issue on its own.

We have to say that there exist many example of systems in which the idea of architectural hybridization is present, even if not in an explicit way. This is particularly true in systems and applications for which a fail stop safe state can be defined. For instance, in automated guided trains or subways a typical safe state is to have the train stopped. For that, some components are developed with the single purpose of monitoring some safety constraints, being able to actuate on train brakes if necessary. These components are very simple (and hence more reliable) and are typically separated from the main (payload) systems, so that they constitute independent failure domains. Whenever a safety rule is violated, there is an immediate switch to the safe state. In KARYON we address a more complex problem, in which the safe state may not be a system stop, and may be a fail operational state. Additionally, we aim at allowing the function to be performed in possibly several modes, as a result of different combinations of faults affecting sensor data and complex components. Moreover, given that the quality of sensor data and the quality of wireless communications are strongly affected by intermittent faults, we need to accommodate the possibility of the system to recover from a degraded mode of operation to which it might have switched. This requires a more versatile design approach, allowing the system to perform in several modes (Levels of Service) and to oscillate between the different modes.

1.3 Relation to other KARYON work

The presented architectural approach is strongly connected to some fundamental concepts that are being developed in KARYON, but which are addressed in other deliverables.

Firstly, there was an important preliminary work developed in WP1, concerning the definition of requirements on the architecture, which have been considered during the definition of the architecture and are hence restated and discussed here, both in an abstract way and in the context of concrete instantiations of the generic architectural pattern.

Secondly, given that KARYON is concerned with faults affecting the accuracy of the sensor data that input to the system, the architecture must encompass some means to deal with these faults. The approach that is taken is to develop an abstract sensor model, which allows considering sensors as abstract entities with a well defined interface providing the sensor data with a corresponding validity measure (which can be seen as a measure of how trustworthy this data is). The architecture explores this abstraction by including components that deal with data validity in order to manage system behaviour. Run time monitoring is thus concerned with collecting validity estimates to detect operational changes (caused by faults) to determine when to change the operation mode.

A third important aspect in the overall conceptual definition of the architecture concerns the use of environment models, which are used as fusion mechanisms for better evaluating the validity of information received from remote nodes via wireless networks. These environment models are encapsulated within the abstract sensor model and are discussed in detail in another deliverable.

KARYON is concerned with safety-critical systems. Therefore, the architecture must provide the means for handling safety concerns and for allowing safety analyses to be conducted. An underlying fundamental concept, the fourth one in this enumeration, is that safety requirements that are ultimately allocated to system components are essentially expressed in terms of the validity metrics mentioned before. This is a fundamentally new idea underlying the whole KARYON concept, which allows capturing the essence of the safety problem being addressed: in cooperative systems it is very hard to design solutions covering all possible faults, and hence some may happen which will have a negative impact on the validity of the information used in the control processes, requiring timely detection and handling in order to exclude implied safety risks. Validity is thus a key concept, both to express design time safety requirements allocated system components, as well as capture the relevant (health) state of the system in run time.

Finally, the Safety Kernel concept plays a decisive role in the management of the Level of Service of the cooperative functionalities. In KARYON we acknowledge the fact that it may not be possible to predict all possible hazards leading to faults that affect data quality or computation timeliness. Therefore, the design approach followed in KARYON is to specify the quality or timeliness that are needed to guarantee functional safety in some Level of Service, adding the necessary means to detect violations of that specification. The Safety Kernel is the part of the system where the safety rules defined in design time and the safety-related information that is collected in run time come together, and where decisions on necessary changes in the LoS are taken. The concept is present in this document, but it is developed in the context of WP4.

1.4 Structure of the deliverable

We start by introducing, in Section 2, a set of concepts and definitions that are important to give some context for the architectural options described in the deliverable. The KARYON architectural pattern is then described in Section 3. The description starts by focusing on the system and fault models, explaining the architectural components that are considered and included in the architecture, how they are structured and related to each other to form the overall architecture, and what is assumed concerning the faults that can affect each of these components. In addition, the text provides functional and data-oriented views of the proposed KARYON architecture, including a description of the necessary functional components, of their role within the hybrid architecture, and of the data and control flows that exist in a KARYON system. Some important issues that are addressed in other work packages, but which are related to the defined architectural pattern, are described in Section 4. Finally, Section 5 provides examples of how the generic architecture can be instantiated when considering specific cooperative functionalities in the avionics and automotive domains. These examples are obviously related to the use cases considered in the project and thus to the test cases that are defined in the scope of WP5 and which will serve to demonstrate the project contributions. The deliverable is concluded in Section 6.

2. Fundamental concepts and definitions

In order to give the necessary context for the KARYON architecture presented in this deliverable, in this section we introduce several concepts, definitions and terminology. These are related with the application context focused on cooperative systems, and with the objective of achieving solutions with improved performance without sacrificing safety. We also provide our view on concepts related to functional safety and to the idea of exploiting hybridization as a basic modelling approach.

2.1 Cooperative systems

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 fixed rules, without the need to communicate with each other. But is communication can take place, then vehicles might be able to cooperate in order to reach some further agreements on the way they behave. These may allow improving the traffic flow, the safety and the overall energy consumption.

Cooperation entails a number of aspects, like communication, the quality of exchanged data and the cooperation scope, which we address next.

2.1.1 Communication

When considering cooperative vehicles, communication between vehicles becomes a fundamental focal issue. This is in contrast with non cooperating vehicles, even autonomous ones, which operate in a fully independent way, without the need to actively interact with other vehicles or entities, therefore without needing to communicate.

Since the ability to communicate is absolutely required to achieve cooperation, it will not be possible to cooperate when communication channels are not available or are not functioning with the necessary quality. Therefore, there is a problem when dealing with cooperative vehicular applications, because, due to mobility, vehicles must communicate using wireless communication networks, which are known to be prone to interferences and much less reliable than wired networks. And the problem tends to be seen as increasingly difficult when considering that these cooperative applications must satisfy functional safety requirements.

Our basic approach in this respect is to accept the fact that communication might not always be possible, devising the solutions that will allow the systems to switch from active cooperation to non-active cooperation. That is, when communication is possible, then cooperation may take place. Otherwise, no cooperation will take place, although the systems will be actively trying to cooperate again and they will all be aware that it is not possible to actively cooperate.

In KARYON we are interested in the possibility of exploiting the ability to communicate, while taking care of the fact that this might not be possible. We thus focus on the dynamic aspects associated to communication with varying levels of quality, and on ensuring safety requirements despite such dynamics. When communication is not possible, vehicles will be operating just as normal autonomous vehicles that might use predefined rules for coordinating. Finding solutions to ensure a safe operation in these conditions falls out of the scope of the project, because this is covered by state-of-the-art solutions.

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 Quality of information

When sending and receiving information to/from other entities or vehicles, and being this information sent through communication networks with quality attributes varying over time, the quality of the information will also vary. In fact, it is important to note that we are considering real-time systems and real-time sensor data, like the position, the speed or the altitude, which vary over time. Therefore, when transmitting sensor data as required in cooperative applications, this data is degrading over time. If network properties such as the communication latency cannot be bounded in design time, which is the case we consider, then it becomes necessary to deal with the potential degradation of the information quality.

In KARYON we look at this problem with particular attention. We generalize the problem of failures affecting sensor data quality and devise an **abstract sensor model** for that purpose. This model also serves to deal with other failure affecting sensors, not just communication uncertainties affecting the collection of sensor data.

Based on that, in KARYON we consistently use the notion of **data validity**, as an expression of how good is data used in the control processes. The notion is reflected both in the defined KARYON architectural pattern, as well as in the way the safety reasoning in developed.

2.1.3 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 relatively close 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 out of the 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. There is a considerable amount of work in classical literature of distributed systems addressing the problem of group membership [1,4], some of which specifically focusing on wireless environments [3]. There are also impossibility results [7], setting the bounds of what may be aimed. All of these works can be considered to address the issue of defining a cooperation scope.

A remark should be made, however, to note that the particularities of automotive or avionics scenarios may allow to consider alternative (and practical) approaches to address the problem. For instance, if considering the coordinated roundabout or coordinated road crossing cases, it seems possible to consider that there may be a central entity that is part of the infrastructure, which provides scope information to all vehicles that are in a certain vicinity of this entity.

2.2 Level of Service

Managing the trade-off between performance improvements and the safety risks this additional performance will bring, implies that some risk management process is put in place. 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" as well as explain some notions related to the process of adjusting the LoS.

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 have to care about the aspects mentioned above. Engineers responsible for defining these control algorithms will try to make vehicles move faster, closer, smoothly and following straight trajectories. They will also consider (as much as possible) all the possible 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.

But system designers also have to make sure that the resulting system will perform the functions safely, that is, excluding the possibility that a vehicle will collide into another, or will hit a pedestrian (in the case of automobiles), or will fall down to the ground (in the case of aircrafts). Therefore, the control algorithms will embed the knowledge about what must be done to achieve a safe control. This requires making considerations about relevant physical processes, for instance how fast can a car move in order to stop within a certain distance given a certain condition of the road, what is the permissible roll angle of an aircraft, given speed and other conditions, to avoid losing the necessary lift or complete control of the aircraft, or how frequent is it necessary to read short and long range front vehicle sensors to ensure that a pedestrian is detected in due time, given a certain car speed and pedestrian movement model.

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**, but in fact there are also assumptions that need to be made concerning the infrastructure on which the control system is running and on the control system itself. In typical designs of safety-critical control systems, all the relevant bounds concerning the infrastructure must be known, so that they are taken into account in the design. And if some faults are known to possibly occur, they are either treated with fault-tolerance solutions that allow providing an adequate fault semantics to the system programmer, or they are considered to be irrelevant from a safety perspective, and simply ignored. In any case, there should never be any relevant variable, with potential impact on safety, whose value is not bounded and known in design time.

The resulting nominal control system, designed under a specific set of assumptions, provides a certain trade-off between the performance (measure by the metrics enumerated before) and the degree of safety. This trade-off sets the **Level of Service (LoS)** in which the function is provided by this nominal control system. In other words, **in a given LoS the maximum performance and the corresponding safety level are fixed**.

There are essentially two ways in which the LoS can be improved. One way is by finding more clever ways of defining a control algorithm, in ways not considered before. This is strictly in the realm of the theory of control systems and algorithmics of control systems, and is not what we are interested in KARYON. This is why we do not go into details concerning specific control algorithms or specific functionalities, and only use them as toy examples, namely in the definition of use cases for the proof-of-concept demonstrations. The other way of improving the LoS is by exploiting the availability of new and emerging technologies, which is what we are interested in KARYON.

More specifically, in KARYON we want to improve performance without reducing safety by leveraging on two things:

- The availability of more resources, namely computational power and an increased variety of sensors;
- The possibility to exploit cooperation between vehicles, which may be seen as an additional way of collecting more sensor data.

It may be argued that with increased resources, the single LoS achieved with a traditional approach could also be improved. This is indeed true up to a certain point, and in fact this is how safety-critical computer-based systems have been improving over the years (e.g., using faster control loops).

However, the achievable improvements are limited by the increasing complexity of the hardware and, in particular, of the software components used in these systems. This complexity leads to an increasing number of faults, and it is becoming harder and harder to mask all of them in design time, so that the system designer can just focus on the development of the control algorithms assuming a favourable failure semantics of the underlying infrastructure. Simply adding hardware redundancy, as a solution to deal with faults introduced by complexity, is not a panacea because it is not always possible to get rid of common mode failures and the approach implies relevant additional costs. And concerning the replication of complex software components, the problems are even bigger.

But the major obstacle to using traditional approaches is that wireless communication, which is required for cooperation, is inherently uncertain. This makes it hard to use remote information in control algorithms that rely on design time established bounds, in particular when these bounds must hold with a very high probability.

The option followed in KARYON is to consider that there is more than one LoS in which a cooperative function can be executed. The lowest LoS is achieved with the baseline solution, that is, with a nominal control system that is designed based on well-known bounds on every significant variable, on redundancy measures that will enforce some assumed fault model. In this LoS the achieved performance (in terms of maximum speed, minimum safety distance, etc) will be the same as the performance achieved with a classical design providing just one LoS. No additional resources are employed to run the function in this LoS. In particular, no communication is envisaged, which means that all vehicles will be acting autonomously, and all in the same LoS. In fact, in this LoS the functionality turns out to be non-cooperative, which we have previously identified as inevitable when communication is not possible.

In order to exploit the additional resources, more complex control algorithms are designed, additional sensor data is used, and a new nominal control system is defined, which allows achieving an improved LoS. In some sense, this is like having more than one nominal control

system in place, although it is clear that most components will have a single version and will be used in the different combinations yielding the different nominal control system configurations.

From the perspective of the control system designer that is defining a complex algorithm, it will be nevertheless necessary to make a number of assumptions on which this algorithm is relying. For instance, the designer may assume a certain bound on the communication latency, may assume the availability of some sensor data with a well-known quality, etc. The new algorithm will hence allow achieving an improved performance and a higher LoS, as long as these assumptions are verified.

It is clearly possible, by definition, that some of these assumptions might not hold in run time. Therefore, in a KARYON system it will be always necessary to monitor the relevant variables in order to detect possible violations of the assumptions. And if some assumption does not hold anymore, then it will be necessary to switch the system into a configuration in which no component relies on the violated assumption. In other words, there will be a set of safety rules associated to each non-basic LoS, which will have to hold for the function to be provided in that LoS.

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 of each vehicle configuration.

From a performance perspective, it is better if the LoS, whatever it may be, is consistent across the cooperating vehicles, and if this 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 have referred to a LoS that is consistent among cooperative vehicles as a **cooperative LoS**. However, we often simple use LoS to designate the local configuration of the nominal control system, irrespectively of the LoS of other vehicles.

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 should be executing in the baseline LoS, and thus they will be 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.

2.3 Functional safety concepts

Safety is an intuitive but abstract concept. One may find several different definitions but they are similar in intension. Some examples are:

- Absence of unacceptable, or unreasonable, risk.
- Freedom from accidents or losses.
- A property of a system that it will not endanger human life or the environment.

Safety can be accomplished by many different means ranging from passive safety, e.g. safety belt in cars, to active safety measures like functional safety. **Functional safety** has also many different definitions depending on applicable domain:

- Part of the overall safety of a system or piece of equipment that depends on the system or equipment operating correctly in response to its inputs, including the safe management of likely operator errors, hardware failures and environmental changes.
- For the automotive domain, ISO 26262 defines it as absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems [12].
- In the avionics domain, the RTCA-DO178B/C's Design Assurance Level (DAL), defines the safety level through the effect a failure will have on an aircraft.

Safety is a system attribute and can only be determined considering the system as a whole and the environment with which it interacts. This implies that functional safety is end-to-end in scope, i.e., safety is not a component attribute.

A vehicle can operate under different modes: human controlled, autonomous, coordinated and cooperative. It may also be a possibility of having mixed modes if subsystems are allowed to operate under different modes. An operational situation can be defined as a limited view, in time and space, of the environment in which the vehicle operates. The operational mode is also included in the operational situation. It is the operational situation and its potential sources of harm that is analysed for safety. The risk is defined as the product of the probability of occurrence of harm, or exposure for harm, and the severity. The severity is an estimation of the danger to human life or environment. In ISO 26262 an automotive safety integrity level (ASIL) is determined by a combination of: exposure, severity and controllability. The controllability is defined as the ability for involved persons to timely interact in order to avoid the harm or control the harm. It should be noted that all three parameters have a dimension of uncertainty (probability, estimation and ability). A safety integrity level is a measure of how much risk reduction the system must achieve in order to be safe. Safety integrity level is tightly connected to data integrity for functional safety. Data integrity is related to the accuracy, precision and consistency of data over its life cycle. Functional safety is therefore related to the means of maintaining and assuring data integrity.

The ability for vehicles to communicate and share information is a necessary condition for having a cooperative system of vehicles. A more formal definition of a cooperative (control) system of vehicles is a set of individual vehicles which form relations in order to share information and act together to achieve a common objective. There can be several objectives for cooperation e.g. comfort, traffic management, reduce fuel consumption etc. It can be argued that shared information can be used for risk reduction and thus possibly increase safety. A cooperative system per se does not necessarily mean that additional risks have been introduced in the system and thus a need for risk reduction. Referring to the definition from ISO 26262 there are no arguments that the internal system of a vehicle should become degraded or malfunction just because the vehicle participates in a cooperative system. In fact, this will depend on the health of the communication network, as we discussed in the previous section. One of the risks introduced by a cooperative system is when the vehicles separation in time and space are lower than what had been considered safe when the vehicle operates alone. The

vehicles must still be safe from their internal perspective but there are new uncertainties introduced about the states of the other vehicles. This uncertainty and risk is reduced by communicating and sharing information about important states. Thereby the vehicles can safely operate much more tightly to achieve better traffic management and thus better traffic flow. As long as the cooperative system has a common view of those states that is good enough for current operational situation, the system is safe. If there are uncertainties about the common state or different views, then the cooperative system is potentially unsafe.

The safety analysis is always static and today's approach is to take the worst case assumptions and make the system safe according to those assumptions. A vehicle is a static entity and this approach is viable for today's situation and operating modes. Moving to fully autonomous, coordinated and cooperative mode, the controllability factor for involved persons will become much degraded. The vehicle will change from a closed system to an open system, i.e. a system that interacts with its environment. The system must also be adaptive to handle a dynamic environment. The safety analysis must shift from a self-view to a relational-view. Today's safety standards do not address these new situations.

To be able to take full benefit from coordinated and cooperative systems it seems necessary to move from a purely static view on safety integrity levels during design time into partly dynamic safety assertions in run time. By doing so, the actual run time operational situation will dictate the current need for some of the safety integrity levels. This implies that the safety analysis has to be stored in the vehicles as safety rules which map operational situations to safety integrity levels. The vehicles must have a representation, an environment model, of current operational situation. This must semantically be in accordance to the model of operational situation that was used for the static safety analysis. Hence the safety analysis is still static but it is no longer the worst case scenario that dictates the needed safety integrity level. The selection of some of the safety integrity levels becomes a dynamic run time process as well as the assertion that those integrity levels are achieved. If they are not achieved measures have to be taken to avoid the unsafe situation. One such measure is to separate the individual vehicles of the cooperative system in time and space to be more fault tolerant.

2.4 Hybridization

The concept of architectural hybridization plays an important role in KARYON, in particular in the definition of the general architectural pattern. KARYON proposed to explore the concept as a baseline design principle and therefore we explain the concept in this section.

We start with an overview of the different and fundamental approaches for defining system models. We are essentially interested in showing the difference between homogeneous system models, which assume that the entire system enjoys the same set of properties, and hybrid system models, in which different properties may be assumed for different parts of the system. This is an important difference, with impact on how solutions are designed and on how the system will perform. We describe specific advantages of using hybrid system models in comparison to homogeneous ones.

However, simply assuming that a hybrid system model is adequate to represent the real system is not enough. This must be reflected on the architecture and it is necessary to materialize the assumptions, ensuring that they hold in practice. This is why architectural hybridization is essential, as it defines a set of principles for architecting the system and, in fact, enabling the construction of hybrid systems. We also elaborate on this, providing some examples of architecturally hybrid systems.

2.4.1 Hybrid system models

In a general sense, when designing a system or an application, or simply the solution for a given problem, it is necessary to clearly identify and specify a set of requirements for that system or problem, and a set of assumptions about the properties of the environment for which the problem is to be solved. While the set of requirements is what defines the problem, the set of assumptions has an implicit impact on the possible solutions, determining, for instance, their complexity. The set of assumptions allow to characterize the relevant attributes of the environment for which the solutions will be developed, and thus constitute the system model.

The system model provides an abstraction of the real system, allowing for the separation of concerns between the underlying system properties that the solution designer can take as granted, and how these properties are provided or enforced. Therefore, when we use the term system model we refer to an abstract representation of a real system, hiding details related to hardware, network and software components.

Abstracting is good, but it is important to ensure that the abstraction is accurate with respect to the reality it represents. There is an issue of assumption coverage [17] that is relevant when the solution is deployed, which is that assumptions must hold with a high enough probability given a concrete system and environment. In essence, the right assumptions must be made. Additionally, the system model should be simple enough to be useful when designing some solution, but it should also be detailed enough to capture the essential characteristics of the system and allow better solutions to be defined.

Assumptions can be defined along several dimensions, depending on what is relevant for the problem at stake. For instance, in the distributed systems literature [14] a distributed system model includes assumptions about: (i) failures, (ii) synchrony, (iii) network topology and (iv) message buffering. In KARYON we define fault models for system components, but are not concerned with topology issues not message buffering issues, which we consider to be abstracted by the communication subsystems. Since we consider systems that interact with their physical environment, the temporal and timeliness aspects are also important, and thus it is relevant to devote attention to synchrony assumptions, defined by a synchrony model. In fact, fault assumptions can be related and may depend on synchrony assumptions, in the sense that if some synchrony is assumed, then it might be necessary to also assume timing faults in the fault model. The same could be said regarding security-related assumptions and their implications on the fault model. However, in KARYON we do not consider security issues.

There is a wealth of knowledge on the definition of homogeneous system models, and on their use in the definition of algorithmic solutions, architectures and systems. For instance, when considering the synchrony dimension, the two well-know models of synchrony that have been traditionally used are the synchronous [13] and the asynchronous [11] models. The shortcomings of these homogeneous models are clear when dealing with problems where it is necessary to reconcile predictability with uncertainty [23], such as we do in KARYON.

Recalling the KARYON main objective, which is to provide system solutions for predictable and safe coordination of smart vehicles that autonomously cooperate and interact in an open and inherently uncertain environment, the need for reconciling predictability with uncertainty is evident. Let us reason again in terms of the synchrony dimension. Should we consider the asynchronous system model, we would have no way of addressing timeliness requirements and providing timeliness guarantees for the behaviour of the developed systems. In essence, ensuring functional safety would not be possible, given that even simple hazards require some (temporally) bounded system reaction, something that cannot be handled when considering an asynchronous model. On the other hand, despite the technology improvements in computing and communication, we should also not use a synchronous model to characterize the system homogeneously. For example, to deal with uncertain wireless communication delays, a synchronous model would either postulate a very high bound for the message delivery delay,

which could be unacceptable for performance, or else, by postulating a lower bound, the risk of violating the assumption could be too high and unacceptable.

It is possible to move away from the extreme sides of the spectrum of choices (be it about synchrony, security, integrity, or others), defining intermediate models for whatever considered dimension. For instance, in the synchrony dimension there exist models of partial synchrony, such as the Partially Synchronous model [10] or the Timed Asynchronous model [9]. In these cases, synchrony is assumed to vary over time and, in this sense, is not an invariant property. However, since the property is assumed to be common to the entire system, the synchrony model is still homogeneous in the space dimension.

In contrast with homogeneous models, a hybrid system model allows possibly several stripes of the assumption spectrum to be represented, exploiting the space dimension. Then, provided it is possible to find a mapping of such hybrid models onto (correspondingly hybrid) architectural models that reflect reality (the networking and computational environment), it will be possible to exploit the increased expressiveness of the hybrid models to design improved solutions and, in particular, to address the conflicting goals of predictability and uncertainty.

In essence, hybrid system models represent systems in which different parts have different properties and can rely on different sets of assumptions (e.g., faults, synchronism). Interestingly, it is possible that some of these assumptions, applicable to some part of the system, lie in some intermediate point of the possible spectrum. Therefore, hybrid models allow the best to be taken from both dimensions: different loci of the system may have different properties, and these properties may vary over time.

In theoretical and practical terms, hybrid models have a number of advantages when compared to homogeneous models, as explained in what follows (a detailed discussion can be found in [24], focusing in particular on synchrony models).

Hybrid systems models are:

- Expressive models with respect to reality— Real systems are not homogeneous. Whatever the dimension (synchrony, integrity, etc) they generally have components that enjoy different properties, because these components use and depend on different resources (e.g., hardware devices, networks). Homogeneous models simply cannot take advantage from this, being confined to use worst-case assumptions (e.g., the most severe failure mode, the weakest synchrony).
- Sound theoretical basis for crystal-clear proofs of correctness— By using a hybrid model, the heterogeneous properties of the different loci of the system (the space dimension) are by nature represented, and we are in consequence forced to explicitly make correctness assertions about each of these loci, and about the interfaces to one another. In contrast, in homogeneous models (and particularly if they make weak assumptions) designers are tempted to make implicit assumptions that are not explicit in the model, which may lead to problems ahead.
- Naturally supported by hybrid architectures— Sisters to hybrid systems models, hybrid architectures accommodate the existence of actual components or subsystems possessing different properties than the rest of the system. Hybrid models and architectures provide feasibility conditions for powerful abstractions which are to a large extent not implementable on canonical (homogeneous) models: timely execution triggers (also known as watchdogs); secure signatures or highly reliable execution kernels. Hybrid models and architectures may drastically increase the usefulness and applicability of all these abstractions.
- Enablers of concepts for building totally new algorithms— A powerful yet simple concept behind the first experiments with hybrid models was: use the weakest possible model for the generic system; imagine that a “toolbox” of simple but stronger low-level services is available, locally accessible to processes (e.g., timely execution triggers;

timely executed actions; trusted store); these local services can be distributed via alternative channels, to obtain further strength (e.g., synchronous channels; trusted global time; trusted binary agreement); devise algorithms which, by working between the two space-time realms, the generic and the enhanced subsystem containing the "toolbox", achieve new properties (e.g., making an asynchronous process enjoy timely execution).

Having explained the concept of hybrid system models, and their advantages over homogeneous models, in the next we address the architectural hybridization principle, as a fundamental enabler of the concept.

2.4.2 Architectural hybridization

Hybrid modelling of distributed systems is the path to achieving incrementally stronger behaviour taking the best of two worlds: retaining essentially weak models (of integrity, synchrony, security, etc), with consequent benefits for correctness (since assumptions are hardly violated); allowing strong models to be considered, which are essential to fulfil predictability and safety needs.

Architectural hybridization was proposed as a new paradigm to architect modular systems, based on a few simple principles:

- Systems may have realms with different non-functional properties, such as synchronism, faulty behavior, quality-of-service, etc.
- The properties of each realm are obtained by construction of the subsystem(s) therein.
- These subsystems have well-defined encapsulation and interfaces through which the former properties manifest themselves.

As to the construction, architectural hybridization is an enabler of the construction of realistic hybrid systems. In fact, it is quite straightforward to build architecturally-hybrid systems, and we provide some examples below.

The first example is of a system with a watchdog subsystem. The watchdog is used to reset or restart the overall system when something wrong happens in the main part of the system, typically when the main system becomes slow or inactive. The watchdog is essentially a counter device, which has a register that is programmed with some value, and a counter register that is continuously incremented. When the value in the counter register equals the value in the programmable register, the watchdog activates the reset signal. The main system has to periodically reset the programmable register to a higher value, to avoid system resets. When the main system becomes slow or stops, this will be implicitly detected because the programmable register will not be reset on time. In this example, it is easy to see that the system has two different parts, and is thus architecturally hybrid: the main system, which is assumed to fail or to behave untimely, and the watchdog, which is assumed to behave correctly and timely. These are reasonable assumptions, because the watchdog is essentially independent from the main system and it is a much simple subsystem. This ensures that faults affecting the main system will not propagate to the watchdog, and due to its simplicity the probability of the watchdog failing on its own is much lower than the probability of failure of the main system. Interestingly, the resulting global system exhibits better properties than the main system alone: it will either behave in a timely way or it will restart. In any case, untimely behaviours have been ruled out and this may be a useful property in many situations, when a fail-stop behaviour is admissible.

A second example is of a system with a Timely Computing Base (TCB) [22]. In such a system there is a generic part, called payload part, which corresponds to the baseline system where application processes execute to provide the intended application functionality. Then there is a control part, called the TCB, which like the watchdog is a separate part, but which provides richer supporting services to the payload part, like timing failure detection and timely execution

of critical functions. It must be noted that the services provided by the TCB are distributed services, which implies that hybridization is extended to the network architecture. Clearly, the TCB part must be implemented in such a way that it enjoys better properties (reliability and timeliness) than the payload part. Some construction principles like interposition (ensuring that accessing to critical resources cannot be made bypassing the control part) and shielding (the control part is protected from faults affecting timeliness) must be respected to make sure that the services can be provided with the expected properties. One implementation of a TCB was done using Real-Time Linux and two switched Ethernet networks [5], where one of the networks was used exclusively for the TCB, whose services were implemented as real-time tasks. In this system, the payload part was the normal Linux part, using the other network. Another example implementation of a TCB, in which a completely separate hardware platform was used for the TCB subsystem, is described in [16].

One final example is a system with a Trusted Timely Computing Base (TTCB), in which hybridization is used not only to achieve a timely subsystem, but also a trusted subsystem, capable of providing security-related services like trusted random number generation and trusted block agreement [8]. Although the TTCB described in [8] was also implemented in Real-time Linux with specific changes in the kernel to enforce security properties, other COTS trusted hardware, such as the Trusted Platform Module (TPM) [20], can be used to obtain tamperproofness. In fact, a TPM can be seen as a special subsystem with better (security) properties which, when used in a generic (unsecure) system, ends up forming an architecturally hybrid system.

As to the usefulness of architectural hybridization, and considering the previous examples, it is clear that the overall system will be improved by making it able to use the services of a better component or a better subsystem. For instance, in the case of the TCB it is possible to perform timely actions despite the asynchrony of the payload system, or to detect and react in a timely way to possible delays occurring in the payload part. On the other hand, with a TTCB it is possible to drastically augment resilience to intrusions, making it possible to solve fundamental problems such as consensus in the presence of uncertain attacks and vulnerabilities [15]. Note that in homogeneous systems, where the same fault, synchrony or security model applies to the entire system, the only way to achieve the intended (e.g., synchrony, security) properties is by enforcing these properties in the entire system, which is typically an over killer. With architectural hybridization, only the restricted part of the system that has the better properties needs to be constructed with the aim of achieving those properties, which is much easier. And still, the provided services will make it easier to solve many problems that would otherwise not be solvable.

In KARYON we apply architectural hybridization to structure the system in two parts: the part where complex functions might execute without the need to ensure, in design time, that they will meet some timeliness requirements, and the part where the baseline system will be implemented, which encompasses all the components that are needed to provide a baseline LoS and the components that will be in charge of managing the system configuration in order to secure the safety requirements. While in this deliverable we provide, in Section 3, a description of KARYON architectural pattern in which this hybrid structure is visible, a more detailed discussion of the means to enforce architectural hybridization is provided in the scope of Safety Kernel definition, in deliverable D4.2 of WP4.

3. KARYON architecture

This section is dedicated to the description of the generic KARYON architecture. We start by providing a description of the nominal control system architecture, describing the architectural components and explaining how we structure the system in accordance to the considered hybrid system model. Then we describe the considered fault model, also referring to the abstract sensor model which is considered for abstracting sensor faults. The complete architectural pattern including the Safety Kernel is then described, with an explanation of the interactions between the architectural components and providing details on the functional role of the Safety Kernel components. After that we discuss the architecture from the perspective of information flows, introducing the main data abstractions that we need to consider and explaining the data flows between the functional components. Finally, we refer to the requirements listed in the previous section, discussing how the presented architecture contributes to address those requirements.

3.1 Nominal control system architecture

The bottom line for the definition of the KARYON architecture is the knowledge that we are essentially dealing with control systems, involving elementary components such as depicted in Figure 1.

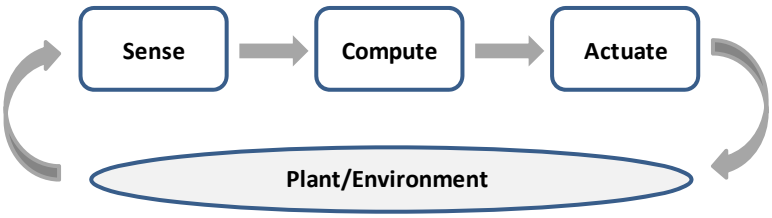


Figure 1: Basic control loop.

A basic control system involves sensing, processing and actuation. We say that this is the *nominal control system*, because it includes the strictly necessary components to realise the desired control activities. This view abstracts the existing software and hardware components, as well as the communication channels connecting the components. In this view, there is an implicit feedback that develops through the environment. That is, through actuation it will be possible to change the behaviour of the controlled entity (which in KARYON is a vehicle), and this change will be perceived through the observation of physical variables that develop through the environment, like the ground relative speed or the distance to some physical object.

It is well-known that it is easier to ensure control stability, in which controlled variables are kept within desired bounds, when the system and the environment are well defined and all variables can be characterized in a precise way. From a modelling perspective, this translates into considering synchronous models, well-defined failure modes, known event patterns, etc. Control stability is fundamental to meet safety requirements, whenever these are defined.

Solutions for stable and safe control under precisely defined conditions are well known in fully described in the literature. They imply a detailed system analysis at design time (i.e., statically), to prove that the necessary safety conditions are met. However, in KARYON this simple model is not enough, in particular because we consider a distributed system that requires communication elements to be added to the picture.

In a first step, we enrich the initial simple model by making explicit the fact that in a control system there may exist, in fact, several sensors, computing elements and actuators. We also make explicit that this basic system composed of sensing (Sense), computing (Compute) and actuation (Actuate) components, is the nominal control system, as shown in Figure 2. In fact, the nominal system is the target system that we want to improve, so that the cooperative

functions that are realised by using the components of this system will achieve a better performance, as defined in Section 2.2.

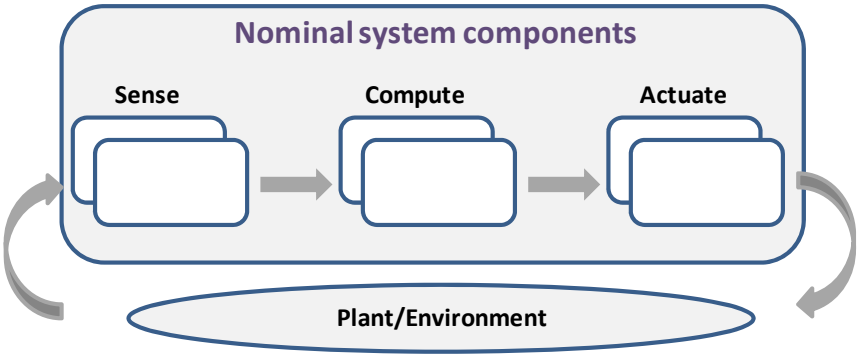


Figure 2: Nominal control system.

We consider that with the represented components it is only possible to provide local functionality, because none of the components supports the interaction with other nominal systems, which would be necessary to provide cooperative functionality. Therefore, in order to explicitly represent the need to communicate with other nominal systems, which is needed in KARYON, we add communication components to the nominal system model.

The new model is shown in Figure 3 and it now includes all the component types that we need. Sensing and actuation components implement the interface between the system and the environment. Sensing components consume information from the environment and produce information to the system. Actuation components, on the other hand, consume information from the system and produce information to the environment. Computing and communication components are just different in the sense that they consume and produce information from and to the system.

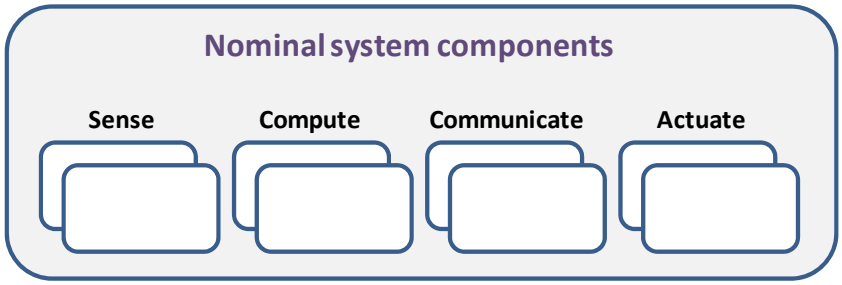


Figure 3: Nominal system for cooperative functionality.

Communication components provide networking functionality, that is, they provide the means to connect a nominal system to other nominal systems. This communication is performed through *operational networks*, which may differ from the internal networks that are used to connect the nominal system components regarding synchrony and reliability. Interestingly, it would be possible to use sensors and actuators to abstract the collection of data from, and production of data to, remote nominal systems. But in this way we make explicit the fact that we consider distributed systems that need to communicate.

It is important to say that the set of components that constitute the nominal system can be used in the provision of multiple functionalities. Adding a new functionality can thus be done by reusing some of the existing components and, possibly, adding just a few new ones.

When considering the need to support cooperative functionalities, and when adding communication components to the nominal system, we are implicitly adding uncertainty, which cannot be handled at design time. In fact, since communication will be done though wireless networks, this implies that it will be hard and inappropriate to assume fixed upper bounds for communication latency. Again from a modelling perspective, we are moving away from purely

synchronous models and strong failure modes, which would allow us to use the well-known techniques for building real-time safety-critical systems. But this is what we proposed to do in KARYON, that is, exploiting cooperation to achieve improved functionality, while dealing with the increased uncertainty that this will bring to the system.

Besides exploiting cooperation, the objective is also to exploit the possible use of more complex control algorithms and of a richer variety of sensors. As we have discussed previously, this implies that functionality will be provided with more than one LoS, which must be reflected in the system architecture. In particular, and given that we introduce complex components whose behaviour might not be proven in design time to satisfy some bounds, the architecture will not be homogeneous anymore. As illustrated in Figure 4, the nominal system is separated in two parts by a so called "hybridization line".

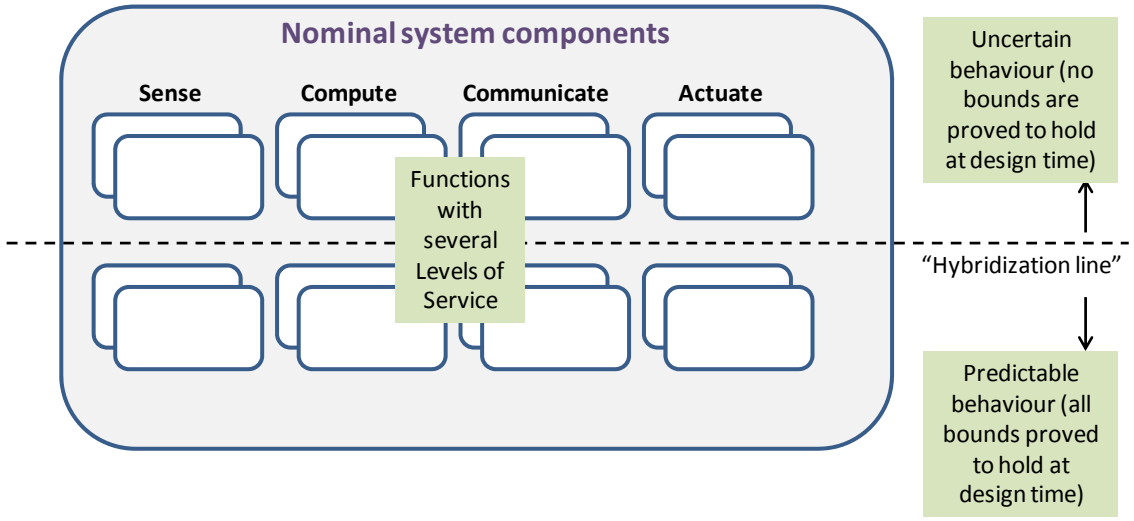


Figure 4: Nominal system for improved performance.

The hybridization line separates the system in two parts, denoting the application of the architectural hybridization paradigm. This allows making explicit the fact that different properties are assumed for each of the parts above and below the line.

In general, as depicted, all kinds of components, from sensors to actuators, could in principle be found either above or below the hybridization line. For instance, some additional distance sensors could be added which would provide more precise distance information under good lighting conditions. However, these sensors will not perform well at night, and therefore it would not possibly to assume fixed error bounds on the information collected from these sensors. Similar examples could be given for processing and communication components. Concerning actuation components, the situation is different because these are the final components in the control flow, which means that uncertainties affecting actuators will be directly reflected in uncertain actuation on the environment. This is undesirable and would require special solutions based on interposition or on the use of redundancy to overcome uncertainty. We nevertheless leave actuation components on both sides of the hybridization line to express the theoretical possibility of using complex actuators, despite the fact that in the proposed architecture we do not provide specific measures to deal with actuator uncertainties. In fact, in the fault model described ahead, we consider that actuators are reliable and predictable.

3.2 Fault model

In KARYON we consider that some components can experience faults, which will have an impact either on timeliness or on the quality of data. In fact, one of the main motivations of the project is to provide solutions for dealing with the effects of faults, which are likely to occur when adding complexity to the system and when relying on wireless networks. In what follows, we describe the faults assumed for the components represented in Figure 4.

Sensor components can experience various faults affecting their output. When sensors are above the hybridization line, these faults can be both in the time and in the value domain. Otherwise they can be only in the value domain.

Faults in the time domain include crash faults, i.e., when a sensor does not provide further output, and timing faults, when a sensor produces a late output. Because a sensor data is a time-value entity (i.e., the value of the entity is varying over time), a timing failure, if not detected, may be appear as a fault in the value domain further ahead in the system (e.g. a late position information is a wrong position information). Sensor failures are abstracted in a data centric way and expressed by a validity estimate.

There is a subset of sensors that is always correct to the needed extent. This is necessary to provide the baseline LoS, and prove safety in design time. The required sensor properties, namely reliability, must be achieved by construction, for instance using redundancy.

Computing components above the hybridization line can fail by stopping or producing late outputs, but they do not produce value faults (and no Byzantine behaviours are accepted, like sending inconsistent values to sets of other components). Below the hybridization line, it is assumed that all computing components are as correct as necessary (by implementation). In other works, in the case of these components all the potential risks to safety have been removed in design time.

Communication components can be modelled just like sensor components. In fact, it is possible to apply the abstract sensor model (explained in D2.2) to communication components and express communication faults by a validity estimate. Above the hybridization line, communication components can either crash of produce timing faults. If no abstract sensor model is used, then timing faults will have to be handled in the same way as with computing components. In principle, all communication components will be above the hybridization line, because it is hard to achieve timely communication in wireless networks with sufficiently high probability, as usually necessary in the considered applications.

Actuator components are assumed not to fail. In KARYON we are essentially worried with faults affecting the correctness of perception, rather than the correctness of actuation. Failures in the mechanical system including the mechanical parts of the actuators are outside the scope of KARYON and are not considered (this is actually the point in fault-tolerant control that changes the control laws for the plant to deal with this class of failures). The hardware/software controlling an actuator is not considered fault-free a priori. Depending on the SIL that is required to run a specific control function, the adequate reliability (validity of data controlling the actuator) has to be ensured.

3.3 Complete architectural pattern

In order to deal with the possible faults affecting the timeliness of computing components or the quality of sensor data, it is necessary to include in the architectural pattern some components that will be in charge of managing the system configuration, that is, of adjusting the LoS of the functionalities using the faulty components.

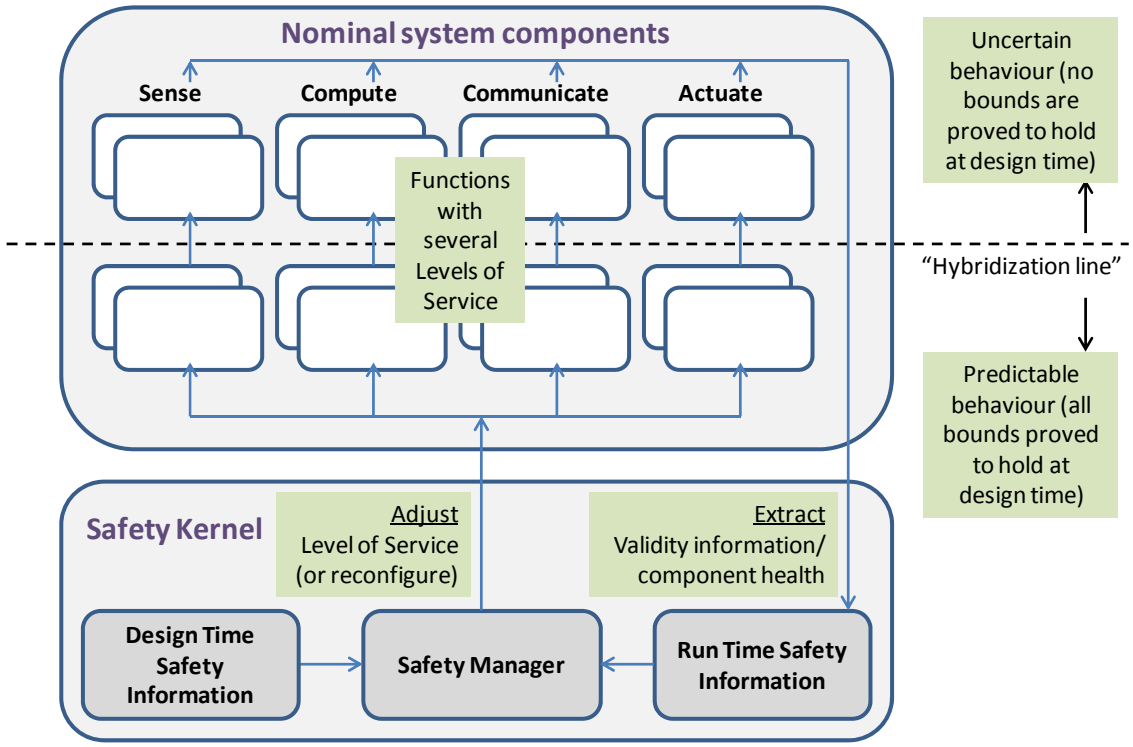


Figure 5: Complete KARYON architectural pattern.

In addition to the nominal system we add a *Safety Manager* component and associated *Design Time Safety Information* and *Run Time Safety Information* components. They constitute what we have been generically referring to as the Safety Kernel. We also highlight the fact that these components are located below the hybridization line. This is necessary because it is fundamental to make sure that the Safety Kernel enjoys the properties of the "better" part of the system and can thus be proven in design time to satisfy all the requirements that were defined as an outcome of the safety analysis. This structuring is in accordance with the principles of the architectural hybridization paradigm.

The Safety Kernel is separated from the nominal system in order to make it clear that the components belonging to the Safety Kernel are generic, i.e., they can serve any particular application. On the contrary, each component of the nominal control part is application-dependent. In other contexts, namely in the definition of the Safety Kernel architecture in D4.2, we draw a line separating the Safety Kernel from the nominal system, to which we call "Semantics line" precisely because the only the nominal system components are dependent on the application semantics.

We now examine the proposed architectural pattern under the light of the hybrid modelling approach, we provide a functional description of the Safety Kernel components, and we discuss the overall system behaviour in order to adjust the level of service of each functionality to match the available conditions and meet the safety objectives.

3.3.1 Exploiting hybridization for improved performance

Above the hybridization line we have the nominal system components that implement complex algorithms or realise functions whose results may be uncertain, namely due to faults. These components can be used to achieve improved performance, without the need to ensure that they will always satisfy some requirements concerning timeliness or concerning the quality of the data they produce. These components are used in combination with other components that may be below the hybridization line. Altogether, they constitute the nominal control system that provides the intended functionality (or several ones), possibly with a variety of LoS for each

functionality, each of them corresponding to a certain combination of components or, in other works, a certain system configuration. It is not possible to prove at design time that the functionality will be safe in some arbitrary LoS independently of the anticipated conditions and faults. That is, some functionality provided with a certain LoS might not be safe if the conditions degrade, namely when there are failures affecting some components and leading to degraded data quality. However, it must be possible to statically prove that the functionality will be safe in a certain LoS, provided that some assumptions are satisfied. In other words, it is necessary to define in design time the safety rules that must be satisfied for each functionality to be safely provided in each LoS. In addition, it is necessary to make sure that the LoS will be adjusted in run time to guarantee that safety rules are always satisfied. In this pattern, the lower LoS will always correspond to a configuration in which all the components are below the hybridization line.

Above the hybridization line it is possible to explicitly accept weaker fault and synchrony models, which can be satisfied with less expensive resources, also allowing the use of a wider range of technologies (e.g., wireless networks, soft real-time schedulers) that are compatible with those weaker assumptions. The system will be dynamic and adapt in response to faults and to the available integrity level.

Below the line the system will be static. All the functional components must be statically proven to satisfy a set of safety requirements, and if any faults could possibly affect these components, the necessary fault tolerance measures will need to be implemented to handle these faults. This means that these components, and the Safety Kernel in particular, will always operate correctly with respect to the assumed system and fault model for this part of the system. In addition, since the components that are needed for providing some functionality with the baseline LoS are all below the hybridization line, the functionality will also be provided in a safe way.

When adding complex components (above the hybridization line) or simply alternative components for executing some functions (below the hybridization line), the Safety Manager becomes fundamental to manage the system configuration, allowing the system to switch between levels of service depending on the observed run time safety information.

3.3.2 Functional description of Safety Kernel components

In contrast with the baseline model of a control system, shown in Figure 3 or in Figure 5, in Figure 5 it becomes clear that there is an additional control loop, in which the nominal system is being controlled by the Safety Kernel, more specifically by the Safety Manager component. In the execution of this control loop it is necessary to use design time and run time safety information.

3.3.2.1 Run Time Safety Information

According to the considered hybrid system model, we assume that some nominal system components might exhibit untimely behaviour. Other components, below the hybridization line, may behave timely but may still be affected by faults that will be reflected on the quality of data they produce, which flows to other system components. For instance, a sensor that may fail in several different ways will produce sensor data with varying validity, depending on the concrete faults that may have occurred and their direct impact on the sensor data values. And a communication link experiencing interferences may omit or delay the delivery of some messages, which will degrade the validity of data that depends on the timeliness of this communication.

A crucial aspect of the proposed architecture is that instead of requiring the enforcement of some data validity or integrity levels, it just requires awareness about the validity of the data flowing in the system. From a more practical perspective (which will be discussed further ahead), we do not need to enforce predictability, because we use mechanisms to monitor the

relevant variables and detect timings faults, which is all that is needed to decide the LoS under which some functionality should be executed. The set of collected information is represented in the architecture by the *Run Time Safety Information* component, which also abstracts the concrete mechanisms that must be put in place to do this information collection.

It should be noted that it is possible to collect different kinds of information that may serve to derive the validity of data and directly reason about safety. In fact, it may be possible that some of this data directly reports on the health of components, explicitly providing indications about the occurrence of faults affecting the component behaviour. For instance, it may be possible to know that some component crashed, simply stopping producing information.

Finally, we also note that knowledge about the context, meaning the physical surrounding environment, is usually important to reason about safety when considering vehicles that move and interact with this physical environment. This means that not only the validity of data is important for safety, but data itself is necessarily important (if this data describes the physical context). However, since this data is to be used in the control algorithm, changes in the context can be reflected on how the function is performed rather than on the LoS.

3.3.2.2 Design Time Safety Information

From the safety analysis of each cooperative functionality, safety requirements are derived and allocated to system components. Given that a functionality can be provided with several LoS, and given that in a lower LoS the performance of the functionality is limited as a means to exclude some risks, the integrity requirements associated to a lower LoS will be less stringent than the integrity requirements associated to a higher LoS. For instance, for a functionality to be performed by vehicles at a possibly high speed or with lower safety distances (as allowed in a higher LoS), it will be necessary to make sure that all sensor information used in the control processes will be highly accurate (high validity) and that all components required to provide this LoS are performing as expected (e.g., timely). On the other hand, at low enforced speed or with a large enforced safety distance (determined by a lower LoS), the requirements on the accuracy of data will be less stringent (low validity will be accepted), and the dependence on some components (possibly untimely) will be excluded. This means that even if some failures are affecting the health of components or the accuracy of data, the functionality can be provided in this LoS.

It is clear that for a functionality to be provided with a certain LoS some requirements concerning the validity of data and timeliness must be satisfied. And there is a different set of safety requirements that must be satisfied for each LoS.

We remind that there are other safety requirements which must be considered by the designer of the control algorithms (implemented in nominal system components), as previously explained in Section 2.2. These are requirements specific to the application or functionality, which are taken into account in the control algorithms. And they may be different for each LoS in which the functionality might be provided. For instance, this means that the actual speed of a vehicle is not only determined by the LoS (which only sets a bound on the maximum speed), but by other control rules defined in the control algorithms (which take into account context information, like road or weather conditions).

When designing a control algorithm that will be used in some specific LoS, the designer will know that assumptions concerning timeliness and sensor data validity will be satisfied, so he/she can rely on this knowledge in the design. And it will be the responsibility of the Safety Kernel to make sure that a certain LoS is enforced, in accordance to the observed timeliness and sensor data validity.

The Safety Kernel therefore needs to know what is required for all functionalities to be provided safely in each LoS. This is determined in design time and hence the design time safety information consists of sets of safety rules establishing the conditions for functional safety

assurance in each LoS. One functionality will only be safe in a given LoS (above the baseline one), if the associated set of safety rules are satisfied at run time. As discussed above, this necessarily depends on the validity of data and on the timeliness of complex components, which may be affected by faults. Therefore, in order to manage the LoS it is necessary to have both the set of safety rules and the collected run time safety information.

Note that when a function is provided in the lowest (baseline) LoS it is not necessary to verify if safety rules are met, because this has been done at design time. In other words, the requirements on the system health and validity of data associated to the provision of a functionality in the lowest LoS, must be guaranteed at design time.

For each LoS there is an associated set of safety rules. It must be proven at design time that if the safety rules are met, then the function will be safe in this LoS. The same has to be done for all LoS and all sets of safety rules. However, it is not necessary to prove that these conditions will be met at run time. In fact, this is what distinguishes a higher LoS from the lowest LoS. For the latter it is necessary to prove that: a) the function will be safe if safety rules are met and; b) safety rules will be satisfied in run time. The corollary is that the function will be safe in the lowest LoS.

In run time, what needs to be done is to compare the current state of the system (conveyed by the run time safety information) with the safety rules for the current LoS. This is one of the tasks of the Safety Kernel.

3.3.2.3 Safety Manager

The role of the Safety Manager is to control the mode of operation of the nominal system components and hence adjust the LoS of each function. In order to do that, the Safety Manager needs to know the actual state of the nominal system, which is provided by the run time safety information. Then, given this state and given the safety rules defined at design time and stored as design time safety information, it decides whether the current LoS can be kept, or if the conditions determine a change (either to a lower or to a higher LoS). This basic behaviour is illustrated in Figure 6.

The Safety Manager is periodically evaluating rules and determining possible adjustments of the LoS for some functionality. The period must be well-known so that it may be possible to prove that the function will be provided safely.

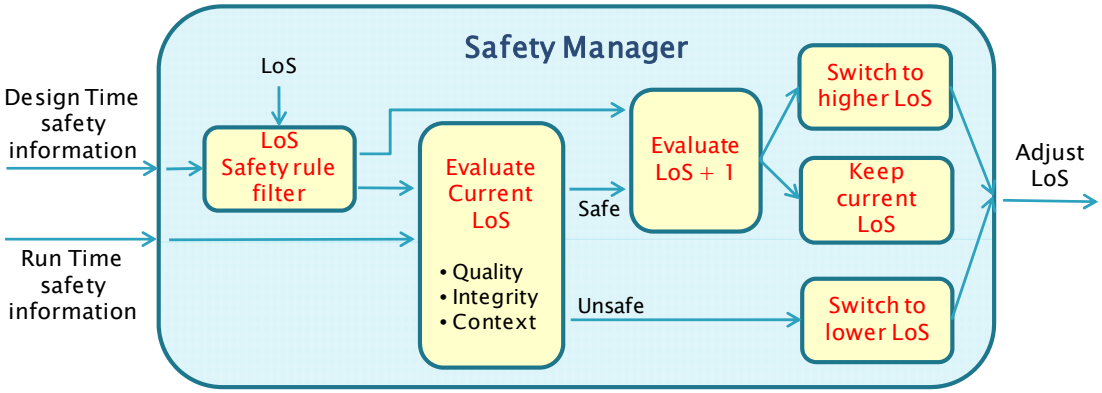


Figure 6: Safety Manager basic behaviour.

Given that there is a different set of safety rules for each LoS, the current LoS will determine which set of rules will be firstly evaluated. If some rule is not satisfied, it becomes necessary to change the operation mode of nominal components, to force a switch to a lower LoS and prevent the system to enter in an unsafe situation. On the other hand, if all rules are satisfied, then the Safety Manager will need to evaluate rules for the next higher LoS (if it exists).

Depending on the outcome, it may be possible to switch to the next higher LoS, or keep in the same LoS.

Given that the Safety Manager executes in a timely manner, and that it will execute with a known period, it will be possible to know, in design time, how much time it may take to switch from any higher LoS to the baseline one. This amount of time must be known by the function designer and must be taken into account in the design of the control algorithms, namely in the establishment of safety margins.

The specific solutions to collect run time safety information and to trigger reconfigurations or simple adjustments of the nominal system components are not discussed in this deliverable. They are explained in deliverable D4.2, concerning the Safety Kernel architecture.

3.4 Information flow view

The description provided so far was essentially focused on the components and their functions, explaining why they need to be located above or below the hybridization line. Now we pay more attention to the interactions between the components, providing a data-oriented perspective of the KARYON architecture.

Central to this view is the notion that we have two fundamentally different kinds of data. On one hand, there is application or service related data, which is necessary for the provisioning of the intended cooperative functionality. Considering the basic model of a control system presented in Figure 1, this is the data that flows from the environment through the sensors, computing components and actuators, back to the environment. In the absence of relevant risks, this basic control model would be enough and there would be no need to consider any other kind of data. However, given that the nominal control system is subject to faults, we need to handle these faults and, for that, we need to collect information regarding the health of the system. Therefore, besides the sensor data that flows through nominal system components, there will be a flow of data concerned with data quality or timeliness. In fact, the purpose of the abstract sensor model that we elaborate in KARYON is to abstract sensor faults and to provide this quality information, through a validity attribute.

These two different kinds of data are shown in Figure 7 as “Service data” and “Quality data”, and are included in the Run Time Information Database, which is just an abstraction to represent all run time produced data. Static information, on the other hand, is abstracted by the Design Time Safety Information Database, which specifically contains the safety rules derived in design time.

In addition to the information databases, in the figure we represent the components of the nominal control system and the Safety Manager. This data centric view is necessarily very abstract, in accordance with the functional view presented earlier. Since we do not consider concrete functions that could be used in some cooperative application, we just know that there will be information flowing among these components, but we do not define concrete flows. What is relevant is that all the information (service and quality data) produced by sensing, communication and computing components constitutes run time information that may be required by other components, and should be made available to them. There may be several possible approaches to implement the communication between each component, but these need to be defined at the implementation level. The only requirement on the implementation is that it must be possible to support information exchange among all components. Therefore, the idea that there is an abstract common information repository, perfectly serves to represent this requirement. In the figure we represent all the data flows, irrespectively of their nature. In what follows, we provide more detailed views and explanations of each data flow.

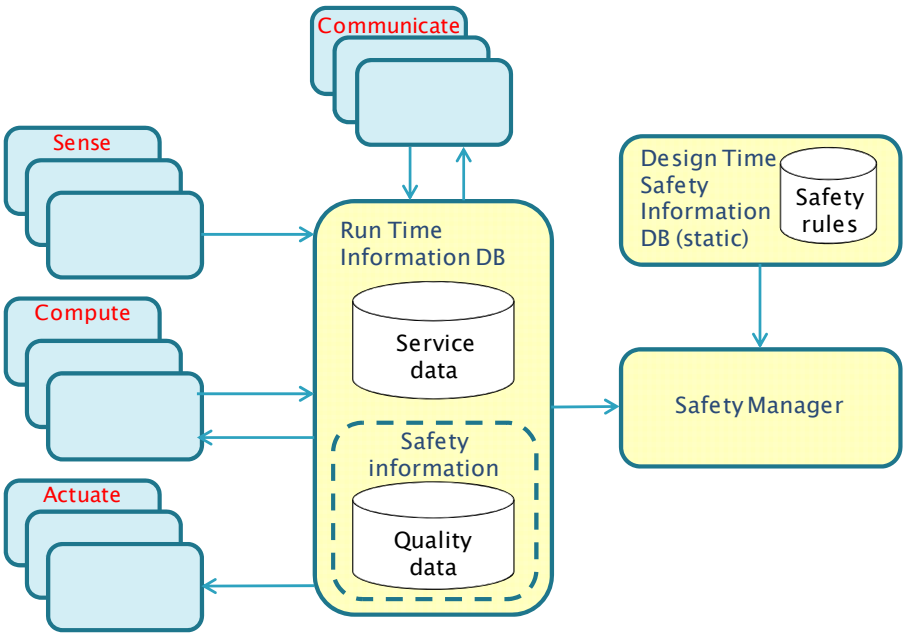


Figure 7: KARYON architecture (data centric view).

The flow of service data is just like the flow of data in a typical control system. This is clearly visible in Figure 8, in which only the service data flow and the relevant components are represented.

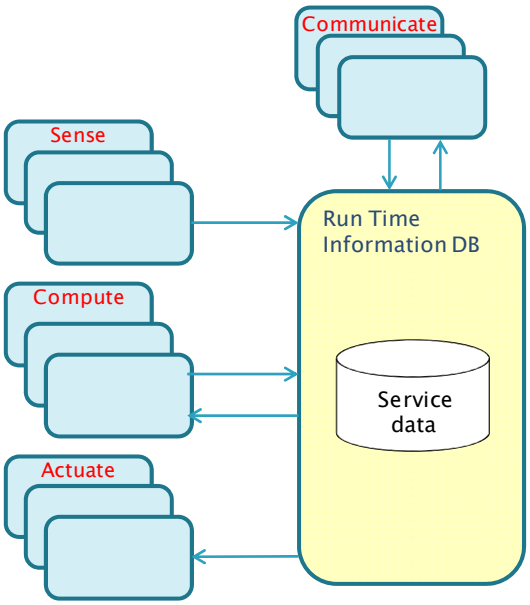


Figure 8: Service data flow.

Data is gathered by sensing components from the environment and by communication components from operational networks. These components then provide the collected information to computing components (through the Run Time Information Database), which process this information and produce new service data that may be either consumed by other computing components, by communication or by actuation components. Communication components send this information through operational networks, while actuation components use the information to actuate on the environment, thus closing the control loop.

Differently from service data, which refers to environment variables, quality information refers to the operational conditions of the nominal control system. This data originates from sensing, computing and communication components, all of them possibly subject to faults, as mentioned

in Section 3.2. This is illustrated in Figure 9, which shows the specific quality data flow through the relevant components.

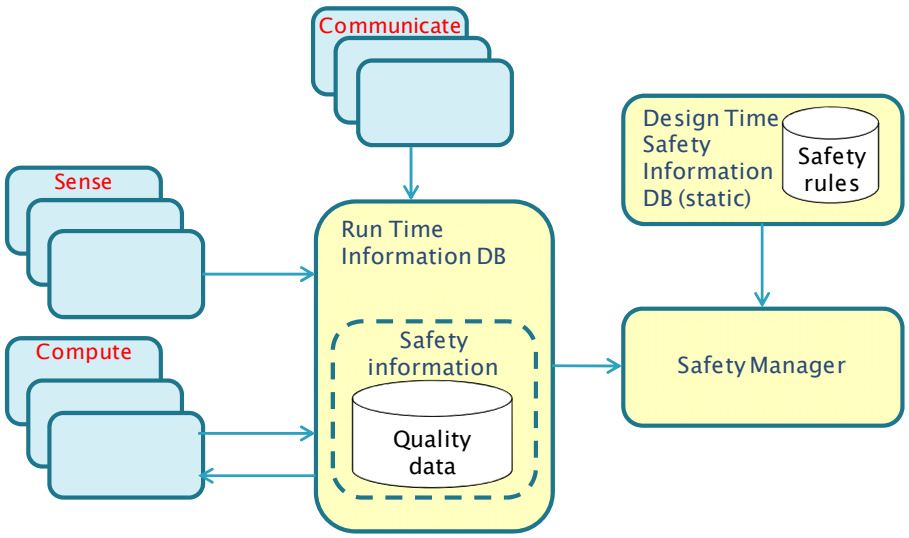


Figure 9: Quality data flow.

In the figure it is possible to observe that computing components can also consume quality information, for instance to consider this information in the algorithm or to propagate this information to the output, possibly modified depending on the operations performed by the component. The main purpose of producing and gathering quality information is to make it available to the Safety Manager, as also represented.

An important issue concerning the quality data is that it should be trustworthy. In other words, it is useless for the Safety Manager to use information about the validity of sensor data or the timeliness of components if this data may be imprecise. The confidence on this quality data must be established at design time, and it must be assured by the implementation. This issue is addressed in detail in deliverable D2.2.

One final observation is that there is no output from the Safety Manager. This is due to the fact that the Safety Manager output cannot be considered service data, nor quality data. The flow of information that gets out of the Safety Manager is a control flow, which is directed to the relevant components that need to be reconfigured. This flow is represented in Figure 5. The concrete mechanisms and solutions to realise adjustments of the LoS, as well as the interfaces between the Safety Kernel and the nominal control system, are developed in task T4.2 and preliminary presented in deliverable D4.2.

4. Implications on services and mechanisms

In this section we identify and we provide a brief discussion of a set of issues that are implied by the KARYON architecture described in Section 3. These issues are addressed in other work packages, namely WP3 and WP4. Additionally, other work in WP2 is concerned with the definition of an abstract sensor model, encapsulating the sensor failure modes, and with the definition of an approach to use environment models in sensor fusion.

Characterization of quality information

Given that we assume that part of the system can be affected by faults (described by considered failure modes), which will be reflected on the quality of data, one issue is that it is necessary to find adequate forms for characterizing and representing the "quality" of data. We have already introduced this issue as a fundamental one, in Section 2.1.2, referring how and where we deal with it in KARYON, but here we elaborate a little more on the issue, already with the knowledge of the proposed KARYON generic architecture.

This sensor data, which is used in algorithms of the nominal control system, is provided by sensors and by communication components. It represents the state of physical variables, like distance, speed, temperature, heading, etc. Therefore, when using a data value representing some of these physical variables, there is an error between this data value the real physical value.

The error is affected by faults, which introduce fixed deviations from the correct value, and is also increasing over time, given that the real physical entity is changing while the value is not updated (thus getting older and, at worse, more inaccurate). This is why it is necessary to continually update the value, to prevent the error to become too large. Under controlled conditions it is possible to make sure that errors are bounded, which allows the design of solutions that may be proven safe for the assumed maximum error. In KARYON we want to keep track of the validity of data, which requires the ability to characterize this error at run time. This requires mechanisms for detecting faults, for fusing data, for being able to compare sensor data with expected data, etc. so that it may be possible to elaborate on the accuracy, or validity of that data. Furthermore, it will be necessary to find a generic way to represent this validity through some quality metrics, which might be easily used along with some algebra to reflect changes in this quality along the processing flow within the system. This is achieved with the abstract sensor model developed in KARYON, which encapsulates the mechanisms that are needed to derive the validity attribute. Sensor fusion solutions also allow dealing with the changes in validity along the processing chain.

It is also important to note that since the quality of data depends on the passage of time, it is important to preserve information regarding the time at which some data may have been collected. Once again, temporal information is also to be taken into account in the scope of the abstract sensor model.

Mapping between quality and integrity

While the aim is to be able to derive the quality of sensor data, assigning some validity attribute to it, reasoning about safety has to be done by considering desired safety integrity levels with respect to the considered hazards. There is an issue of matching the available quality to the needed integrity, which is a fundamental issue that is addressed in KARYON. The problem is made even more complex due to the fact that KARYON considers cooperative functionality, which the existing standards (like ISO 26262) do not explicitly cover.

For the KARYON architectural pattern to be applicable, the issue must be addressed. KARYON is dealing with the issue mainly in the scope of Task 4.1, and the outcomes will be used in conjunction with results from Task 2.2 to define the safety rules that will be stored in the Design Time Safety Information Database.

LoS management timeliness

One important aspect for reasoning about safety concerns the timeliness of the reconfiguration actions.

The timeliness constraints are fundamental as they dictate the maximum amount of time that will take to complete a mode change. If one adds the maximum amount of time that it takes to collect integrity information (that is, the collection period) and check if safety rules are being satisfied, the resulting sum will provide a bound on the maximum amount of time that it will take to accomplish some needed adaptation. This amount of time will have to be taken into consideration when defining the safety rules and will also need to be considered by the designer of the control algorithms in the nominal control system. In fact, given that the functionality needs to be always safe (independently of the LoS), this means that when some failure affects the validity of data the functionality needs to stay safe during transition from one LoS to another LoS. Therefore, it is the functionality designer that has to ensure, knowing the LoS and the time it takes to switch to a new (guaranteed safe) LoS, who has to define the control strategy that will ensure the safety over this time.

From a systems perspective, and reasoning in terms of requirements to the Safety Kernel part of the system (which is responsible for managing the LoS changes), it is necessary that the Safety Kernel architecture allows satisfying the required timeliness constraints. This issue is addressed in detail in Task T4.2.

Actuation safety

As mentioned in the project proposal, the Safety Kernel part of the system "*safeguards the control commands and checks them against the derived set of safety rules*". This must be regarded as a general statement concerning the role of the Safety Kernel, which indeed is in charge of managing the LoS according to the observed integrity of system components and data, so that actuation commands are adequate to the situation and the necessary ones to ensure safety.

However, in real systems, it is not possible to exclude that actuation commands are affected by faults, or even that actuators are faulty. These faults, if not treated adequately, can lead to unsafe situations. But in KARYON, and as stated in Section 3.2, in the fault model we do not consider the possibility that actuators are faulty or that control commands are invalid. This problem can only be addressed with solutions based on redundant actuation physically redundant sensors, which fall out of the scope of the project. KARYON focuses on the problem of faults affecting the data collected from sensors, and timing faults affecting the timeliness of processing components. These are the ones that are intrinsically related to the considered cooperative systems, which rely on external information to achieve the intended performance improvements, as well as on more complex algorithms to process the rich amount of available data.

5. Example applications of the architectural pattern

In this section we jump into a lower level of abstraction, performing a simple but important exercise: we consider specific examples in the avionics and automotive domains, to illustrate the kind of functions that are found in the nominal control system, and to show how they are integrated with the KARYON specific components. In addition, the examples illustrate the application of the LoS concept, which is fundamental to achieve the desired performance improvements (and to allow cooperation to take place), without endangering safety.

5.1 Avionics domain

The example in the avionics domain is the one that will be considered for the purpose of developing a simulation (in the scope of WP5). Therefore, in this deliverable we just provide an outlook of the example, fundamentally discussing how the architectural pattern is applied in this case. We do not delve into details of the simulation itself, or the test cases that will be considered to illustrate the behaviour of the application under fault-free and faulty scenarios, as these are detailed in deliverable D5.1.

In the considered example, the scenarios will include aircrafts and a Remote Piloted Vehicle (RPV), which at some points will be sharing the same airspace. Different manoeuvres can take place, and a fundamental safety requirement is necessarily that the involved airplane and the RPV will be kept within distance to avoid any collision. To ensure that, there will be function in the RPV to collect information from external entities, like other aircrafts, satellites or the Air Traffic Management (ATM). There will be other functions involved in the process of deciding the RPV trajectory and deriving the appropriate control commands, as shown in Figure 10.

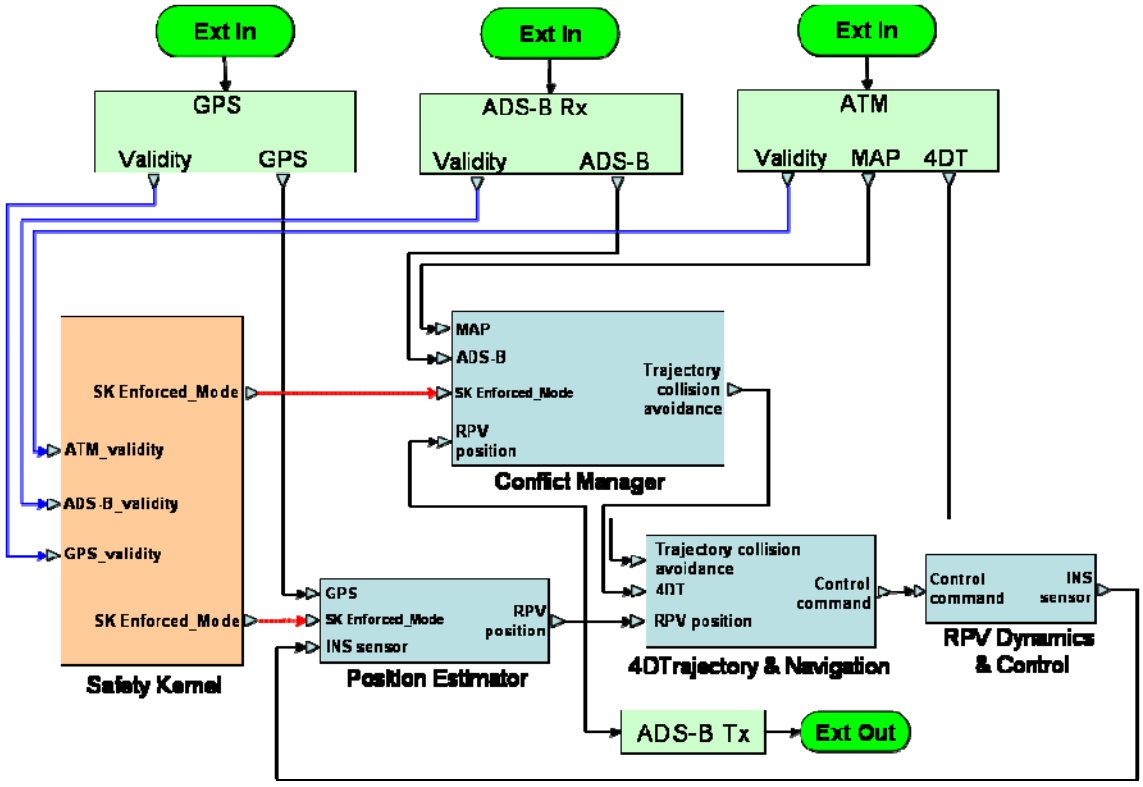


Figure 10: Safety Kernel Integrated with RPV Functionalities.

The components that are part of the nominal control system are the following:

- Communication: GPS, ADS-B Rx, ADS-B Tx and ATM;
- Computing: Conflict Manager, Position Estimator, 4D Trajectory & Navigation
- Sensing and actuation: RPV Dynamics and Control

It must be noted that the GPS and the ATM components could as well be considered sensor components, in the sense that they are only providing input to the system. The ADS-B component, on the other hand, is both used to feed information to the system as well as to consume information (to be sent externally). The RPV Dynamics and Control component represents both an actuator (which drives the dynamics of the RPV) and a sensor, in this case an inertial sensor.

The Safety Kernel is depicted as a single block, abstracting the components of which it is composed.

It is clear from the figure that the control commands eventually sent to actuators will depend on the output of the 4D Trajectory and Navigation component which, on its turn, depends on the output of the Position Estimator and Conflict Manager components.

In this example, none of the components is considered to be above the hybridization line. That is, all the components are considered to be timely, which in a real implementation is something that would have to be proven in design time. Therefore, the only faults that are considered are those that affect the validity of input data. In this case, input data is coming from the communication components, and this data can be affected by faults in the communication. Interestingly, in the figure it becomes clear that these components are modelled as abstract sensors, providing a validity measure on their output. To explain how communication faults can affect the validity of data produced by these components, we provide a simple example. Consider that at some moment it becomes impossible to communicate through the ADS-B component. In this case, given that no information is received, there will be a point in time when this component is supposed to provide information on its output, and it will provide some output with a low associated validity. The actual data that will be sent to the output of the ADS-B components and its validity are not relevant.

Concerning the definition of Levels of Service, it is visible in the figure that two of the computing components have an input named “SK Enforced_Mode”. This input is made available because the two components (Position Estimator and Conflict Manager), can operate in more than one mode, leading to the provision of different LoS.

The role of the Safety Kernel in this example is the following. It collects (periodically) the validity measures from the communication components (which means that all of them might be subject to faults and thus might provide data with varying validity), deciding, based on that and on the safety rules (not explicitly shown, but included within the Safety Kernel block), which mode of operation should be selected for the two computing components.

The LoS management control cycle, and its role in guaranteeing that the functionality will be safe, can thus be explained as follows. We have already seen that a fault in communication will lead to a decrease in the validity of the output of the communication components. This will happen in a bounded amount of time, because the output of these communication components has to be periodically updated. The Safety Kernel also executes periodically, matching the collected validity information against the safety rules defined in design time, to determine the adequate LoS for each functionality. The Safety Kernel will then apply the necessary mode changes corresponding to the LoS that need to be enforced, which have also been defined in design time. The time it takes for the Safety Kernel to perform the needed operations in each cycle will be bounded, which means that the mode changes will also be applied to the respective components in a bounded amount of time. Since the entire system has been designed and proven to be timely, it is known that the mode changes will become effective in due time. Depending

on the new mode of operation, the Conflict Manager and the Position Estimator components will execute different algorithms, possible using different input data. For instance, if a failure leads to a degradation of the positioning information sent by the GPS component, this may lead to a mode change in which the GPS data will not be used, and only the data from the inertial sensor will be used. In this case this certainly corresponds to a lower LoS, which ultimately should lead to an increase of safety distances and possibly a change in the RPV trajectory. In the new mode of operation, the output of the Position Estimator component will be a position that will have a larger associated error, and this is how the trajectory definition may be affected. Clearly, since all components are below the hybridization line, all of these implications will occur in bounded amounts of time. It is thus possible for the designer of the functionality to calculate the safety distance that should be considered as adequate in the highest LoS, knowing that when there is a problem with the GPS the new safety distance will be enforced after a certain well known maximum amount of time.

In summary, this example illustrates how the KARYON architectural pattern is applied in the case of a functionality in the avionics domain, also illustrating the mechanisms and interactions that allow the system to perform safety despite faults affecting the quality of input data.

5.2 Automotive domain

This section presents the application of the KARYON architectural pattern to the automotive domain, materialising the functions that are usually included in the nominal control system. The functionalities for each element are depicted and the compatibility of this architecture with an example cooperative automotive application is shown.

5.2.1 Instantiation of the architectural pattern

The functional view of the KARYON generic architecture is presented in Figure .

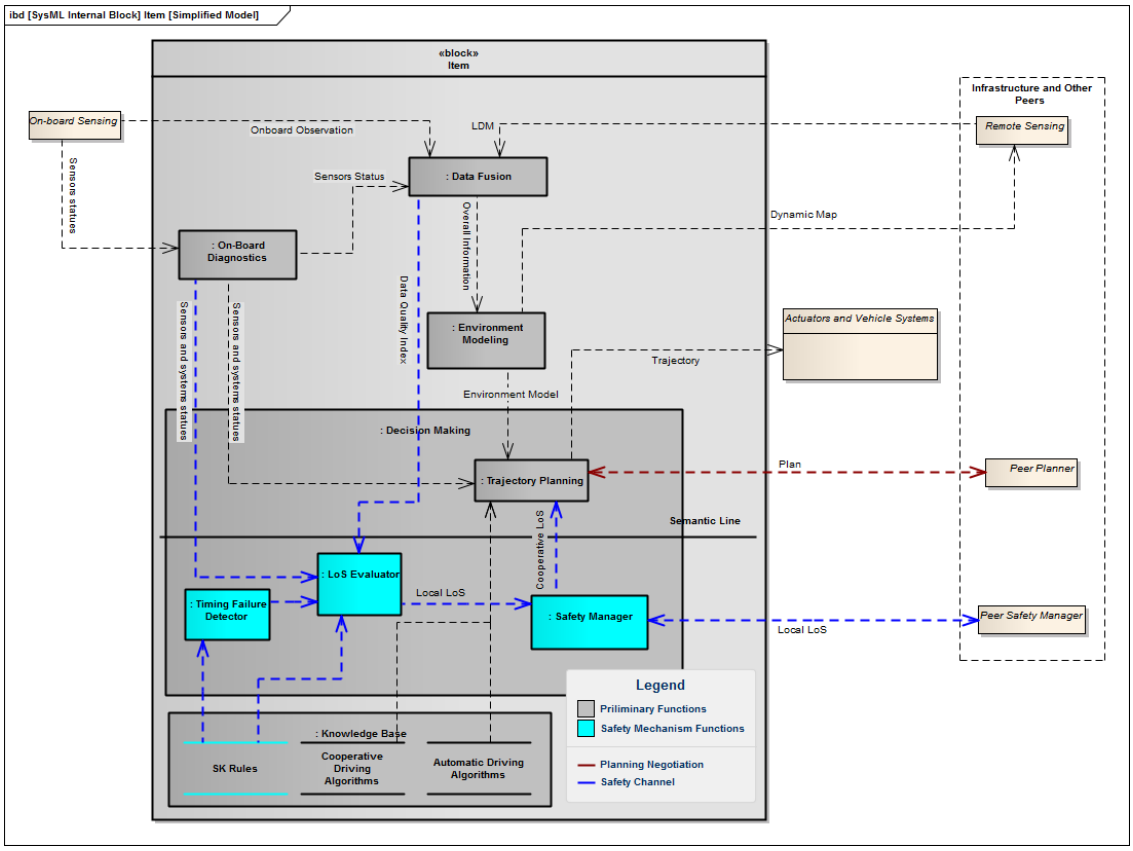


Figure 11: Functional view of the Nominal KARYON Architecture.

The blue functions are those which compose the Safety Kernel. Table 1 provides a brief description for each function illustrated in this figure.

Function	Task
On-board Sensing	The sensors installed on the vehicle to provide information about the position of the vehicle and other objects on the road. GPS, TV camera, LIDAR, inertial sensors and speed sensors are some examples of such sensors.
Remote Sensing	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.
Data Fusion	Combines the information collected from different sources. Simultaneous localization and mapping (SLAM), multi-body SLAM, track-to-track fusion, Bayesian filtering and map matching are some possible techniques to apply.
Environment Modelling	Centralizes the information coming from different components and also facilitates access to the information via views specified to each component in the system.
On-board diagnostics	Detects the faults and malfunctions of on-board devices.
Decision Making	• Determination of the Level of Service according to the data model. • Identification of the applicable driving functions relevant to the specific traffic area. • Dynamically assignment of the functions' priorities according to the context. Neuro-Fuzzy, Evolving Neuro-Fuzzy and tree-based neural fuzzy inference system are some possible techniques to apply.
Knowledge base	Contains the rules and regulations. RDF(-S) and OWL can be used as languages.
Automatic Driving Algorithms	The driving functions and manoeuvres algorithms that can be performed autonomously independent of other vehicles intention.
Cooperative Driving Algorithms	The manoeuvres algorithms which require the information/action regarding other vehicles intention.
Actuators and Vehicles Systems	Actuators which execute the low level commands given by trajectory planner. Steering, traction, and braking systems are some of such actuators.
Trajectory Planning	According to the path and the driving primitive specified in decision making, it plans and adjusts the trajectory to follow.
Peer planner	The other vehicles function which is responsible for negotiating on

	driving primitives and trajectories to be followed.
Timing Failure Detector	Monitors data arrival times such as data received from communication or on-board sensors. It detects the failures according to rules defined in knowledge base.
LoS Evaluator	Evaluates the local Level of Service which belongs to the this vehicle independent of other vehicles. It's just based on on-board systems state and sensors data quality.
Safety Manager	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.
Peer Safety Manager	The other vehicles function which is responsible for negotiating on driving primitives and trajectories to be followed.
SK Rules	Safety Kernel Rules

Table 1: Functions description.

5.2.2 Platooning and Platoon Merging Example

Platooning, as one of the most complex cooperative functions, is a comprehensive example to illustrate the compatibility and consistency of the provided architecture. Figure 11 displays a scenario in which some vehicles are running in a platoon. There is a vehicle (C) intending to join the existing platoon and also another vehicle (D) which is going to merge to this platoon.

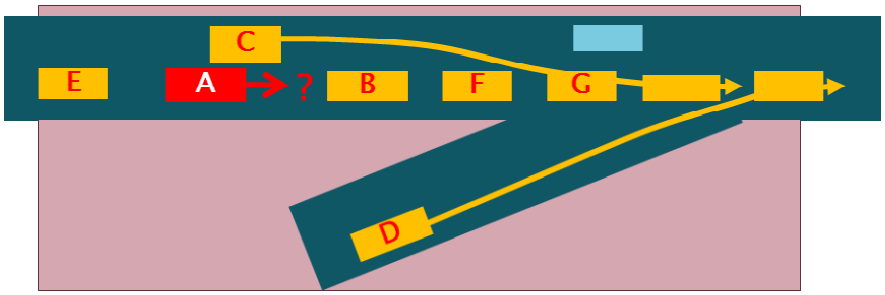


Figure 11: Platooning example scenario.

Should A facilitate the joining manoeuvre of C or the merging of D? Or should it keep the Platooning distance? According to **Error! Reference source not found.**, the decision is left to Decision Making function. But the decision is necessarily determined by the Level of Service, which must be known to let vehicles decide how they can cooperate in a safe way.

Let us focus on vehicle D. Figure 12 illustrates how this vehicle senses the environment in order to obtain the necessary data for performing the intended functions.

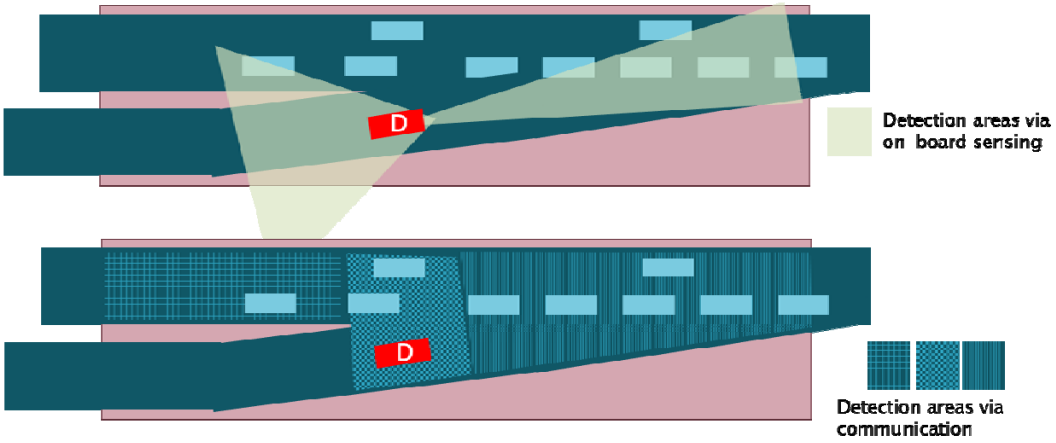
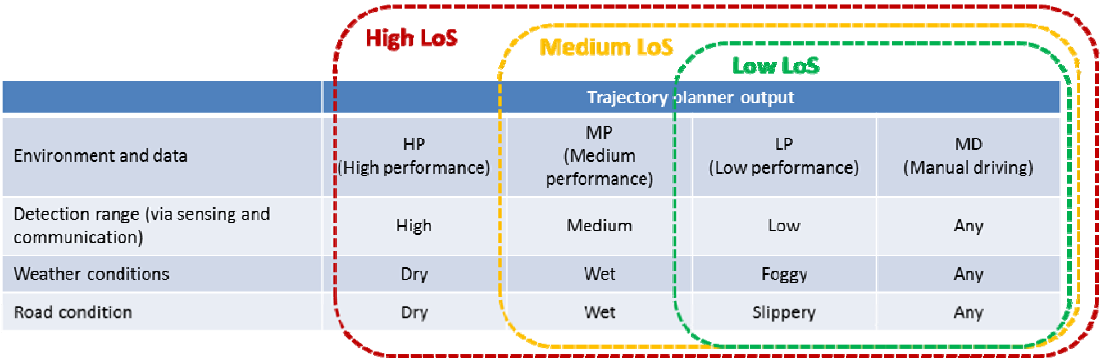


Figure 12: Sensing the Environment and gathering information from other sources.

As it is shown, D is gathering information from different sources. This information is combined by Data Fusion and centralized in Environment Model. In decision making, the sub-function LoS evaluator uses this information to determine the local Level of Service. Figure 13 provides a simple scheme showing how the appropriate LoS is chosen.



	Trajectory planner output			
Environment and data	HP (High performance)	MP (Medium performance)	LP (Low performance)	MD (Manual driving)
Detection range (via sensing and communication)	High	Medium	Low	Any
Weather conditions	Dry	Wet	Foggy	Any
Road condition	Dry	Wet	Slippery	Any

Figure 13: Criteria for LoS Determination.

Having the local LoS, the negotiation on cooperative LoS is performed so that the cooperative LoS is determined by the Safety Manager. Then the Trajectory planner starts a new round of negotiations with other's Peer Planner based on cooperative LoS. Let's consider the following table as assumed priorities by this moment:

Primitive	A	B	C	D	E
Platooning	3	1	N.A.	N.A.	1
Platoon joining	2	2	2	N.A.	2
Platoon merging	1	3	1	1	3

Table 2: Primitives priorities for each vehicle.

Regarding this table the vehicles A, B, C, D and E are involved in three cooperative driving primitives:

- vehicles A and C should let D merge the platoon: A will prepare some space for D, and C will not occupy that space.

- vehicle B maintains the platooning distance.
- vehicle E keeps platooning.

Therefore the merging primitive must be applied as displayed in Figure 14.

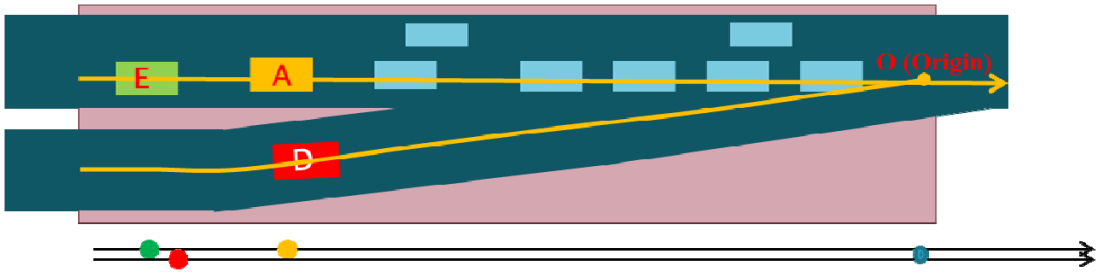


Figure 14: Platoon Merging Primitive.

This figure shows the vehicles A and E Platooning when D is going to merge. A is going to facilitate the merging of D and the trajectory Planner is responsible to provide the appropriate trajectory. Figure illustrates the resulting trajectory.

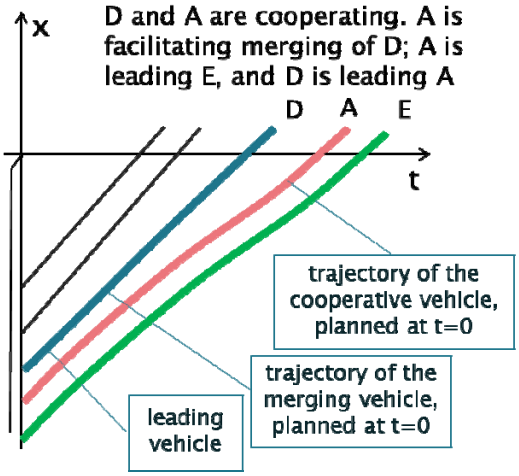


Figure 16: Trajectory planning.

As result vehicle A slows down to facilitate merging of D. Vehicle E slows down too to keep the safe distance in the platoon following A, and D drives with constant speed to merge into the platoon. Finally, these objectives are provided as low level commands to actuators to follow the trajectory.

6. Conclusion

This document provides the generic KARYON architecture and describes in detail the fundamental concepts underlying the architectural decisions. From the deliverable it should be clear how the project aims at structuring a cooperative system so that it might be possible to add more complex algorithms, to use wireless networks for collecting remote information and to rely on an arbitrary range of sensor, without compromising functional safety.

The proposed architectural pattern will be applied in de definition of the proof-of-concept demonstrations. Some examples of how the pattern is reflected in concrete use cases are presented in the present document.

The architectural solutions presented in this deliverable have to be understood in connection to other solutions developed in other work packages and presented in other deliverables, namely in D2.2, D2.4 and D4.2.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki, *Membership algorithms for multicast communication groups*. In 6th International Workshop on Distributed Algorithms (WDAG), pp. 292-312. Springer Verlag, November 1992.

[2] A. Avizienis, *The methodology of n-version programming*. In: Lyu, M.R. (ed.) Software

Fault Tolerance, chap. 2, pp. 23–46. John Wiley & Sons, New York, 1995.

[3] A. Bartoli. *Group-based multicast and dynamic membership in wireless networks with incomplete spatial coverage*. Mob. Netw. Appl. Vol 3, number 2, pp:175-188, August 1998.

[4] K. Birman, and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[5] A. Casimiro, P. Martins and P. Veríssimo. *How to Build a Timely Computing Base using Real-Time Linux*. Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems (WFCS'00), pages 127–134. Porto, Portugal, September 2000.

[6] A. Casimiro, J. Kaiser and P. Veríssimo, *Generic-Event Architecture: Integrating Real-World Aspects in Event-Based Systems*, Lecture Notes in Computer Science (Architecting Dependable Systems IV), vol. 4615, pp. 287-315.

[7] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, *On the impossibility of group membership*. In 15th ACM Symposium on Principles of Distributed Computing (PODC), pp. 322-330, May 1996.

[8] M. Correia, P. Veríssimo and N. F. Neves. *The design of a COTS real-time distributed security kernel*. In Proceedings of the Fourth European Dependable Computing Conference, pages 234–252, October 2002.

[9] F. Cristian and C. Fetzer. *The Timed Asynchronous Distributed System Model*. IEEE Trans. Parallel Distributed Systems, 10(6):642–657, June 1999.

[10] C. Dwork, N. Lynch, and L. Stockmeyer. *Consensus in the presence of partial synchrony*. Journal of the ACM, 35(2):288–323, April 1988.

[11] M.J. Fischer, N.A. Lynch and M.S. Paterson. *Impossibility of Distributed Consensus with One Faulty Process*. JACM, 32(2):374–382, 1985.

[12] ISO 26262, Road vehicles – Functional safety, Part 1-9, 2011.

[13] L. Lamport, R. Shostak and M. Pease. *The Byzantine Generals Problem*. ACM Transactions on Programming Languages and Systems, 4(3), 382–401. 1982.

[14] L. Lamport and N. Lynch. *Distributed computing: Models and methods*. Handbook of Theoretical Computer Science, vol.B: Formal Models and Semantics. J. Van Leeuwen (Ed.), pages 1158–1199. Elsevier Science Publishers. 1990.

[15] N. F. Neves, M. Correia and P. Veríssimo. *Solving vector consensus with a wormhole*. IEEE Transactions on Parallel and Distributed Systems, 16(12):1120–1131, December 2005.

[16] H. Ortiz, A. Casimiro and P. Veríssimo. *Architecture and Implementation of an Embedded Wormhole*. In Proceedings of the 2007 Symposium on Industrial Embedded Systems (SIES'07), pages 341–344. Lisbon, Portugal, July 2007.

[17] D. Powell. *Failure mode assumptions and assumption coverage*. In Proceedings of the 22nd IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS-22), pages 386–395, Boston, USA, July 1992.

[18] B. Randel, J. Xu, *The evolution of the recovery block concept*. In: Lyu, M.R. (ed.) Software Fault Tolerance, chap. 1, pp. 1–22. John Wiley & Sons, New York, 1995.

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

[20] Trusted Computing Group, TPM Main, Part 1 Design Principles. Specification Version 1.2, Revision 62, 2003.

[21] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[22] P. Veríssimo and A. Casimiro. *The Timely Computing Base model and architecture*. IEEE Transactions on Computers, 51(8):916–930, 2002.

[23] P. Veríssimo. *Uncertainty and predictability: Can they be reconciled?* In Future Directions in Distributed Computing, volume 2584 of Lecture Notes in Computer Science, pages 108–113. Springer-Verlag, 2003.

[24] P. Veríssimo. *Travelling through wormholes: a new look at distributed systems models*. SIGACTN: SIGACT News (ACM Special Interest Group on Algorithms and Computation Theory), vol.37, no.1, pages 66–81, 2006.