Safety Kernel for Cooperative Sensor-Based Systems

Pedro Nóbrega da Costa, João Pedro Craveiro, António Casimiro, and José Rufino

Universidade de Lisboa, Faculdade de Ciências, LaSIGE,
{pncosta,jcraveiro}@lasige.di.fc.ul.pt, {casim,ruf}@di.fc.ul.pt

Abstract. Developing smart vehicles, either automobile or aerial, to realise cooperative functionality in open and inherently uncertain environments is a difficult task. One fundamental challenge is to make cooperation predictable and safe, despite the uncertainties affecting the operation. Traditional approaches for the design of safe control systems rely on the possibility of defining safe operational bounds, both in the value and in the temporal domain. Unfortunately, when considering wireless communication networks and varying sources of sensor data, it becomes very hard, or even impossible, to define safe and small enough bounds. To deal with this problem, a possible approach is to consider a hybrid system architecture in which some components may execute with uncertain timeliness, but which also includes some predictable components. In addition, a Safety Kernel implemented in the predictable part of the system will be instrumental to manage the system behaviour and ensure safety.
In this paper, we describe the architecture and role of such Safety Kernel in the context of a hybrid system architecture. The Safety Kernel is responsible for monitoring and managing the run time configuration of the system, as needed to avoid hazardous situations. We specify the individual components of the Safety Kernel and how they interact with other components in the system architecture, including the functional components of the control system. Finally we present a high-level description of a concrete implementation based on time and space partitioning.

Keywords: Architectural hybridization, Autonomous vehicles, Cooperation, Safety, Time and Space Partitioning

1 Introduction

Emerging technological improvements in the domains of embedded computing, sensing and actuation and wireless communication are key enabling factors for the emergence of new applications of smart vehicles autonomously cooperating and interacting to provide improved *functionality*. In the automotive domain, cooperation can be important for achieving increased traffic throughput and energy saving. In the avionics domain, Remotely Piloted Vehicles (RPVs) might be able to autonomously cooperate with other airplanes in the vicinity in order to manage safety distances, making it possible to deploy these systems in shared airspace areas.

All these applications have very strict safety requirements, which have to be handled in design time during the several stages of the system development. Typically, safety requirements call for predictability, that is, the need to make assumptions on worst case values for critical parameters, upon which the system solutions are developed. However,

when considering cooperative systems that interact over inherently uncertain wireless networks, it becomes infeasible or too inefficient to consider upper bounds, for instance on communication delays, which prevents cooperation to be implemented while preserving safety. Moreover, also the control algorithms tend to be more complex as the diversity and richness of available sensory information increases. This also makes it more difficult to ensure temporal and logical correctness of system components, which is problematic when safety is a fundamental attribute. Ideally, we would like to be able to exploit the benefits brought by cooperation, in terms of more precise control with smaller safety margins, without making any concessions on safety, or even improving safety. This is a challenging objective, on which the KARYON project is focused.

More precisely, the KARYON project aims at providing system solutions for predictable and safe coordination of smart vehicles that autonomously cooperate and interact in an open and inherently uncertain environment. Addressing this challenge requires innovative solutions to guarantee safety by dynamically managing the system's performance. On the one hand, performance improvements (e.g., smaller safety distances) can be achieved with a complex control system and cooperation. On the other hand, it is necessary to constrain the system operation whenever this complex control system experiences faults. Such faults, not excluded a priori in design time, may lead to hazardous situations if not handled conveniently. For instance, considering an example with RPVs flying in a group, a fault affecting the operation of a complex component can be handled by increasing the safety distance, at the expense of a reduced performance (due to the need of a larger portion of air space for the set of RPVs). This problem has been recognized many years ago [13, 14] and approaches to address it have been described under the perspective of using hybrid architectures. In [14] the Timely Computing Base is presented, which separates the system in two parts, a payload and a control part, where the former can be of arbitrary complexity and synchrony, while the latter has to be small, predictable and synchronous. The payload part relies on the control part for the execution of critical steps or fault handlers, which allows complex components to be used to achieve overall better performances with the available resources, while making sure that safety concerns can be addressed by the possibility of defining helper and fall back procedures that are timely and correctly executed. In [13], the author also proposes to structure the system in terms of complex control functions and simple control functions, where the latter are executed whenever the former start to behave incorrectly or untimely. He argues that the key issue is to use simplicity to control complexity, which is better than using other fault-tolerance approaches like N-version programming [4] and recovery blocks [11], when considering that resources are limited.

In this paper, we describe the Safety Kernel approach employed in the KARYON project, backed up by a brief description of the surrounding system architecture, which explores the concept of architectural hybridization introduced in [16], and is aligned with the hybrid system models proposed in [14] and [13]. A fundamental issue in architecturally hybrid system concerns the approach that is used to manage the existing redundancy, that is, how to allocate functions to different parts of the system, how to determine when some function is not performing as well as needed and how to select which functions are executed and when they are executed. The Safety Kernel is the part of the KARYON architecture that is in charge of some of these management func-

tions, which are defined to be consistent with the KARYON approach for dealing with functional safety requirements of cooperative vehicles. A KARYON system relies on the Safety Kernel to deal with the uncertainties potentially affecting some functional components, to ensure that the provided cooperative functionalities are safe. The Safety Kernel is in charge of monitoring and managing the operational configuration of the control system components providing the functionality, for which it continuously verifies if a set of predefined safety rules on the timeliness and sensor data validity are satisfied.

The paper is organized as follows. After surveying related work (Sect. 2), we briefly describe the overall system architecture and how it address the problematic of safe cooperative functionality (Sect. 3). Sect. 4 describes the Safety Kernel architecture, namely its components, and how they interact between them and with the rest of the system. Sect. 5 gives a description of a concrete implementation solution. Sect. 6 closes the paper with concluding remarks and future work directions.

2 Related Work

State-of-the-art critical systems are typically built considering models in which assumed properties (e.g., synchrony, faults) are applied to the whole system and do not change over time. Therefore, these models are said to be homogeneous. On the contrary, we advocate that in order achieve performance improvements without sacrificing safety it is necessary to consider *hybrid distributed system models* [15]. These allow to better capture the real properties of the environments in which vehicles operate and in which functionality is implemented. More than that, we believe that *architectural hybridization* [16] is the natural way to architect systems in accordance to the considered hybrid system models. One simple example of a system well described by a hybrid system model is a system with a watchdog. The watchdog is used as a safeguard, to make sure that if something goes wrong in the system then it will be possible to, at least, make the system stop in order to prevent some wrong or unsafe behaviour. Clearly, while the system is assumed to possibly fail, the watchdog is assumed to always operate correctly. Therefore, the watchdog is a subsystem with better properties than the rest of the system, which is possible because it is a simple component.

Mixed criticality [5] is the concept of allowing applications with different levels of criticality to coexist on the same system. In this case, one may want that the properties and assumptions that hold for one application be different from the ones that hold for another application, which is not easily achieved in a system based on a homogeneous model. Mixed criticality models show affinity with hybrid system models, in which assumptions and properties may vary on different parts of the system or may hold only for a period of time. The GENESYS project [10] acknowledged the hybrid nature of systems and developed a component-based generic platform for embedded real-time system. However, GENESYS is significantly focused on the problems related to composition and component interfaces, whereas our interest is on understanding how uncertainty can be characterized and how the performance can be managed while making sure that safety requirements are always satisfied.

The recovery block concept [9] follows an hybrid model, where multiple versions for the same function are developed. First it runs the more complex version of the function (with extra features and more prone to errors). If an error is detected, then a simpler implementation is executed. Simplex [13] follows a similar approach by defining an architecture composed of two system controllers. One simple and proven safe and one with additional features but unreliable. It tolerates faults in the unreliable controller using a decision module that observes the plant to verify if the the controller is being able to keep the controlled system within the desired operational envelope. If not, it switches the execution to the reliable controller, trading off performance for safety. The solution is thus designed by assuming that faults are ultimately reflected on some undesired external behaviour, which can be reliably observed through the existing sensors. In KARYON we look to the problem differently, because we consider that sensor data may not always be valid due to faults affecting sensor, or due to uncertainties affecting the timeliness of communication and hence the promptness (and validity) of the other sensor data received from remote vehicles. Therefore, we define an abstract sensor model that allows the validity of sensor data to be estimated, and we consider that some components may do timing failures due to their complexity. Given that, the solutions for deciding when to change the control algorithm, or when to perform some system reconfiguration, are done in a different way than it is done in Simplex.

The coexistence of reliable and unreliable components calls for mechanisms for fault containment. Virtualization [7] has been widely used as a mechanism to run multiple systems within the same physical computing platform, allowing to provide different environments in each virtual machine and isolation between them. However, most virtualization solutions do not provide strict temporal isolation. One approach to achieve mixed criticality without increased certification expense and providing a complete fault containment (including temporal isolation) between components is to use *time and space partitioning* (TSP). TSP is a concept for safety-critical systems in which applications with different criticality levels and different requirements may coexist in the same execution platform. TSP separates the system's software components into logical containers called partitions, ensuring that faults occurring in one partition do not affect other partitions, with respect to both time and space domains. These two properties ensure that faults are contained to their domain of occurrence, preventing them from propagating to other partitions.

A prominent example of TSP system design is the adoption of the ARINC 653 [1] specification by the civil aviation domain. In the automotive industry, the top-level requirements for an AUTOSAR operating system include provisions that correspond, to some extent, to the notions of temporal and spatial isolation [2]. The specification of the AUTOSAR operating system, however, does not prescribe the use of strict partitioned scheduling as a means to achieve this temporal isolation among applications [3]. We take advantage of TSP properties to develop a solution that integrates in the same platform components of different complexity, some that are proven timely and reliable in design time, and other that may behave in uncertain ways. The latter can be used to implement improved functions, exploiting the additional information made available through cooperation, without compromising safety. The overall approach can still be viewed as sufficiently modular to be adopted by existing legacy systems.

3 System architecture for safe cooperation

Cooperation and information sharing is one way to increase the performance and efficiency of one set of neighbour vehicles. However, when dealing with safety critical systems, cooperation comes with a major challenge: the need of dealing with uncertainty and, at the same time, guarantee the system's safety. This section describes how to integrate and manage unpredictable components in a safety critical system, and how it is reflected in the system's architecture.

3.1 Levels of Service

We consider one system (a vehicle) as a set of multiple components, such as sensors, actuators and computing elements, in charge of acquiring information, processing it, and outputting a result.

Safety is guaranteed by managing in runtime the performance provided by each function in the system. For that, we consider that functions may be performed in different ways, each way leading to different results in qualitative terms, but possibly requiring different resources and implying different degrees of timeliness. One approach to achieve multiple performance levels is to have a component able to adapt its own behaviour based on some given external parameter. The other approach is to have more than one component, each implementing the function in a different way, executed in parallel and producing different possible outputs, from which one is chosen to be used. In this case, at least one of the implementations must be proven, at design time, to provide the necessary performance to satisfy safety requirements.

We define Level of Service (LoS) of a cooperative functionality as the combined performance achieved by the set of components executing in the system. Because we consider cooperative functionality, it is expected that cooperating entities will be aware of this LoS, and that each of these entities will eventually perform the cooperative functionality with that LoS. At least, each entity must be aware of the LoS with which each other entity is capable of performing the cooperative functionality. On each LoS the functionality is realised with a different performance and requires a different set of assumptions to ensure safety. For instance, to achieve higher performance the required safety integrity levels are also higher and, as such, the set of safety requirements that must be met at runtime is more stringent for this LoS than for other LoS corresponding to a lower performance. In runtime, it must be possible to select the LoS that provides the highest performance while making sure that all unacceptable risks are avoided and that, based on runtime constraints, safety is achieved.

3.2 Hybrid Architecture

We exploit the advantages of architectural hybridization with the purpose of allowing the integration of functions with different criticality levels and different requirements in the same system. This gives us the possibility to include not only real-time components proven timely in design time, but also components that, due to their complexity, cannot

to be proven timely (with reasonably small bounds). In practice, this allows us to include a bigger variety of components, namely those that would have to be discarded in approaches requiring all components to be proven timely in design time.

The two types of components (i.e. the ones that are proven to be timely safe and the ones that are not), are logically divided by a so-called hybridization line. Predictable components are said to be under the hybridization line, while components exhibiting uncertain behaviour are said to be above it, defining two different realms that enjoy different properties and provide different guarantees. Since the realm above the hybridization line is a source of possible uncertainty, it must be guaranteed that components below the hybridization line can maintain the system in a safe state (i.e., realise the functionality in a safe way), despite the possible misbehaviour of the ones above the line. We further consider another separation within the predictable part of the system. This separation is between the components that provide some sort of function that is necessary for the overall functionality (and are thus aware of the application semantics), and components that are fully unaware of the application semantics, only providing services related to the management of safety and in charge of switching the system and component behaviour according to the required LoS. These components constitute the Safety Kernel, which we consider to be separated from the other components by a so-called semantics line (see Fig. 1).

3.3 Safety Kernel

To ensure that safety requirements are always satisfied, the LoS has to be adjusted in runtime to a level in which all safety constraints are met given the observed integrity of the system components and sensor data. This adjustment is done by the Safety Kernel, which encompasses a set of components in charge of monitoring the execution of the functional components and controlling the LoS in which the system operates. Therefore, the Safety Kernel components must be reliable, proven in design time to behave correctly and timely, similarly to other components below the hybridization line.

We envision a generic Safety Kernel, providing services that are independent of the specifically considered functionality. As mentioned above, the Safety Kernel is unaware of any application semantics. This makes it modular, facilitates its use and integration in existing systems, and its development and validation process. Still, the Safety Kernel must know about safety constraints that need to be satisfied in each LoS. These safety constraints (defined at design time) take into account all the functionalities to be provided, their interdependencies, and other issues that are important when making a safety analysis, and are stored in a Safety Rules Database (part of the Safety Kernel).

4 Safety Kernel architecture

To achieve a Safety Kernel that is reliable, simple enough to be proven safe at design time and always behaving in a correct and timely way at runtime, its function is centred around a simple rule checker.

Each component in the system whose output has to be monitored commits to produce information regarding the validity of its output, which we call validity data. A set

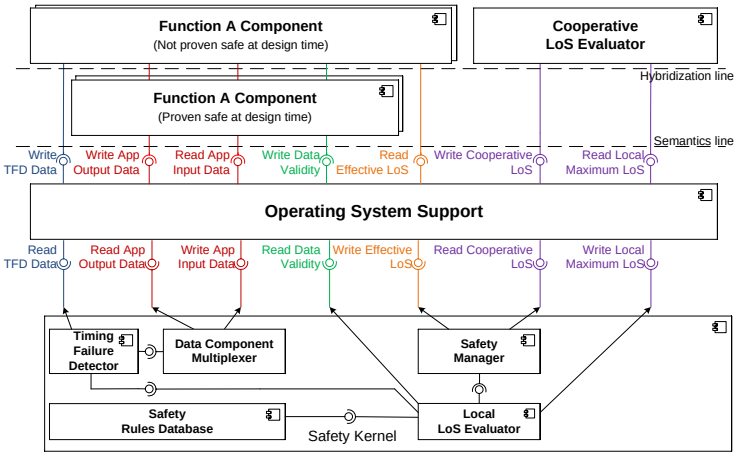

Fig. 1. Components and interfaces

of predefined safety rules establish bounds to the validity data produced by components, allowing to assert whether that validity data fulfils the current safety requirements.

The evaluation of the validity data against these rules defines the LoS in which the system is able to perform each cooperative functionality. As such, the main task of the Safety Kernel is to periodically read the validity data produced by functional components, compare it with a set of predefined rules to determine the suitable combination of LoS for the managed functionalities, and make the necessary adjustments on the operation of system components to bring one or more functionalities into their new LoS.

The Safety Kernel must as well act as a timing failure detector, looking up for timing failures in components above the hybridization line.

In Fig. 1 we present a schematic view of the system architecture in the form of a Unified Modelling Language (UML) component model, with the Safety Kernel and the components which it may use for its operation.

4.1 Relation of the Safety Kernel with the overall architecture

As mentioned previously, the Safety Kernel is positioned below both the hybridization line and the semantics line. We assume the existence of another set of semantics-independent functionalities corresponding to functionalities provided by an operating system. The remaining components we consider are located in the other two architectural levels above the semantics line. The task of the Safety Kernel is to control these remaining components, based on the validity data and on timeliness observations, and defining the performance level of each component (in case they can perform in different ways) as to meet a certain LoS. This requires the Safety Kernel to perform multiple tasks. For the components located above the hybridization line, and because they are not proven to be timely at design time, their timeliness must be continuously monitored. For functions that are implemented using multiple components (and in which each compo-

nent produces one possible output to be used by other components), the Safety Kernel must choose which of the produced outputs will be forwarded to other components.

The control of the LoS is done not only based on the safety evaluation of the local components, but also on the evaluation made by the cooperative neighbour nodes. Based on this, the Safety Kernel defines the LoS to be applied to each of the functions of the system.

4.2 Operating system support

Communication between functions and the Safety Kernel is handled by some sort of Operating System (OS) support. As such, the interfaces required by the Safety Kernel shall be provided by the OS, which ensures that the primitives composing these interfaces are provided in READ–WRITE pairs constituting an information flow channel with one writer and one or more readers.

In each pair of READ–WRITE primitives, one of the primitives is intended to be used by one of the Safety Kernel components, whereas the other one shall be used by a component external to the Safety Kernel, as pictured in Fig.1. Both types of primitive should be non-blocking, atomic, and the OS support should provide the following guarantees: **(i)** the value that is returned by a READ call is the one written in the last invocation of the corresponding WRITE call (until overwritten, a value can be read multiple times and/or by multiple readers); and **(ii)** the value provided in a WRITE call overwrites the value previously provided by the same writer. The way these information flow channels are implemented is abstracted by the OS support, and should be transparent to the remaining components. It is the responsibility of the OS support to ensure, by whichever means necessary, that READs are consistent with the latest WRITE. The OS support must also provide scheduling mechanisms which allow temporal predictability of the interaction flows we here describe.

4.3 Safety Kernel components

To perform its role, the Safety Kernel exchanges information with the remaining components of the system. The exchanges with different types of component embody different aspects of the Safety Kernel operation. For this reason, we see the Safety Kernel as a set of components, with clearly defined and separated concerns, combined to verify and guarantee the operational conditions for safety.

Safety Rules Database. The Safety Rules Database contains the rules used to evaluate, at runtime, the system safety and to decide under which LoS the system may operate. The system safety is evaluated by assessing the data validity information with respect to the bounds expressed in the stored rules. Each LoS has its own set of requirements and therefore its own set of rules that must be checked to realise whether the system is able or not to perform in that LoS. The complexity of the rules can vary from a collection of independent checks of data validity to a sequence of interdependent checks of data validity. Based on safety analysis, these rules are generated and verified by application developers and merged at integration time in the Safety Kernel, which is independent

from their generation, validation and semantics, which allows to have a generic Safety Kernel, independent from the applications.

Component Data Multiplexers. As explained in Sect. 3.1, functions can be implemented by more than one component executing simultaneously: at least one must be predictable and some may execute with uncertain timeliness. The output by the reliable component can always be used, whereas the remaining outputs from other components may have low validity or may be be produced too late. The task of the Data Component Multiplexers is to decide which of the outputs will be forwarded to components that use these outputs as inputs, discarding the others. To do so, each component sends its output value to the Data Multiplexer who, in its turn, reads it and forwards the one that corresponds to the determined LoS to components.

Communication between components and the Data Multiplexer is realised by the *Data Multiplexer Interface*, that provides the primitives for both reading and writing the outputs and inputs of components (see Fig. 1), abstracting the process both to components who provide output and the ones who seek input. In practice, this allows to mask timing failures produced by unpredictable components, since the result of another component (e.g., the safe one) can be forwarded as soon as the timing failure is detected.

Timing Failure Detector. The Timing Failure Detector (TFD) component is in charge of detecting failures in the time domain in components above the hybridization line. Hence, this component acts as a watchdog, detecting delays and crashes. Using the primitives provided by the *TFD Data Inteface*, components above the hybridization line must periodically send a heartbeat to the TFD. When executed, the TFD must, for all the functions, check whether their heartbeats are still valid or not (i. e., recently produced). Upon a detected delay or crash on some function, this will be considered when evaluation safety rules and will lead to a LoS change and, consequently, to the selection of a different component output by the Component Data Multiplexer.

Local LoS Evaluator. The role of the Local LoS Evaluator is to, periodically, evaluate and check the validity data sent by each function against the Safety Rules Database. This interaction is supported by the primitives provided by the *Data Validity Interface*. Based on data validity analysis, the Local LoS Evaluator determines the maximum LoS at which the local functions are able to safely perform. The Local LoS Evaluator can, from this information, determine the maximum cooperative LoS at which the node can participate in each cooperative functionality. This result is then made available; while it can be read and used by any part of the system, the maximum cooperative LoS is meant to be used by the Cooperative LoS Evaluator, external to the Safety Kernel (Sect. 4.4).

Safety Manager. The Safety Manager is the component in charge of controlling the operation of the system and that ultimately enforces one LoS to be effectively adopted by system components. Periodically, using the *Cooperative LoS Interface*, the Safety Manager reads the valid agreed cooperative LoS, if available, as well as the Local LoS

calculated by the Local LoS evaluator. The effective LoS is chosen by the Safety Manager as the lowest LoS between the Local and the Cooperative LoS: $\mathit{EffectiveLoS} = \min(\mathit{LocalLoS}, \mathit{CooperativeLoS})$.

Using the *Write_Enforced_LoS* primitive, the Safety Manager outputs the effective LoS to all the functions that should reconfigure themselves in accordance to it. This LoS is also made available to the Cooperative LoS Evaluator (presented in Sect. 4.4). The reconfiguration and adjustment mechanisms are responsibility of each function, and are out of the scope of the Safety Manager.

4.4 Cooperative LoS Evaluator

For each cooperative function, the Cooperative LoS Evaluator has the purpose of participating in an agreement on the LoS. This agreement is performed with the remaining participating nodes, so as to take into account the maximum cooperative LoS each of them can achieve. Although the Cooperative LoS Evaluator plays an important role towards safety, it cannot be part of the Safety Kernel mainly due to the uncertainty in the communication with other nodes. Therefore, it is located above the hybridization line, outside the Safety Kernel. This component must maintain knowledge about which nodes are participating in the cooperative functionality (group membership).

The output of the Cooperative LoS Evaluator is the agreed cooperative LoS, made available to the Safety Manager using the *Cooperative LoS Interface*. If this component is not able to reach an agreement with the nodes included in the group membership it will not produce any output. The Safety Manager shall be prepared to deal with this situation, by establishing an LoS which guarantees safe co-existence with the remaining participating nodes in spite of the impossibility to cooperate.

5 Implementation

Based on the requirements described previously, this chapter details a preliminary implementation of a system based on the TSP concept. Finally, we present a practical example of implementation using a TSP architecture, detailing how it will support scheduling and communication between the system components.

5.1 Time and Space Partitioning

TSP separates software components into logical containers (partitions), ensuring fault containment between them (i.e., faults that occur in one partition do not affect other partitions), with respect to both time and space. Partitions are managed by an underlying layer responsible for enforcing TSP properties, partition scheduling and dispatching, memory protection, and communication between partitions.

We take advantage of the properties provided by TSP to contain faults between components above and below the hybridization line. By isolating unpredictable components, we can ensure that failures and delays that may occur in these components do not affect the remaining.

5.2 AIR Implementation

AIR [12] is a TSP architecture implementation using an intermediary layer called Partition Management Kernel (PMK) to ensure TSP properties. Although inspired in the ARINC 653 specification, AIR aims to improve upon such specification. More specifically, it deliberately diverts from ARINC 653 where the latters limitations can be overcome to the benefit of additional functionality and flexibility without compromising safety. For instance, despite the strict prohibition thereto in ARINC 653, the architecture is being improved to safely schedule applications over multiple processor cores [6].

Partitioning. In order to prevent errors from propagating from one component to another, these must be isolated in partitions. One partition may, however, host more than one component. The division in partitions is only mandatory between components on which we want to ensure fault containment by enforcing temporal and spatial segregation. In an extreme case, all components under the hybridization line may be in one partition, and the ones above the line in another partition. In an opposite example, every component may have its own partition.

Partitions are scheduled by the PMK according to a fixed schedule defined at design time, bounding the time assigned to each partition and ensuring that timing faults do not propagate from one partition to another. This schedule repeats itself over a time period called Major Time Frame (MTF). This ensures that scheduling is completely predictable, and that the time assigned to each partition is known and bounded — so that delays in one partition do not affect the others.

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

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

6 Conclusion

In this paper we addressed the problematic of trading off the degree to which we fulfil the goals of a cooperation between vehicles (the performance of the cooperative functionality) for increased overall safety. The Safety Kernel is the component in charge of

monitoring and controlling the execution of system's components, ensuring they adapt to runtime constraints. Due to the uncertainty present in some components, we suggested an implementation based on the TSP concept employed in the civil aviation and aerospace industries, which provides fault containment between components. By isolating non-safe components, we ensure that failures and delays that possibly occur in these components do not affect any other ones.

As future work, we envision implementing the described Safety Kernel solution within the scope of the considered system architecture, demonstrating its effectiveness and innovations via computer simulations with fault injection support to experimentally evaluate safety assurance according to the ISO 26262 [8] safety standard and evaluate the results in realistic scenarios of the civil aviation and automotive industries.

Acknowledgements. This work was partially supported by the EC, through project KARYON (IST-FP7-STREP-288195); and by FCT, through Multiannual Funding to LaSIGE (UI 408), the CMU|Portugal program, and the Individual Doctoral Grant SFRH/BD/60193/2009.

References

1. AEEC: Avionics application software standard interface. ARINC Specification 653, Airlines Electronic Engineering Committee (AEEC) (Jan 1997, rev Nov 2010)
2. AUTOSAR: Requirements on operating system, v3.1.0, release 4.1, revision 1 (Jan 2013)
3. AUTOSAR: Specification of operating system, v5.1.0, release 4.1, revision 1 (Feb 2013)
4. Avižienis, A.A.: 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)
5. Barhorst, J., Belote, T., Binns, P., Hoffman, J., Paunicka, J. Sarathy, P., Scoredos, J., Stanfill, P., Stuart, D., Urzi, R.: A research agenda for mixed-criticality systems (2009), white paper
6. Craveiro, J.P., Rufino, J., Singhoff, F.: Architecture, mechanisms and scheduling analysis tool for multicore time- and space-partitioned systems. ACM SIGBED Review 8(3) (Jul 2011)
7. Heiser, G.: The role of virtualization in embedded systems. In: First workshop on Isolation and integration in embedded systems (IIES'08). Glasgow, Scotland (Apr 2008)
8. ISO: ISO 26262: Road vehicles - functional safety. Int'l Standard ISO/FDIS 26262 (2011)
9. Kim, K.H.: The distributed recovery block scheme. In: Lyu, M.R. (ed.) Software Fault Tolerance, chap. 8, pp. 189–209. John Wiley Sons (1995)
10. Obermaisser, R., Kopetz, H. (eds.): GENESYS: A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems (Sep 2009)
11. Randel, B., Xu, J.: 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)
12. Rufino, J., Craveiro, J., Verissimo, P.: Architecting robustness and timeliness in a new generation of aerospace systems. In: Casimiro, A., de Lemos, R., Gacek, C. (eds.) Architecting Dependable Systems VII, LNCS, vol. 6420, pp. 146–170. Springer Berlin Heidelberg (2010)
13. Sha, L.: Using simplicity to control complexity. IEEE Software 18(4), 20–28 (Jul/Aug 2001)
14. Verissimo, P., Casimiro, A., Fetzer, C.: The timely computing base: Timely actions in the presence of uncertain timeliness. In: International Conference on Dependable Systems and Networks. pp. 533–542. New York City, NY (Jun 2000)
15. Verissimo, P.: Uncertainty and predictability: Can they be reconciled? In: Schiper, A., Shvartsman, A., Weatherspoon, H., Zhao, B. (eds.) Future Directions in Distributed Computing, LNCS, vol. 2584, pp. 108–113. Springer Berlin Heidelberg (2003)
16. Verissimo, P.E.: Travelling through wormholes: a new look at distributed systems models. SIGACT News 37(1), 66–81 (Mar 2006)