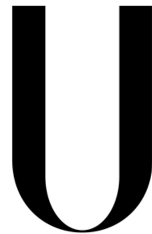
UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática



SAFETY KERNEL FOR COOPERATIVE SENSOR-BASED SYSTEMS

Pedro Nuno Pereira Nóbrega da Costa

DISSERTAÇÃO
MESTRADO EM SEGURANÇA INFORMÁTICA

2013

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

SAFETY KERNEL FOR COOPERATIVE SENSOR-BASED SYSTEMS

Pedro Nuno Pereira Nóbrega da Costa

DISSERTAÇÃO

Dissertação orientada pelo Prof. Doutor José Manuel de Sousa de Matos Rufino
e co-orientada pelo Mestre João Pedro Gonçalves Crespo Craveiro

MESTRADO EM SEGURANÇA INFORMÁTICA

2013

Acknowledgments

During the last year that now ends with the writing of this thesis, I received the support of many people, without whom this work would have been much harder, or even impossible, to accomplish. I, hereby, leave my eternal gratefulness to them.

First, I would like to express my gratitude to my supervisor, Professor José Rufino, and to my co-advisor, (now also Professor and very soon PhD) João Craveiro, for the opportunity they gave me to work with them and for their orientation, patience and continuous support over the last year which were vital to the research and writing of this thesis.

One special word for the Department of Informatics and specially to the Navigators and LaSIGE research group for the way they welcomed the new students and for always having been a superb place to work, study and learn.

Furthermore, a special thanks to Professor António Casimiro for its support along all the year and collaboration in the paper published in the ASCoMS 2013 proceedings. A word of appreciation goes as well to all the remaining people involved in the KARYON project for their collaboration, comments and ideas, which were undoubtedly very helpful in the various stages of this work.

I thank my fellow labmates of the Great Lab 25, with whom it was a pleasure to work with and for creating the best work environment I could ever imagine. To all the discussions, inspiration, humorous moments, lunches, dinners, sleepless nights, coffee breaks, bridge breaks, no-reason breaks, and to all the support you gave me along this year, both in good and bad moments (specially the latter), a big thank you.

Last but not least, a HUGE thank you for my family: my Mother, Father and Sister, for the support and hope given through all the stages of my studies and thesis. Thank you for always having believed in me and for never losing the patient, even when all hope and strength to continue seemed lost. Without your support I would never been able to accomplish this goal. Thank you.

This work was partially supported by the European Commission — through project KARYON (IST-FP7-STREP-288195) — and by Fundação para a Ciência e a Tecnologia (FCT) — through Multiannual Funding to LaSIGE (UI 408) and the CMU|Portugal program.

"All rising to great place is by a winding stair" - Francis Bacon

Abstract

Future safety-critical systems, used in, for example, the aerospacial, aeronautic and automotive industries, call for innovative computing architectures, with increased complexity. These systems must still cope with strict requirements, not only in terms of safety and reliability, but also in terms of size, weight and power consumption (SWaP).

Traditional approaches used in the design of such critical systems, rely on proving and guaranteeing, at design time, the safety and predictability of their applications. However, with the emergence of new technological solutions and the increase of the complexity of applications, it gets harder or even infeasible to prove their safety by design, limiting the scope and possible features to include in such systems. For instance, the use of wireless communications opens a new world of possibilities: it may be used to develop smart vehicles that cooperate with each other to achieve some common goal. However, due to its uncertainty, the development of such applications for safety-critical systems turns out to be a challenging task.

In this thesis, we propose a hybrid architecture, in which simple and predictable components coexist with complex and unpredictable ones, without compromising safety, despite the unavoidable uncertainty. The inclusion of complex components into safety-critical systems allows the emergence of new applications that provide new features or that improve the existing ones. Furthermore, we want to deal with the uncertainty that characterizes wireless communications and provide mechanisms which allow systems to cooperate with each other in a safe way.

We rely on a component called Safety Kernel, in charge of monitoring and managing the runtime configuration of the system, forcing it to adapt to faults and runtime constraints in order to avoid hazardous situations. We describe the architecture and role of such 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 prototype implementation of such Safety Kernel over AIR, an architecture based on the concept of Time- and Space Partitioning (TSP) developed for aerospace systems.

Keywords: Architectural hybridization, Real-time Systems, Cooperation, Autonomous systems, Time-and-Space Partitioned systems

Resumo

Os sistemas críticos, usados em indústrias como a aeroespacial, aeronáutica ou automóvel, requerem novas soluções tecnológicas para responder à constante procura por novas funcionalidades que respondam aos novos desafios do futuro, tornando-se cada vez mais complexos. Estes sistemas necessitam, contudo, de respeitar elevados e rígidos requisitos, não só em termos de segurança na operação e fiabilidade, mas também em termos de requisitos de tamanho, peso e consumo energético.

Arquiteturas tradicionais usados no desenho deste tipo de sistemas críticos baseiam a segurança na operação possibilidade de provar, em tempo de desenvolvimento, que o sistema garante a previsibilidade necessária. Contudo, o aparecimento de novas tecnologias acarreta um aumento na complexidade das aplicações usadas, o que torna o objetivo de provar a sua fiabilidade uma tarefa árdua ou mesmo impossível, limitando as funcionalidades passíveis de serem integradas nestes sistemas. Por exemplo, o aparecimento de comunicações sem fios abriu um novo mundo de oportunidades: a mesma poderia permitir um conjunto de veículos comunicar e cooperar mutuamente para atingir um objetivo comum. Contudo, a incerteza que caracteriza este tipo de comunicações tem travado o desenvolvimento de aplicações passiveis de ser usados por sistemas críticos.

Nesta tese, propomos uma arquitetura híbrida, constituída por componentes simples e previsíveis que coexistem com componentes complexos e imprevisíveis sem que isso, sem que essa coexistencia ponha em causa as garantias de segurança na operação. A possibilidade de incluir novas aplicações, que façam uso de novas tecnologias, abre portas à introdução de novas funcionalidades em sistemas críticos, permitindo melhorar a performance e serviço prestado pelos sistemas atualmente existentes.

A nossa arquitetura assenta num componente chamado Núcleo de Segurança (Safety Kernel), que tem como tarefa a monitorização dos requisitos de segurança e a gestão da configuração do sistema, assegurando-se que este se adapta às limitações observadas e que podem por em causa a segurança do sistema, evitando assim possíveis acidentes. Este documento descreve a arquitetura deste componente bem como a integração e interação do mesmo na arquitetura do sistema, apresentando a implementação de um protótipo do mesmo na arquitetura AIR - uma arquitetura baseada no conceito de compartimentação no espaço e tempo (CET) desenvolvida para sistemas aeroespaciais..

Palavras-chave: Hibridização arquitetural, Sistemas de tempo-real, Cooperação, Sistemas Autónomos, Compartimentação no Espaço e Tempo

Resumo Alargado

O constante aumento de tráfego automóvel nas nossas cidades ou o crescente número de aviões que diariamente cruzam os nossos céus colocam um grande desafio ao futuro da nossa sociedade. Visto que não é possível construir novas estradas, expandir os nossos céus ou construir novos aeroportos ao mesmo ritmo a que o tráfego aumenta, é imperativo desenvolver novas soluções que permitam aumentar a capacidade das infraestruturas atuais de maneira a lidar com este problema.

Os sistemas de controlo usados em veículos automóveis ou aeronáuticos são compostos por dezenas de componentes interligados como sensores e atuadores. Recentes desenvolvimentos tecnológicos no domínio deste tipo de sistemas embebidos e no domínio das comunicações sem fios são um ponto chave no aparecimento de novas aplicações capazes de fornecer novas funcionalidades e de dar resposta à necessidade de novas soluções. Por exemplo, os desenvolvimentos dos últimos anos no domínio das comunicações sem fios permitem o desenvolvimento de veículos inteligentes capazes de operar autonomamente e de comunicar e cooperar com outros veículos, capazes de fornecer serviços impossíveis nos dias de hoje. A importância da cooperação em sistemas de controlo já foi demonstrada em missões militares e em redes de sensores sem fios. Recentemente, a sua adoção em sistemas de transporte tem recebido grande atenção como um meio de ultrapassar os desafios do futuro. Por exemplo, na indústria automóvel, a cooperação entre conjuntos de veículos permitir atingir uma redução nos problemas de tráfego, consumo energético e até acidentes através da negociação de passagem em cruzamentos ou da navegação em grupo em autoestrada. No domínio da aviação, a cooperação pode permitir que aviões não tripulados (conhecidos como Unmanned Aerial Vehicles (UAVs) cooperem com aviões tradicionais, permitindo o seu uso em espaços aéreos partilhados, impossível nos dias de hoje.

No entanto, as aplicações usadas em sistemas críticos, como são exemplo os usados nas industrias descritas, têm elevados requisitos em termos de segurança na operação, previsibilidade e confiabilidade. Tradicionalmente, estes requisitos são garantidos durante o desenvolvimento dos sistemas através da sua verificação, validação e certificação. Contudo, quando se consideram aplicações mais complexas e que usam fontes de informação imprevisíveis, como comunicações sem fios, a tarefa de garantir a sua previsibilidade em

tempo de desenho torna-se difícil ou mesmo impossível, o que levanta problemas sérios quando se precisa de garantir segurança na operação de um sistema.

Idealmente, gostaríamos de tirar partido dos benefícios trazidos por estas novas tecnologias e funcionalidades, como a cooperação entre veículos, com vantagens implícitas no que diz respeito ao serviço prestado pelos mesmos, sem descurar a segurança na operação dos mesmos. Mas, devido à dificuldade em provar a sua previsibilidade, torna-se necessário lidar com as possíveis incertezas e falhas que possam acontecer, pelo que a sua inclusão neste tipo de sistemas-criticos é uma tarefa desafiante.

Além dos requisitos já descritos, este tipo de sistemas possui ainda elevadas restrições em termos de tamanho, peso e consumo energético, obrigando a que a inclusão de novos componentes e funcionalidades não aumente as necessidades deste conjunto de requisitos. Para fazer face a este problema, as indústrias do mundo aeronáutico e automóvel têm seguido uma nova tendência, cujo objetivo é a integração de múltiplas funções na mesma plataforma, o que permite obter melhor eficiência na utilização dos recursos disponíveis. Neste campo, a adoção de standards como o ARINC 653 e o AUTOSAR são prova desta tendência. Como as diversas funções alojadas podem possui diferentes níveis de criticidade e de confiabilidade, a sua integração levanta questões no que toca a segurança na operação do sistema. Este problema requer mecanismos de contenção de falhas entre as diferentes funções, que garantam que uma eventual falha numa função de baixa criticidade não afeta a execução das demais funções. Esta necessidade levou ao aparecimento do conceito de sistemas Compartimentados no Espaço e Tempo (CET), no qual as diferentes aplicações de um sistema são compartimentadas e isoladas entre si no domínio do espaço e tempo, garantido que eventuais falhas são contidas ao seu domínio de ocorrência.

A motivação para este trabalho prende-se assim com a necessidade de encontrar técnicas que permitam a integração de componentes e outras fontes de incerteza em sistemas críticos sem que eventuais falhas isso ponham em causa a sua correta operação. Para tal, esta tese propõe uma solução baseada no conceito de hibridização arquitetural, que permite que componentes com um comportamento imprevisível coexistam no mesmo sistema com componentes provados corretos, permitindo tirar partido das novas funcionalidades fornecidas pelos componentes incertos e sem comprometer a segurança na operação. Isto requer mecanismos para gerir a configuração do sistema e garantir que o mesmo adapta o seu comportamento a restrições observadas em tempo de execução com o objetivo de garantir a sua segurança.

Com esse objetivo, nesta tese descrevemos um componente que adicionamos à arquitetura descrita — denominado Núcleo de Segurança — que devido à sua simplicidade é passível de ser provado seguro e por isso implementado na parte previsível do sistema. O Núcleo de Segurança tem como objetivo gerir a configuração do sistema e com isso garantir a sua correta operação. Nesta tese descrevemos a arquitetura e as funções deste Núcleo de Segurança no contexto da arquitetura descrita. O Núcleo de Segurança baseia a

sua operação num conjunto de regras predefinidas, usadas para monitorizar a execução do sistema e reconfigura-lo em caso de necessidade. Descrevemos em detalhe os componentes individuais do Núcleo de Segurança e as interações que os mesmos têm com o resto do sistema. Por fim, com o intuito de demonstrar a sua viabilidade, apresentamos um protótipo de uma implementação do Núcleo de Segurança num sistema baseado na arquitetura AIR, uma arquitetura desenvolvida para a indústria aeroespacial que implementa o conceito de Compartimentação no Espaço e Tempo (CET) e que segue a especificação ARINC 653.

Do trabalho desta tese foi produzido o seguinte artigo científico, publicado numa conferência internacional:

- Pedro Nóbrega da Costa, João Pedro Craveiro, António Casimiro, and José Rufino. Safety kernel for cooperative sensor-based systems. In *Safecomp 2013 Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS)*, Toulouse, France, September 2013

Como trabalho futuro, propomos a implementação do Núcleo de Segurança num ambiente real, onde sejam testados resultados do mesmo num cenário realista, bem como a verificação, validação e certificação da implementação do Núcleo de Segurança de acordo com as normas usadas pela indústria.

Contents

List of Figures xix

List of Tables xxi

Acronyms xxiv

1 Introduction 1

1.1 Motivation . . . 3
1.2 Objectives . . . 3
1.3 Contributions . . . 4
1.4 Institutional Context . . . 4
1.5 Publications . . . 5
1.6 Outline . . . 5

2 Related Work 7

2.1 Real-Time Systems . . . 7
2.2 Safety Concepts . . . 8
2.3 Hybrid Models . . . 9
2.3.1 Other hybrid systems examples . . . 10
2.4 Fault Containment . . . 11
2.4.1 Time and Space Partitioning - TSP . . . 12
2.5 Industry Standards . . . 12
2.5.1 Integrated Modular Avionics . . . 12
2.5.2 ARINC 653 . . . 13
2.5.3 AutoSAR . . . 16
2.6 ARINC In Space RTOS - AIR . . . 16
2.6.1 AIR Architecture . . . 17
2.6.2 APEX Interface . . . 18
2.6.3 Partition Management Kernel . . . 18
2.6.4 Mode Based Scheduling and Adaptability . . . 20
2.7 Summary . . . 20

3 Cooperative Sensor-Based Systems 23
3.1 Sensor-based systems . . . 23
3.2 Cooperative Systems . . . 24
3.2.1 Communication . . . 25
3.3 Fault Model . . . 26
3.3.1 Quality of data . . . 27
3.4 Summary . . . 28

4 System Architecture 29
4.1 Levels of Service . . . 29
4.2 Hybrid Architecture . . . 31
4.3 Safety Kernel . . . 34
4.4 Summary . . . 35

5 Safety Kernel 37
5.1 Overview . . . 37
5.2 Safety Kernel design issues . . . 37
5.2.1 Runtime information . . . 37
5.2.2 Design time information . . . 38
5.2.3 Adapting the Level of Service . . . 39
5.3 Architecture . . . 40
5.3.1 Components of the Safety Kernel . . . 40
5.3.2 External Components . . . 46
5.3.3 Interfaces . . . 48
5.3.4 Operating System Support . . . 53
5.4 Summary . . . 54

6 Safety Kernel Implementation 55
6.1 Use Case system description . . . 55
6.1.1 Functionalities Definition . . . 56
6.1.2 Configuration Rules . . . 58
6.1.3 Data Multiplexing . . . 59
6.2 Prototype . . . 60
6.2.1 Communication and timing failures detection . . . 60
6.2.2 Rules Implementation . . . 61
6.2.3 Execution Samples . . . 63
6.3 Summary . . . 64

7 Conclusion 67
7.1 Future Work . . . 67

Bibliography **75**

List of Figures

2.1 Real-Time Classes Utility/Time function 8
2.2 IMA architecture 13
2.3 ARINC 653 architecture 14
2.4 AIR architecture from [14] 18
2.5 Two-level hierarchical scheduling 19
2.6 AIR Interpartition communication mechanisms 20

3.1 Basic Control Loop 23
3.2 Example of Adaptive Cruise Control Functionality 24

4.1 Hybrid Architecture 31
4.2 ACC functionality with multiple Levels of Service 32
4.3 One component used in the provision of multiple functionalities 33
4.4 Reconfigurable Component with multiple modes of execution 33
4.5 System Architecture with Safety Kernel component 35

5.1 System components overview and interaction 41
5.2 Local LoS basic behaviour 44
5.3 Safety Manager basic behaviour 45
5.4 Data Component Multiplexer basic behaviour 46
5.5 Cooperative LoS Evaluator basic behaviour 47
5.6 Data Validity Interface flow 50
5.7 Cooperative LoS Interface 50
5.8 Mode Switch Interface 51
5.9 Data Component Multiplexer Interface 53

6.1 Use case system components 56
6.2 System implementation over AIR 61
6.3 Prototype Sample Execution 63
6.4 Prototype - Reduced GPS validity example 64
6.5 Prototype - Timing fault example 64

List of Tables

5.1 Interfaces for Safety Kernel support 49

6.1 Auto-Steering functionality safety requirements 57
6.2 Adaptive Cruise Control functionality safety requirements 58
6.3 Configuration Rules 59
6.4 Speed Calculator Multiplexing Rules 60

Acronyms

ACC Adaptive Cruise Control.

AIR ARINC 653 In Space RTOS.

APEX Application Executive.

AS Auto-Steering.

ASC Auto Steering Calculator.

ASIL Automotive Safety Integrity Levels.

AUTOSAR AUTomotive Open System ARchitecture.

BSC Basic Speed Calculator.

CAN Control Area Network.

COTS Commercial of the Shelf.

DAL Design Assurance Level.

DIMA Distributed Integrated Modular Avionics.

GPS Global Positioning System.

HM Health Monitoring.

IMA Integrated Modular Avionics.

LoS Level of Service.

MTF Major Time Frame.

OS Operating System.

PAL POS Adaptation Layer.

PMK Partition Management Kernel.

POS Partition OS.

PST Partition Scheduling Table.

RTEMS Real-Time Executive for Multiprocessor Systems.

RTOS Real-Time Operating System.

SSC Smart Speed Calculator.

SWaP Size, Weight and Power consumption.

TSP Time and Space Partioning.

UAV Unmanned Aerial Vehicle.

Chapter 1

Introduction

The constantly increasing automotive traffic in our cities or the increasing number of planes crossing the airspace puts substantial challenges to future societies. Given the fact that it is not possible to build new roads or extend the airspace at the same pace as the traffic increases, the throughput and capacity of current infrastructures has to be improved by other means, emphasizing the need for new technological solutions that cope with this problem.

Control systems used by individual vehicles are composed of dozens of connected *components* such as sensors and actuators each used for a different purpose or *function*. Emerging technological improvements in the domains of such components, such as embedded computing, sensing, actuation and wireless communication are key enabling factors for the emergence of new applications able to cope with the demand for new functionalities. For instance, recent improvements in the domain of wireless communication are a key factor in the development of smart systems that autonomously cooperate and interact, being able to provide new and improved functionalities. The importance of cooperation had already been acknowledged for using in military systems or wireless sensor networks. Finally, the use of cooperation in transportation systems has received considerable attention over the last few decades as a mean to overcome future challenges. By way of illustration, in the automotive domain, cooperation between smart cars may allow a reduction of traffic and fuel consumptions. In the avionics domain, Unmanned Aerial Vehicles (UAVs) might be able to cooperate with other airplanes, allowing to use these systems in shared airspace areas with regular airplanes, impossible with current solutions.

However, applications used in *safety-critical* systems, such as the ones used by automotive, avionic and aeronautic industries, have very strict requirements in terms of safety, reliability and predictability. Traditional approaches for the design of safe control systems rely on the possibility of guaranteeing these requirements in design time during the several stages of the system development. Nevertheless, when considering applications based on more complex algorithms and using new sources of data such as wireless communication networks, it becomes difficult to ensure their correctness in design time, which is prob-

lematic when safety is a fundamental attribute. Ideally, we would like to be able to exploit the benefits and functionalities brought by new applications such as cooperation between vehicles, with implicit gains to vehicles as a whole and to traffic, without making any concessions on safety. But, due to the difficulty of guaranteeing their safety, it becomes necessary to deal with the possible impact of the uncertainties, for which their inclusion in safety-critical systems shows to be a challenging task.

Furthermore, requirements in terms of Size, Weight and Power consumption (SWaP) for this kind of systems also became more strict, rising the need to develop and integrate new functionalities in such systems without increasing their demand for this set of requirements. To cope with this, several aerospace and automotive industries have been following a trend towards the integration of multiple functions in the same computing platform, achieving an optimization in the use of available resources, on which the adoption of standards like the ARINC 653 and AUTOSAR are examples of. As the multiple functions hosted in one system may have distinct criticality and safety levels, their integration in the same platform rises safety issues, requiring fault containment mechanisms that ensure that faults in low-criticality functions do not affect the execution of high-criticality ones, leading to the emergence of the Time and Space Partioning (TSP) concept.

Addressing the challenge of including new functionalities without compromising safety requires innovative solutions to guarantee reliability of one system even in the occurrence of unforeseen failures.. To deal with it, this thesis proposes a solution based on the concept or architectural hybridization, allowing to have some components that are not proven-safe, and therefore that may show an unpredictable behaviour along with some others proven-safe and predictable components, being able to exploit the advantages provided by new applications without compromising safety. This requires mechanisms for managing the system's configuration and constrain its execution whenever it experiences faults in order to guarantee safety.

For that, we define an additional component to our architecture — the Safety Kernel — implemented in the predictable part of the system that manages the system behaviour and ensures safety. In this thesis, we describe the architecture and role of such Safety Kernel in the context of a hybrid system architecture. Based on a predefined set of rules, the Safety Kernel monitors the operation of the system and manages its operational configuration and forces it to adapt to observed runtime constraints.

We describe the individual components of the Safety Kernel backed up by a brief description of the surrounding system architecture and how it interacts with the remaining *functional components* of the system architecture — the components in charge of providing functionalities to the system. Finally we present a prototype implementation of the Safety Kernel based on ARINC 653 In Space RTOS (AIR), a TSP architecture developed with the aerospace in mind, but applicable to other safety-critical industries.

This work is integrated in the KARYON [1], an European project that proposes a new perspective to improve performance of smart vehicle coordination, whose objective 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.

1.1 Motivation

The present work was motivated by the following observations:

- Current state of the art smart vehicles, such as UAVs or smart cars, are not allowed to operate in the public air space or roads due to the risk of causing severe damage cannot be excluded with sufficient certainty;

- The emergence of new technological solutions implies an increase in the complexity of one system, which makes the task of proving its safety and predictability in design time a hard, or even impossible, task, limiting the integration of such solutions in safety-critical systems.

- Even on proven safe systems, failures may still happen due to unpredictable situations. Safety-critical systems must be able adapt themselves to unexpected situations in order to avoid possible catastrophic consequences.

1.2 Objectives

The main objective of the present work is to present the definition, requirements, design and prototype implementation of the Safety Kernel, a component to be used by safety-critical systems, that aims at providing predictability and ensuring safety despite the existence of sources of uncertainty. We present its architecture and an example of implementation within the AIR architecture. The Safety Kernel must be a trusted component, simple enough to have its reliability proven in design time by means of verification, validation and certification.

In the presence of failures, the Safety Kernel must be able to take action to avoid any catastrophic consequence. This means that, at all time, the Safety Kernel must be aware of the system's constraints and, when needed, trigger some system reconfiguration so that the new configuration meets the observed constraints.

Furthermore, we present an architectural pattern based on a hybrid model composed of components based on different assumptions and providing different trust degrees, as well as the integration of the Safety Kernel in such architecture.

[1] KARYON — Kernel-Based ARchitecture for safetY-critical cONtrol — www.karyon-project.eu

1.3 Contributions

The main contributions of the work described in this thesis are:

1. The definition of a hybrid architecture for safety-critical systems that:
   (a) is composed by both reliable and unreliable components;
   (b) allows a flexible and modular integration of new components;
   (c) may operate with different configurations, providing different guarantees and functionalities;
2. The definition of a generic Safety Kernel that:
   (a) monitors and looks up for constraints and faults in runtime;
   (b) manages the system's configuration based on such observations;
   (c) create mechanisms that enable the cooperation between systems;
3. A prototype implementation of the Safety Kernel using the AIR architecture.

1.4 Institutional Context

This work took place at the Navigators research group, part of the Large-Scale Information Systems Laboratory (LaSIGE-FCUL), unit of the Informatics Department (DI) of the University of Lisbon, Faculty of Sciences, and was developed within the scope of KARYON, an European project, that consists on a consortium with members from Faculty of Sciences of University of Lisbon, Chalmers University of Technology, SP, Embraer, GMV, University of Magdeburg and 4S Group. The main objective of the KARYON project is, as stated in its mission, "to provide system solutions for predictable and safe coordination of smart vehicles that autonomously cooperate and interact in an open and inherently uncertain environment", which is achieved by "investigating new ways of achieving fault-tolerant distributed control" and with the "provision of a safety kernel to constraining system operation in order to avoid hazardous situations."

As a junior researcher, the author of this thesis was a member of the KARYON project, having participated in multiple project activities, such as regular meetings for identification of problems and discussion of possible solutions, project meetings with other partners, helping in the production of project deliverables and publication of a scientific paper at an international conference.

1.5 Publications

The work present in this thesis generated the following publication:

- Pedro Nóbrega da Costa, João Pedro Craveiro, António Casimiro, and José Rufino. Safety kernel for cooperative sensor-based systems. In *Safecomp 2013 Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS)*, Toulouse, France, September 2013 [30].

And the following report:

- Mário Calha, João Craveiro, Pedro Nóbrega da Costa, and António Casimiro. First report on safety kernel definition. Technical report, KARYON Deliverable WP4.2, FCUL, 2013 [10].

1.6 Outline

The remainder of this thesis is structured as follows:

Chapter 2 - Reviews some basic concepts for understanding the context of the work, including real-time systems, concepts about safety and how it is handled and guaranteed in some approaches emerging both from the civil aviation and automotive worlds. It is also given a detailed overview over the AIR architecture;

Chapter 3 – Describes in detail the challenges of achieving cooperation in sensor based systems and the fault model considered for this work;

Chapter 4 – Gives a high-level view of the system architecture;

Chapter 5 - Describes the requirements and the architecture of the Safety Kernel component;

Chapter 6 - Presents an use cases system and gives an example of a prototype implementation of the Safety Kernel under the AIR system;

Chapter 7 - Closes this documents with concluding remarks and some possible future developments.

Chapter 2

Related Work

2.1 Real-Time Systems

A *real-time system* is, according to its definition, a system whose requirements are defined in terms of *timeliness* constraints [27, 45, 41]. In other words, a real-time system is a system whose correctness is defined not only with constraints in the value domain, but also in the time one, so that its response and progression must satisfy strict timeliness requirements. As such, and contrary to a common misconception, the main goal of a real-time system is not to achieve a high performance level, but rather to ensure *predictability* and *determinism* in its execution. These goals are usually achieved through strict resource scheduling analysis that define how resources are used in runtime by applications, guaranteeing that each application has the needed resources enough time to complete their actions before their *deadline*.

The scope of real-time systems is wide, and may go from simple control systems, such as an oven temperature or a traffic light controller to more complex systems such as the onboard navigation system of a car or plane or an air traffic control system.

Based on the timeliness requirements and on their impact on the system's correctness, three different classes of real-time systems are defined: hard real-time, soft real-time and safety-critical systems.

A *hard real-time* system is the system in which no timing failures may occur. As such, in case of a timing failure, its guarantees do not hold and its correctness is not guaranteed. Its correctness is hardly dependent on the assumption that all the computations produce the desired result before their *deadline* — the instant when the utility of a computation result drops to zero, as illustrated in Figure 2.1 a). One example of such real-time system is an on-board flight control system, in which a missed deadline may compromise the entire system's correctness.

Soft real-time systems are, in the other hand, systems where occasional timing failures are accepted and tolerated. Missing a deadline leads to a consequence a reduction in the utility of a computation result, lowering the quality of the service provided, reaching

progressively the zero if multiple deadlines are missed. An example of such systems are online video streaming services, where missing some frames (under a given threshold) is acceptable, having as a consequence a reduction in the quality of the video provided, but, as long as the missed deadlines remain under that given threshold, the service is able to continue its operation without any major consequences. This is illustrated in Figure 2.1 b).

Finally, *safety-critical* systems are the ones where timing failures must be avoided but, in case they occur, they must be handled and treated as exceptions to avoid possible catastrophic consequences. As such, a safety-critical system is the one where a mishandled timing failure may have catastrophic consequences, such as the loss of human lives. One example of such is an air-traffic control system, where a single mishandled timing failure may lead to a mid-air plane collision. In this case, the system must be able to, somehow, handle a possible failure to avoid any major consequences. Figure 2.1 c) represents this, where a catastrophic event is represented with a negative value of the utility of the computation result.

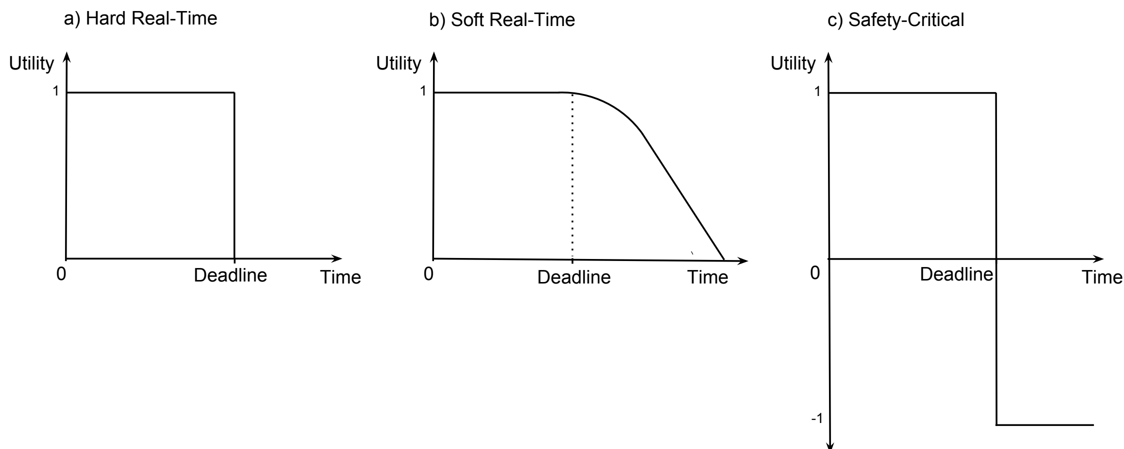


Figure 2.1: Real-Time Classes Utility/Time function

2.2 Safety Concepts

Safety is an intuitive concept that may be defined in several different ways, but similar in intention. When talking about dependable systems, some definitions for safety may be:

- the absence of unacceptable — where risk is the product of the probability of occurrence of a failure and its severity *risk* [21];
- the freedom from accidents or losses [28];
- the guarantee that a given property is respected or the degree to which a system, upon failing, does so in a non-catastrophic manner [45].

When designing a safety-critical system, it is necessary to prove and guarantee in design time that all safety requirements and system specifications will be fulfilled in runtime. This is done by means of *verification* and *validation* of the software and hardware, through the multiple stages of development of one system. Verification and validation include multiple techniques such as dynamic and static testing of the system, formal verification, fault injection and *safety analysis* [42].

These techniques intend to, according to some assumptions, statically prove and guarantee the system's safety in all cases, even in the presence of faults. The typical approach to do so, is based on making worst case assumptions and ensure safety (or reduce the risk to an acceptable level) under those pessimistic assumptions. For high criticality systems, these assumptions tend to be inflexible and are generally very constrained due to the need of guaranteeing strict and tight safety requirements.

Besides verification and validation, when talking about commercial systems, it is also necessary to talk about their *certification* process. Certification is usually required for legal reasons, to prove that a system complies with all the requirements specified by the government or some organization and that is acceptable for a commercial use. Certification must, therefore, follow specific guidelines and standards.

In the context of the avionics industry, certification must obey to standard guidelines such as the RTCA-DO-178 [35] or, in the automotive context, to the ISO 26262 [23] that define rules and practices for the complete development lifecycle of electrical and electronic systems. In general, certification is done by analysing the system through all its development stages and compare it with independent defined safety requirements. These safety requirements vary according to the criticality level of the evaluated component that according to the standard may have different levels. For instance, the DO-178 defines the so-called Design Assurance Level (DAL), whereas for the automotive industry, the ISO 26262 defines the Automotive Safety Integrity Levels (ASIL). These levels are determined at the beginning of the development of a system and are based on a safety analysis made for each component that measures the consequences that a failure on that component would have on the system's safety, ranging from the catastrophic to the no effect levels. Based on the calculated level, each standard specifies the requirements and methods that must be used to achieve that level and guarantee safety or the reduction of the risk to an acceptable level.

2.3 Hybrid Models

When designing a system, 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, which has a an implicit impact on the possible solutions [32]. The

set of assumptions characterize the relevant attributes of the environment for which the solutions will be developed constituting what is generally denominated as system model. The system model provides an abstraction of the real system, defines what the designer can take as granted, and how these properties are provided or enforced.

In a *homogeneous* system model, the set of assumptions that is made to the system is global, being applied to the system and all its components as a whole and do not change over time [12]. On the other hand, *hybrid* system models [43] represent systems in which different parts have different properties and can rely on different sets of assumptions (e.g., fault model, synchronism) and in which these assumptions may vary with time, presenting a number of advantages when compared with homogeneous approaches [44].

One simple example of a system well described by a hybrid system model is a system controlled by 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.

For the problem presented in this thesis, we will exploit the expressiveness of hybrid models to define a hybrid architecture and to design a solution that addresses the conflicting goals of predictability and uncertainty, by merging in the same system components assumed to be both reliable and unreliable.

2.3.1 Other hybrid systems examples

The Simplex [38, 39] approach proposes this same idea of composing one system with both complex and simple control components with similar function in order to achieve a control system with improved performance without compromising control safety. In this approach, the idea is to "use simplicity to control complexity", by having two alternative control components executed in parallel that might be used for controlling some system. One is designed to achieve improved control at the expense of an increased complexity of the control algorithm. Other is designed to achieve a basic control, using a simple 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 prone to errors that may bring the controlled system to an unsafe state. The simple algorithm provides the necessary redundancy to compensate possible problems in the execution of the complex algorithm. Upon a detected failure in the complex one, the simpler one is called upon to take care of the execution of the system.

The recovery block concept [26] follows a hybrid model, where multiple alternatives for the same function are developed. Before executing the function, the system saves is state, and executes the first alternative, submitting its result to an acceptance test. If an

error is detected, the system rolls back to the saved state, and executes another alternative, preferably simpler than the previous, until finding one that passes the test.

In the N-version programming [6], also multiple versions are developed and executed simultaneously, whose results are compared and voted, being chosen the one with majority of voters. However, this approach requires to have a majority of correct results, instead of only one like the recovery block scheme. Still, the Simplex shows to provide a better reliability when compared with both N-version programming and recovery blocks [26], when considering that the available resources are limited.

From the Simplex approach we inherit the idea of implementing control functions redundantly, each implementation with different degrees of reliability and features, as explained above. However, Simplex is designed by assuming that faults are reflected on some external behaviour that may be observed by some existing sensor. We, on the other hand, consider that sensors may suffer from failures that affect the validity or the timeliness of their data, for which we cannot use directly sensor information to determine how well the system is being controlled and to decide when to switch the control algorithm being used by the system.

2.4 Fault Containment

A *mixed criticality system* is, according to its definition, "an integrated suite of hardware, Operating System (OS), middleware services and application software that supports the execution of safety-critical, mission-critical, and non-critical software within a single, secure compute platform" [7]. This coexistence of functions with different levels of criticality raises the need for mechanisms that provide fault containment that is: that faults in one application are contained and do not spread to or affect the execution of the others, avoiding that a fault in a non-critical application compromises the execution of a high-critical one.

Virtualization [19, 20] has been widely used as a mechanism to run multiple systems with different environments (called virtual machines) within the same computing platform. Virtualization provides isolation in the space domain between virtual machines, meaning that the memory space of each one is independent from the others guaranteeing that faults do not propagate from one machine to another. However, typical virtualization solutions do not provide temporal isolation nor predictability, meaning that delays on one virtual machine may still affect the timeliness of another, not being suitable for use with functions with real-time requirements. Hence, real-time systems, call for isolation not only in the space domain, but also in the time domain, leading to the emergency of the TSP concept, described next.

2.4.1 Time and Space Partitioning - TSP

The need to provide fault containment in real-time time systems lead to the emergence of the Time and Space Partioning (TSP) concept [46]. Systems based on TSP are able to integrate multiple components and to provide fault containment between them by enforcing segregation and independence from each other, both in the time and space domains.

Time partitioning means that one component shall not affect the timeliness of another. In other words, temporal failures, such as delays or crashes that happen in one component must be contained and may not affect the temporal requirements of another component; *Space partitioning* is concerned with ensuring that the addressing space memory of each component is independent and that one component may not access to memory of another.

When combined, these two properties ensure that, in the occurrence of a any failure in one given component of the system that same failure (and its possible consequences) will be contained to its domain of occurrence, without affecting the remaining components, providing the fault containment properties needed by real-time systems. This concept has been adopted by industries and their standards, as described in the next section.

2.5 Industry Standards

This section intends to give a glance at the Integrated Modular Avionics (IMA) concept and at two main architectural standards used in the avionics and automotive industries: the ARINC 653 and the AUTOSAR. A deeper view is given in the ARINC 653 since it is the base of the AIR architecture, explained in 2.6, used as use cases architecture for the present work.

2.5.1 Integrated Modular Avionics

Avionic systems are composed by multiple different functions, each possibly having a different degree of criticality, a different set of requirements and being produced by different developers.

Federated avionics are a legacy type of architecture for such systems, where each function is completely independent from the others, having its own dedicated board with its own computing resources. This architecture has been historically adopted as a mean of guaranteeing separation between functions with different criticalities, with the argument that it would prevent a lower criticality function to affect the behaviour of a higher criticality one. However, the fact that resources allocated to one function cannot be shared or reallocated to be used by another function, may lead to an inefficient use of the available resources.

With the increasing number of functions and their complexity present in critical systems like avionics, the use of dedicated boards leads to an exponential growth of the SWaP

requirements for such systems.

Integrated Modular Avionics (IMA) [1] is, in opposition to federated avionic systems, an alternative architecture concept for systems that integrate multiple functions, possibly with different criticalities, in a single shared computing system rather than having individual boards for each function. By integrating multiple functions in the same platform, this approach addresses the needs of modern systems, allowing to achieve a better optimization and efficiency in the use of available resources, reducing the set of SWaP requirements and, as consequence, the overall cost of one system. For instance, upon the construction of the commercial A380 airplane, Airbus declared that the use of IMA allowed to achieve a reduction of 50% in the processor units and a 40% decrease in the weight, compared with previous systems [33], which reduces both construction, maintenance and operational costs. The basic architecture of an IMA system may be seen in Figure 2.2

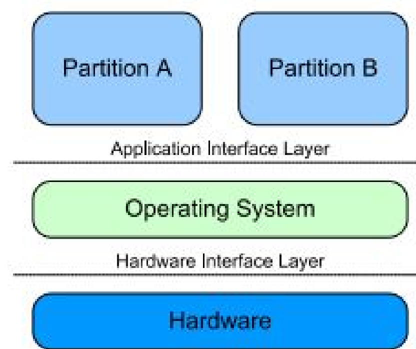


Figure 2.2: IMA architecture

As the figure pictures, under an IMA based system, functions are split and executed in individual logical containers known as *partitions*. These partitions are executed under a common OS that handles and abstracts the hardware layer, releasing the developer from focusing on the target platform and allow him to focus on the software, easing development and certification processes.

2.5.2 ARINC 653

A prominent example of a TSP system design, is the adoption of the ARINC specifications 651 (Design Guidance for IMA) [1] and 653 (Avionics Application Software Interface) by the avionics industry. *ARINC 653* [2] is a standard software specification for IMA systems that exploits TSP to provide providing fault containment and independent validation, verification and certification of applications. ARINC 653 has been adopted as standard by the avionics industry, namely the Airlines Electronic Engineering Committee, and an example of its use in civil aviation is the software used in the recent Airbus A380 and Boeing 787 [24] aircrafts.

ARINC 653 defines an interface between software applications and the underlying operating system layer, called Application Executive (APEX). Following the IMA specification, on the Application Software Layer, applications are executed in individual emph-partitions that consist on one or more *processes* that may invoke the services provided by the APEX interface or, if needed, may bypass it to invoke specific functions provided by the underlying core software layer, case in which are called *system partitions*. ARINC 653 specifies a mandatory set of services to be provided by the APEX and implemented by the OS Kernel such as partition and process management, inter and intra-partition communication, time management and health monitoring. Figure 2.3 shows the basic ARINC 653 architecture.

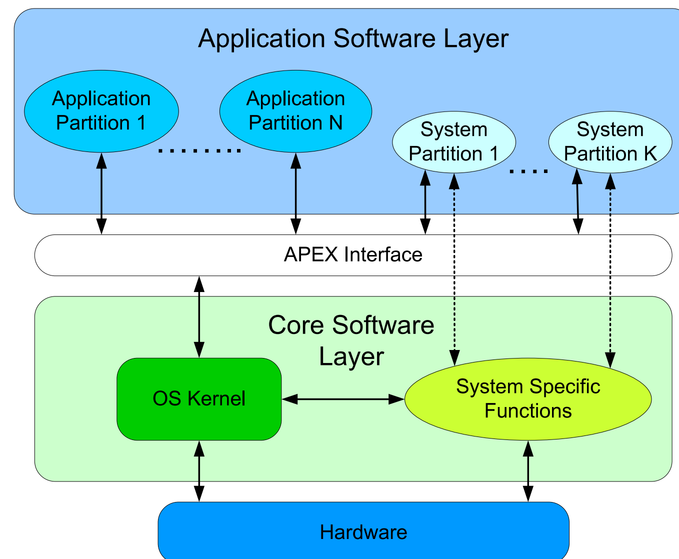


Figure 2.3: ARINC 653 architecture

Time and space partitioning

In order to provide fault containment and to allow independent application development between partitions, ARINC 653 enforces TSP between partitions.

Time partitioning is guaranteed by employing a two-level scheduling hierarchy: on the first level, partitions are scheduled by the system according to fixed and cyclic Partition Scheduling Table (PST), defined off-line at integration time that repeats itself over a Major Time Frame (MTF). On the second level, processes are scheduled inside each partition according to a local preemptive scheduler. The use of a fixed scheduled on the first level guarantees that even if a process of one given partition gets delayed, the remaining partitions are not affected, since the fixed scheduled limits the time assigned to that partition.

Regarding space partitioning, ARINC 653 implicitly ensures it by the concept of partitions. However, it does not specifies how it should be achieved by the operating system,

giving only the needed requirements of restricting applications to the memory addresses belonging to their own partition.

Interpartition Communication

Interpartition communication aims to support the transfer of information between partitions. Its relation with spatial partitioning implies the use of specific services that transfer the data from one partition to another without violating spatial segregation constraints.

Partitions communicate through communication channels that define a link between one source and one or more destination partitions. Partitions access to channels using predefined access points called *ports* that abstract the way the information flows on the channels. Ports provide the required resources to send or receive messages in a specific channel. A partition is allowed to exchange messages through multiple channels via their respective source and destination ports. ARINC 653 defines two kinds of ports to access a communication channels: sampling ports and queuing ports. *Sampling ports* are bufferless in the sense that each occurrence of a message overwrites the previous one. On the other hand, *queuing ports* store messages in a FIFO queue, thus, each new instance of a message cannot overwrite the previous one. Upon the creation of a sampling port, besides specifying the writer and reader partitions it is also possible define a refresh rate period: this parameter defines the minimum rate at which the writer should send messages, which allows to control whether messages arrive at a correct rate in the port or not. Upon reading from a sampling port, the reader receives a validity output parameter that indicates whether the age of the read message is consistent with the required refresh period attribute of the port or not.

These communication mechanisms are implemented by the underlying OS and made available to applications by the APEX interface that provides the primitives needed for communication such as create and destroy ports or send and receive messages.

Health Monitor

ARINC 653 also incorporates an Health Monitoring (HM) component that is in charge of monitoring resources and applications at different levels of the system, looking up for failures in the system, helping confining errors to their domain of occurrence. Upon the occurrence of an error (like a process deadline miss, memory protection violations, or even some hardware failures) an exception is raised and a handler, provided by the application developer may be executed. Hence, the HM is a fundamental component to achieve fault adaptability and dynamic behaviour. Faults may be signalized to the HM by applications or generated by the Operating System or detected by the HM itself. The HM component is spread virtually in the architecture components to ensure fault monitoring at all layers, which is why the HM component does not appear in Figure 2.3.

Extended Services

Besides these basic and mandatory services, ARINC 653 specification - Part 2 [3] describes some extended and optional services, such as a Multiple Module Scheduler that allows to define several scheduling plans instead of a single and static one. This mechanism may also be used to achieve adaptability, allowing to achieve different modes of operation instead of having a static one.

2.5.3 AutoSAR

AUTomotive Open System ARchitecture (AUTOSAR) [4] is a standard software architecture developed by a consortium of car manufacturers and other automotive suppliers for the automotive industry, with the goal of establishing a standard base infrastructure for the development of vehicular software.

AUTOSAR allows to fulfil the requirements of future vehicles, providing easy software updates, flexibility to integrate and transfer software between different systems, the use of Commercial of the Shelf (COTS) components and, as a consequence, an optimization of the production costs. AUTOSAR provides definitions in three areas; first, it defines a standard interface for software applications, implying that software may be developed independently of the target hardware platform and that applications may communicate with each other in a standardized way, using a virtual bus that abstracts how the information flows between different applications.

Secondly, it defines a middleware that specify modules in different areas such as services, communication, operating system and hardware abstraction. Furthermore, it specifies a set of templates to allow exchanging information between different companies. The top-level requirements for an AUTOSAR operating system include provisions that correspond, to some extent, to the notions of temporal and spatial isolation [4]. 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 [5].

2.6 ARINC In Space RTOS - AIR

AIR [37] is an architecture for a new generation of aerospace systems designed to implement and fulfil the requirements of TSP that, despite being inspired in the ARINC 653 specification, aims to improve upon such specification, diverting from it where its 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 [17]. Other

example where AIR diverts from the ARINC 653 specification is related with the Core Software Layer; due to their constraints, Real-Time Operating Systems (RTOSs) are usually more limited and use to lack relevant functions that are present in generic OSs. Porting those functions into a RTOS is a hard and expensive task. Motivated by this fact, instead of providing a single RTOS that is used by all partitions, AIR foresees the use of a different OS in each partition — called Partition OS (POS) — supporting the use of both generic and real-time operating systems [16], which allows to provide functions that were not possible to provide in a RTOS. AIR foresees the definition and use of multiple PSTs rather than a single fixed PST as defined in ARINC 653 basic services that may be interchanged at runtime. This feature allows AIR to define mode-based scheduling (e.g. different phases of a flight) providing a more efficient use of the resources (e.g., by deactivating unnecessary partitions) and also gives support to adaptation mechanisms in case of timing failures in one single partition [15].

The modularity of the AIR architecture and the fault containment properties provided by TSP allow to achieve *composability* property of AIR-based systems: composability means that properties that hold for individual components also hold after the components are assembled in the same system, which allows independent component development and eases the process of verification and validation of software components [18].

2.6.1 AIR Architecture

The AIR architecture, pictured in Figure 2.4 preserves the hardware independence inherited from ARINC 653 specification. It relies on an intermediary Partition Management Kernel (PMK) that acts as a hypervisor of the whole system, in charge of enforcing robust TSP properties, hosting crucial services such as partition management (scheduling and dispatching), communication (inter and intra-partition) and low-level interruptions handling.

Each partition may host a different OS, called POS, which can be either a RTOS or a non-RTOS. The POS Adaptation Layer (PAL) encapsulates each POS, hiding the particularities of each one in order to make them homogeneous and to provide a POS-independent interface to the surrounding components.

Applications access to services provided by the PMK using the APEX interface. APEX provides an interface derived from ARINC 653 specification, providing services such as process and time management, communication and health monitoring to applications. In addition, *system partitions* may bypass the APEX to invoke specific functions of the OS and not available via APEX.

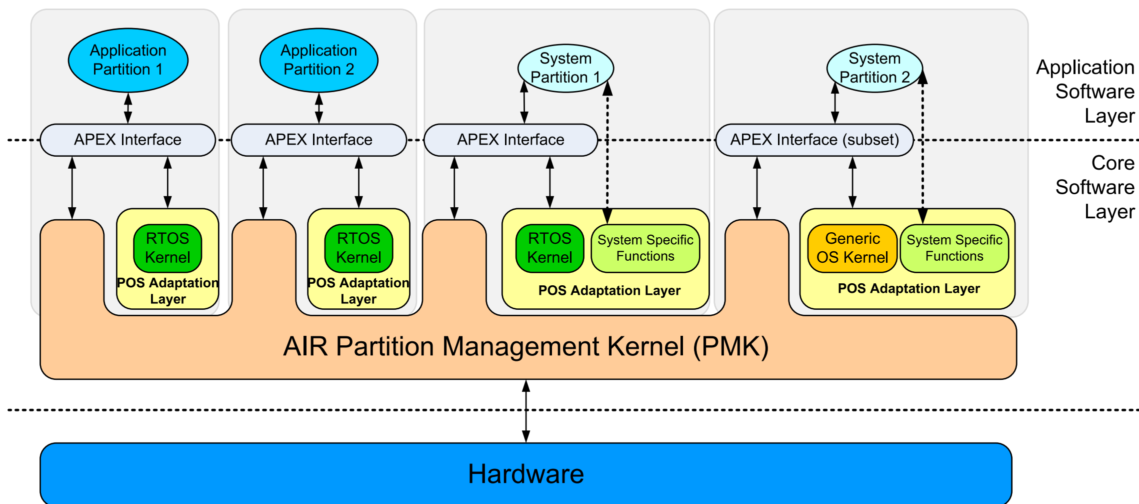


Figure 2.4: AIR architecture from [14]

2.6.2 APEX Interface

According to the ARINC 653 specification, a standard interface between the application software layer and the core software layer called APEX provides the basic services to the applications such as process, partition and time management, communication and health monitoring. APEX may be used by applications to invoke these basic services or bypassed by system partitions to access to specific functions of the POS.

In order to support AIR extra features such as mode-based scheduling, adaptability and online system updates, the APEX base interface has been extended with new services and primitives [34, 15].

2.6.3 Partition Management Kernel

The PMK is the base of the Core Software Layer, and may be seen as an hypervisor, transversal to all partitions. It is responsible for ensuring robust TSP properties, providing communication between different processes and partitions and handling low level interruptions.

Time Partitioning

AIR follows the two-level scheduling fashion proposed by the ARINC 653 specification to enforce temporal partitioning (i.e., that one partition does not interfere with each other's timeliness), as pictured in Figure 2.5. In the first level, partitions are scheduled and dispatched by the PMK, on a cyclic repetition (over a MTF) of a predetermined sequence of time windows — the PST. The PST is calculated offline and defines the amount of time assigned to each partition. In the second level of scheduling, inside each partition's time window, processes are scheduled according to the native POS scheduler.

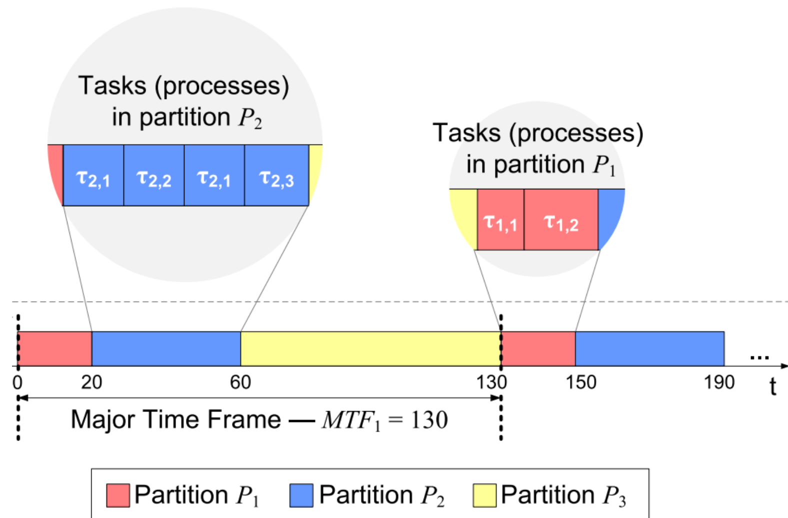


Figure 2.5: Two-level hierarchical scheduling

AIR support mode-based schedules, which means that the system can safely switch, in execution time, between multiple PSTs that are previously defined and integrated into the system.

Space Partitioning

AIR guarantees space partitioning between partitions by having separate addressing spaces for each one and not allowing an application from one partition to access memory addresses of a different partition. In AIR, this achieved through a modular approach based on a mapping between hardware-independent descriptors and the hardware-provided memory protection capabilities (e.g., its Memory Management Unit) [37].

Interpartition Communication

In conformity with the ARINC 653 specification [2], and as explained in 2.5.2, AIR provides two types of ports for accessing communication channels: *sampling ports* and *queuing ports*. Interpatition communication mechanisms are implemented by the PMK layer and made available to applications by the APEX interface.

The relation between interpatition communication and spatial partitioning implies the use of specific services that transfer the data from one partition to another without violating spatial segregation constraints. For that, the PMK must ensure memory protection, to guarantee that ports belonging to a given partition are not accessed by another one. Communication channels transfer data between ports, resorting to memory-to-memory copies mechanisms. Figure. 2.6 illustrates both types of interpartition communication mechanisms.

As pictured, applications access to the communication channels using the queuing

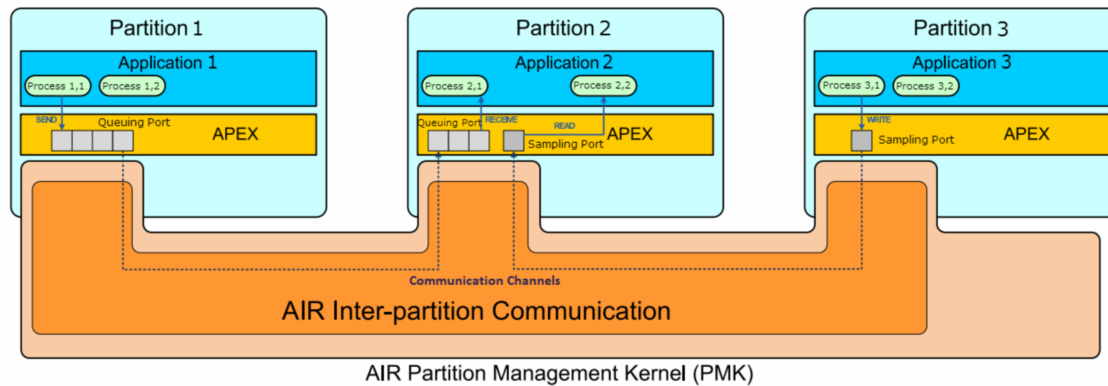


Figure 2.6: AIR Interpartition communication mechanisms

and sampling ports provided by the APEX and by invoking the primitives to write/read (in case of sampling ports) and send/receive (in case of queuing ports).

2.6.4 Mode Based Scheduling and Adaptability

ARINC 653 basic services rely on a a single PST used to schedule partitions, implementing the first level of the scheduling hierarchy. This PST is fixed and defined off-line at system integration time. AIR goes a step further and, instead of using a single PST, gives supports for the definition of a set of PSTs. The use of different PSTs allows to have different scheduling schemes that may be used to provide mode-based scheduling, adaptability and reconfiguration mechanisms, making the system more flexible and able to adapt its behaviour according to different needs. For instance, mode-based scheduling permits the system to adapt scheduling to correspond to a different mode of operation (e.g. take off, landing, etc.) allowing to make a more efficient use of the resources. Authorized system partitions may request scheduling switch by using the primitives provided by the APEX [34].

Furthermore, in [14] it is shown how this mechanism may help mitigating the rate of deadline misses in the presence of temporal faults of a single partition by assigning extra time to partitions experiencing timing delays

2.7 Summary

This chapter presented the basic concepts and related work that help understanding the work performed in the present thesis. First, it introduced real-time systems and their several classes, from soft-real time to safety-critical systems. It then gives a glance on some concepts related to safety and how it is guaranteed by means of validation, verification and certification of software. We then introduced the concept of hybrid models in opposition to homogeneous ones and the need for mechanisms that provide fault containment, from

which we highlight the TSP concept. The chapter closes with an overview over some emerging architectural standards from the civil and automotive worlds and with a detailed description of the AIR architecture. The next chapter will introduce sensor-based systems and the fault model considered in the present work.

Chapter 3

Cooperative Sensor-Based Systems

3.1 Sensor-based systems

A *control system* is a component or device added to one system in charge of controlling its behaviour, typically in an automated way. Examples of such systems are setpoint controllers in chemical plants, fly-by-wire systems on aircrafts or even router protocols that control traffic flow on the Internet. Other examples of emerging control systems are the ones used in autonomous smart vehicles, airplanes or robots, having high requirements in terms of confidence, reliability and safety levels [49]. Besides computation, control implies interaction with the physical world, for which control systems include not only computation units, but also sensors and actuators used to interact with the outside world or system [48]. The interaction between these components and the environment is a central concept in control and constitutes the *control system loop* [49], pictured in Figure 3.1.

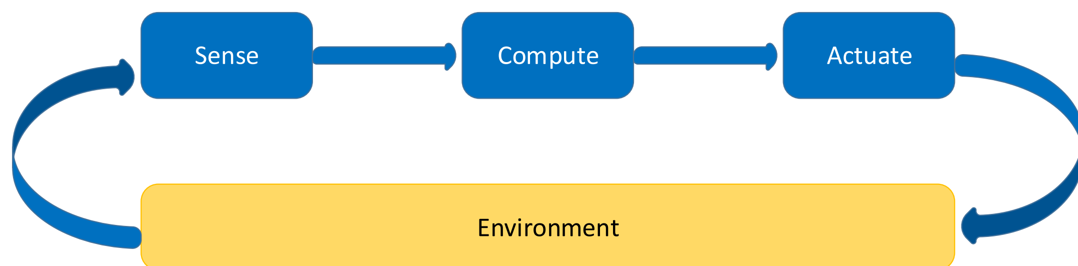


Figure 3.1: Basic Control Loop

Sensors are in charge of sensing and acquiring data whether from the environment or from the system itself and produce information to be consumed by some component of the system. Computational units receive data as input (from sensors or from another computing element), process it and output a result to another component (to an actuator or another computing element). In its turn, actuators consume information produced by a computing component and actuate on the system, whose consequences reflect on the

environment or on the system itself. This basic scheme abstracts the existing software, hardware and communication channels between components, being only focused on the data flow and the feedback between the control system and the actual controlled entity.

In this model, we assume that each component is used for a different *function*, serving a different purpose. A complete control loop (i.e., the flow of data from sensing to the actuation), implements what we call a *functionality*. As such, a functionality is constituted by multiple functions, each implemented by a different component. For instance, considering one vehicle: we may implement an Adaptive Cruise Control (ACC) functionality that adapts the car speed based on the relative position to the surrounding vehicles. In this case, the ACC functionality would be composed by one distance sensor, one component that extracts information about the position of the vehicle relatively to the surrounding vehicles, one component that implements a speed calculator function that receives the relative position as input from the position calculator and outputs a speed, and an actuator that receives the speed calculated by the speed calculator and applies it to the car. Figure 3.2 represents this basic functionality.

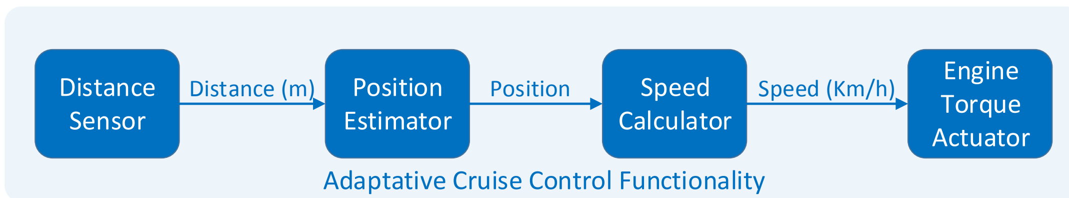


Figure 3.2: Example of Adaptive Cruise Control Functionality

3.2 Cooperative Systems

By cooperative systems, we understand a set of independent systems that actively exchange data and interact in order to help each other to achieve some common goal, such as improving their efficiency as a whole, or to realize some *cooperative functionality*. Cooperation between control systems has wide range of applications such as military or surveillance operations, wireless sensor networks and transportation systems [29].

Research about cooperating control systems is not new theme. For instance, in 1997, the PATH project [11] showed of a group of cars able to cooperate with each other in order to drive in platoon formation in a closed environment. Due to the technological innovations developed over the past decade, the research on this area has been a hot topic, with several new projects being developed regarding cooperation in transportation systems [40].

Cooperation can be used to improve the coordination among multiple independent systems. For instance, applied to the automotive industry, if a set of vehicles is able to cooperate with each other, they may execute cooperative functionalities that coordinate them in crossroads or highways, allowing to reduce traffic congestions, improve their fuel consumption and even avoid collisions or other hazards.

Cooperation may also be used to provide better and earlier sensor data to a system. For instance, if a set of vehicles is able to exchange data, one may be able to get information from the sensors of another vehicle as in a *remote sensor*, allowing it to detect a road obstacle before it is in the range of its local sensors.

However, when considering safety-critical systems, cooperation entails a major challenge regarding the communication between systems and the uncertainty that it arises which may put safety at risk, which we address next.

3.2.1 Communication

When considering cooperation between safety-critical systems, communication is a fundamental challenge that must be addressed. 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, when considering an open environment, there is a problem when dealing with cooperative vehicular applications. Due to mobility, cooperative vehicles must communicate with each other using wireless communication networks, which are known to be prone to interferences and much less reliable than wired networks. This uncertainty tends to be seen as increasingly difficult when considering that, in order to be suitable for use in safety-critical systems, these cooperative applications must satisfy strict safety requirements.

Our basic approach in this respect is to accept the fact that communication might not always be possible, devising solutions that allow to safely switch from a cooperative to non-cooperation behaviour. In other words, when communication is possible, then cooperation between systems may take place. On the other hand, if communication is not possible, no cooperation may take place. However, systems shall continue trying to cooperate again and they will be aware that it is not possible to actively cooperate. As such, we are interested in the possibility of exploiting the ability to communicate, while taking care of the fact that it might not be always possible. We thus focus on the aspects associated with communication and on ensuring safety despite such dynamic and uncertainty. When communication is not possible, vehicles shall operate as normal autonomous (non-cooperative) vehicles. Finding solutions to ensure a safe operation in these conditions falls out of the scope of the present thesis, since it is covered by state-of-the-art solutions.

3.3 Fault Model

Many current state-of-the art safety-critical systems such as automotive vehicles are built based on system models in which components are assumed to be simple enough to be proven reliable and safe in all domains.

In this thesis, we go beyond these assumptions by allowing unreliable components, able to perform more complex functions, to integrate such systems. As such, besides the described uncertainty that arises from unreliable communication between a set of cooperative systems, we also want to deal with local faults that may be experienced in local components such as sensors and computing units.

Since we are dealing with real-time systems, we want to cover faults not only in the quality of the produced data but also in the timeliness of that data. As such, one of the main motivations of the current thesis is to provide solutions for dealing with the effects of faults, which are likely to occur when adding complexity to the system. As such, regarding to local components, we assume the following:

Sensor components may experience faults. These faults may be both in the time and in the value domain. However, we assume that a basic subset of reliable sensors, enough to support a basic system's operation without compromising its safety, is available. This is mandatory to allow proving at design time the correctness and safety of a system. The required sensor properties, namely reliability, must be achieved by construction, for instance, using physical redundancy. Faults in the time domain include crash faults (i.e., when a sensor does not provide further output) and timing faults (i.e., when a sensor produces a late output).

Regarding to computing components, we assume they may experience faults in the time domain, but not in the value domain. As such, we assume that computing components may crash or get delayed. As with sensors, we assumed that a set of computing components are simple enough to be proven safe and correct by implementation and that this set is enough to ensure the safety of the system.

Wireless communications are assumed to be unreliable. Communication components will be assumed to suffer from failures in the time domain, in which messages may got lost or delayed. As such, no guarantees regarding the delivery of messages may be given. We do not consider value faults in communication, since we assume that current wireless communication standards already deal with (e.g., using mechanisms such as checksums).

Actuator components are assumed not to fail. We are concerned with faults affecting the correctness of perception, rather than the correctness of actuation. Reliability in actuation may be achieved using redundant actuators but, since failures in the mechanical parts of the actuators are outside the scope of this thesis, we will not focus on that. As such, we consider single actuators to be reliable, even if they are actually constituted of multiple unreliable redundant actuators.

Furthermore, we do not account for malicious faults in any of the categories. To con-

clude, we assume that all the communications internal to one system are based on a local network able to cope with high levels of reliability and timeliness requirements, which is possible to ensure with exiting state-of-the-art solutions; for instance, the CANELy architecture [36], combines the Control Area Network (CAN) [22] fieldbus communication infrastructure with a set of additional mechanisms to achieve high levels of reliability, availability and improved timeliness, providing the needed requirements.

3.3.1 Quality of data

The precision or quality of the data produced by a sensor may be subject to fluctuations. For instance, in case of an internal failure or due to some external conditions, one sensor may produce outputs with a lower precision than it would in a normal situation. Similar uncertainty may be present in the output of computing elements. As explained in 3.3, we account for timing failures in computing components, which may have impact on the quality of the produced output. For instance, considering a component in charge of doing image analysis, on which at each iteration the quality of the result is improved. If the component gets delayed and is only able to compute a reduced number of iterations, the quality of the produced result will be lower than if it was able to compute a bigger number of iterations.

Besides the variations affecting local components, the information exchanged with another systems is also subject to variations on its quality. Wireless communication links, being unreliable and prone to interferences, may suffer from losses and delays, being necessary to deal with this uncertainty and the potential degradation of the exchanged information.

In all the cases, since we are dealing with real-time data such as distance, speed or position that varies over time, it is need to account for impact that time has on the quality of this data. As such, crashes and timing faults must be reflected in the quality of the output produced by a component.

In this thesis we look to this problem with particular attention, since these are the kind of faults and uncertainties that we want to handle and contain. For that, we generalize the problems of failures and uncertainty affecting sensors, computing elements and communication, by attaching a quality value to each output produced by each of these components that we call *data validity*. Data validity represents how good or precise is the produced output, reflecting the uncertainty present in some components, both in the time and value domains. To reflect the consequences of the passage of time, data validity decreases with time, which makes, eventually, the output of a fault component overdue ensuring that delays and crashes are detected. The way this data validity is calculated is out of the scope of this work. Previous work addressing this challenge has been presented in [9, 25].

3.4 Summary

This chapter started by describing the fundamental concepts regarding control systems based on sensors and actuators. Later, it describes the benefits of cooperation between control and the challenges that it presents when considering safety-critical systems. We presented the fault model that we are considering for this work, focusing on the faults that affect each type of component. We presented the concept of data validity that represents the quality of the output produced by each component, which has a major role in the detection of faults and defining the system's state, as it will be described in the following chapters. Next, we will present the system's architecture, based on a hybrid model, that reflects the impact of cooperation and uncertain components in the system.

Chapter 4

System Architecture

As described, we are considering safety-critical systems, on which we want to include components that may rely on a different set of assumptions, have different requirements, criticality levels and provide different safety guarantees from each other.

We exploit the advantages of hybrid models to define a hybrid architecture able to integrate these components in the same system, which allows us to include a bigger variety of components and features when compared with state-of-the-art architectures, namely those that would have been discarded in approaches requiring all components to be proven safe and timely in design time. Uncertainty and safety must be managed in runtime to ensure that no catastrophic hazards take place. Next we describe how we manage these two opposite forces based on the definition of multiple levels of service and on the Safety Kernel component.

4.1 Levels of Service

Let us consider any kind of system: one can characterize how good or bad that system operates based on some metrics that measure its work. These metrics vary according to the considered system and the intended objective and define the *performance* of one system. For instance, considering an industrial plant in charge of producing some product: we may consider its performance as the absolute number of properly produced products. In this case, the best performance level will be the highest output of good products possible. Other possible metric could be the percentage of properly produced products, which would also take in account the defective ones. In the automotive or avionics industries, also different metrics may be used to measure and characterize the performance achieved by one system, such as the average speed, the time elapsed or the fuel consumption that takes one vehicle to get from a point A to a point B.

Components involved in the system control loop and the algorithms they execute define not only the performance achieved by one system, but also its safety level. Ensuring safety requires some kind of knowledge and assumptions that define particular aspects of

the control algorithm, such as how fast can a car go or how much distance must he maintain from the front vehicle, and also assumptions about the system model and external conditions. Components (and the functions performed by them) are, hence, proven safe at design time based on the given assumptions. The set of assumptions made, the resulting system and the performance that is able to safely provide defines what we call a Level of Service (LoS). As such, a Level of Service (LoS) defines a trade-off between the provided performance, the made assumptions and the safety level that it guarantees.

In typical safety-critical systems, the possible variables that influence the assumptions made in design time must be well known and all the possible faults that eventually may happen and that may put safety at risk are treated with some adequate fault-tolerant mechanism. However, since we want to introduce wireless communication and unreliable components as a mean to improve the provided performance, the increase in the number of possible faults makes impossible to predict and handle all of them in design time.

The way we handle this problem, is to consider that each functionality present in the system may operate in more than one LoS. First, for each functionality, we define a base LoS, henceforth called LoS 0, composed exclusively by components and functions suitable to be proven safe at design time and based on assumptions that are known to hold in a realistic scenario. This LoS is, in a way, the LoS used currently by state-of-the-art solutions, providing an ”always safe” base performance level, with no communication, cooperation, unreliable components or other sources of uncertainty impossible to handle in design time. Above this baseline LoS, we allow the definition of additional LoSs that exploits the advantages provided by cooperation features and complex components to provide an improved performance level. Improved LoSs may be designed and implemented based on more relaxed assumptions than the LoS 0, and may be then proven safe based on those assumptions. For instance, one may assume that the communication latency between two cooperating nodes will be lower than 50 ms, that the reading error of a sensor will be inferior than a given bound, or that a given component will always produce a timely output. Based on these assumptions (that may not always hold), the functions executed under this LoS may be proven safe.

Since these assumptions may not hold, it is necessary to, in runtime, monitor and check all the possible and relevant variations in order to detect violations in the assumptions and, if any of the assumptions of the LoS currently in use does not hold, then it is need to, somehow, switch the configuration of the system to the highest LoS possible whose assumptions are valid (which, as a last resort may be the LoS 0). The definition of multiple LoSs and the use of components with different assumptions reflects directly on the system's architecture, presented in the next section.

4.2 Hybrid Architecture

We design an architecture based on an hybrid model that divides the system in two main realms: the realm where are located the components that, by design, are proven to be safe, and whose assumptions are assumed to hold at any time and the realm where are located all the components whose complexity makes them unsuitable to proven safe by design, or whose assumptions in which they rely may not always hold at runtime.

For the sake of simplicity, for the next architectural scheme figure, we omit the environment and the interactions of the control loop between components, allowing to focus our attention on the components of the system and its architecture. As such, starting from the basic control loop presented in Figure 3.1 that contains the basic interaction flow between the environment and the system, we must make explicit that in fact several control components (i.e., sensors, actuators and computational units) may exist, each being used to embody a different need of the system, or just a redundancy mechanism.

As mentioned, we divide the system architecture in two realms, allowing to introduce and exploit complex components in order to provide additional LoS with an increased performance. To reflect this directly on the architecture, we divide the two parts by an *hybridization line* that clearly separates the trusted components (and that provide the baseline LoS for the untrusted ones that provide the higher LoSs based on more complex functions and cooperation features. Figure 4.1 represents this architectural division. As it

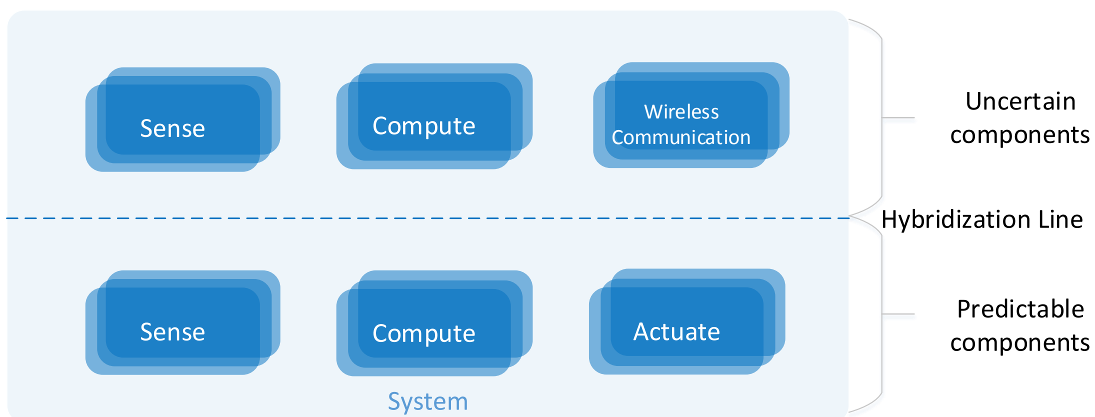


Figure 4.1: Hybrid Architecture

is possible to notice, we do not put actuators above the hybridization line, since, as mentioned before, we assume they are, somehow (e.g. by using redundancy mechanisms), reliable and predictable. Furthermore, we added a new type of components, representing the components used for wireless communications. since wireless communications are not reliable, we place these components exclusively above the hybridization line. For the remaining classes of components (sensors and computational units) as stated, we account for them both under and above the hybridization line.

We further emphasize that 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, despite the possible misbehaviour of the ones above the line, and that no fault on this realm may jeopardize the execution of components under the hybridization line.

For instance, taking a simpler example of the ACC mentioned in 3.1, we may want to improve the LoS 0, by creating an extra LoS that uses not only information from the local (and reliable) sensors, but also information from remote sensors of surrounding vehicles. Furthermore, due to the possible variation in the amount of information coming through the remote sensors, which may require more or less calculations, the execution time of the function in charge of calculating the speed is unpredictable. Figure 4.2 represents this scenario, with two LoSs for the same functionality.

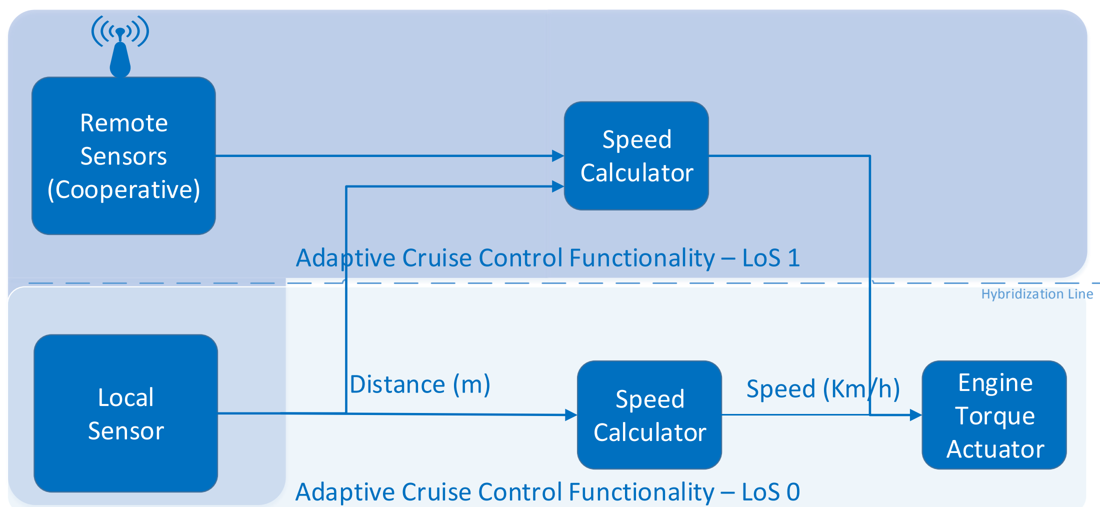


Figure 4.2: ACC functionality with multiple Levels of Service

As it is possible to notice, the LoS 0 is composed by the local sensor, the basic speed calculator and the engine torque actuator, for which it will execute in a predictable way, since it relies exclusively on components under the hybridization line. On the other hand, the LoS 1, depends on remote sensors, with unpredictable timeliness and validity, for which the speed calculator component from this LoS may be unable to produce a timely output to be consumed by the engine torque actuator. As explained, in this example there are two components in charge of producing one speed value to be consumed by the engine torque actuator. In fact, this situation may happen not only with actuators, but also with computing elements; this is because we want one component to be independent from the components that provide him input, allowing to achieve modularity and to have multiple components providing the same *function* with a different implementation. In this case, the different implementations would all be executing and producing different possible

outputs, each with a different performance or quality associated. From the available outputs, one must be chosen to be consumed by components using that function. This choice must be transparent and independent from the consuming components.

It is also important to note that one component may be used in the provision of more than one functionality, allowing, to add new functionalities that reuse already existing components. By way of illustration, the existing distance sensor could also be used to provide data to the Adaptive Headlight Control functionality that adapts the intensity of the headlights based on the distance to the front car, as pictured in Figure 4.3

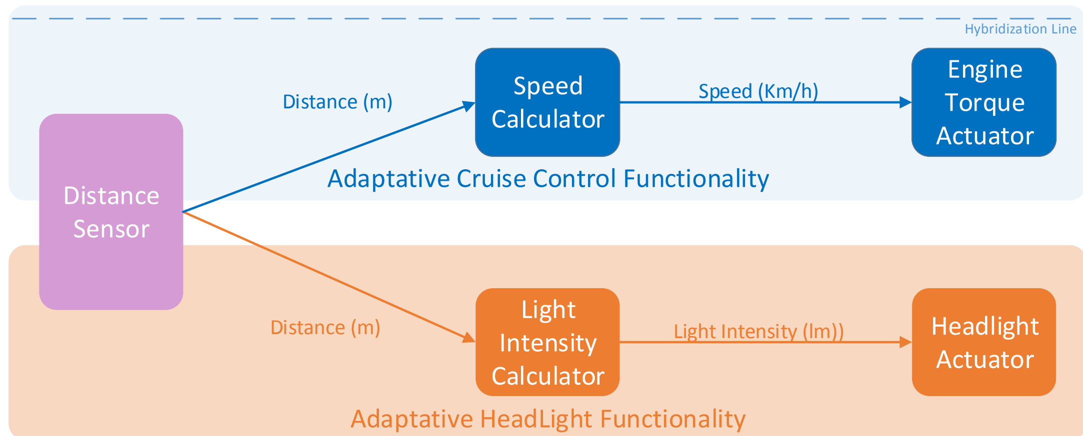


Figure 4.3: One component used in the provision of multiple functionalities

Furthermore, components may be able to change their *mode of operation*, based on some externally adjustable parameter, allowing to have one single component able to adapt its own behaviour to different situations, without the need of having different components for the same task. Each mode may be used to provide the same function with a different algorithm that may be used to provide a different performance and safety guaranties, allowing to adapt the component to some observed constraint or condition. Figure 4.4 exemplifies this situation, where the speed controller component may be, somehow, controlled from outside, so that depending on some external parameter, it adjusts its execution mode.

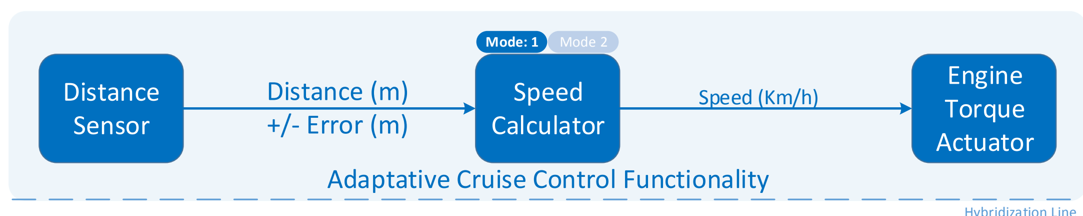


Figure 4.4: Reconfigurable Component with multiple modes of execution

As explained, we are moving away from a system with static behaviour to a dynamic

one, with different LoSs, unpredictable components used to improve the performance of another and adaptable components that change their behaviour based on some external parameter. In order to deal with the increased complexity and with all the possible system configurations, we introduce an additional component to the system — the Safety Kernel — that we present next.

4.3 Safety Kernel

When introducing new sources of uncertainty to the system, it becomes fundamental to manage the system's configuration, providing the means to adapt the system's behaviour depending on some observed runtime information. To do so, and in order to deal with possible faults and ensure that all safety requirements are always satisfied, it is necessary to add to the architecture a proven-safe component in charge of managing the system configuration, by checking if the safety requirements for the current LoS are being respected and, if needed, adjust the LoS to a level in which all safety constraints are met.

This is done by the *Safety Kernel*, which is in charge of monitoring the execution of system's components and controlling the LoS in which functionalities operate, forcing the system to adapt its behaviour in order to meet runtime constraints, avoiding any major consequence in terms of safety. This adaptation shall be done based on some pre-established conditions and reflected in the system with a LoS switch of the affected functionalities. Due to its role in ensuring safety, the Safety Kernel must be a reliable component, proven in design time to behave correctly and timely in all circumstances, or, in other words, be placed below the hybridization line, meaning that the Safety Kernel will always operate correctly with respect to the assumed system model. Figure 4.5 represents the complete system architecture with the inclusion of the Safety Kernel component.

As pictured, the inclusion of the Safety Kernel creates a new information loop in the system; based on some runtime data coming from the system components, the Safety Kernel checks if the safety requirements are being fulfilled and, if needed, triggers a system reconfiguration back to the system, forcing it to adapt its LoS to the current constraints.

In addition, for each function implemented redundantly by more than one component, each one producing an output, the Safety Kernel must act as an intermediary between the producer and consumer functions, by choosing and forwarding one (e.g., the one with highest quality) of the available outputs to the consumers.

We envision a generic Safety Kernel, providing services that are independent of the specifically considered components and functionalities of the system, being independent and unaware of any specific application semantics. To emphasize this fact, we add an additional line to the system architecture — the *semantics line* — under which no application semantic exist. This division highlights the different between the *functional components* of the system — the components used in the provision of actual functional-

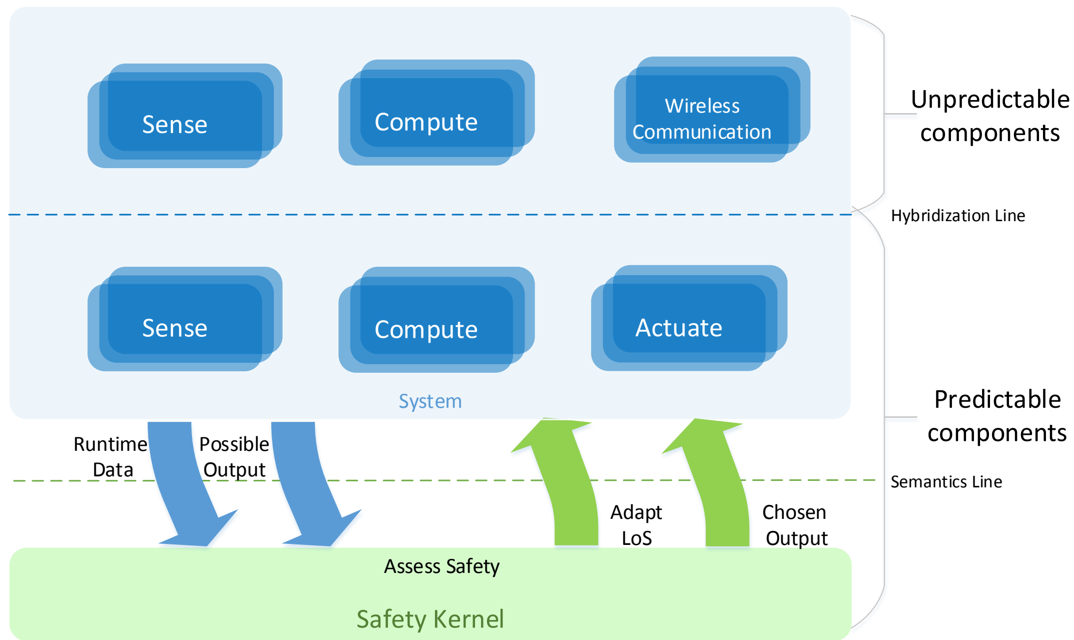


Figure 4.5: System Architecture with Safety Kernel component

ities — and of the Safety Kernel, which is rather used to provide services to the system itself. The semantics independence makes the Safety Kernel modular and facilitates its use and integration in existing systems and eases its development and validation process. This is why we separate the Safety Kernel from the remaining components of the system, making clear this independence and separation of concerns. Still, the Safety Kernel must be aware about safety constraints that need to be satisfied in each LoS. These safety constraints, defined at design time, must take into account all the functionalities to be provided, their interdependencies and other issues that are important when making a safety analysis, and must be stored in a database of rules that is made part of the Safety Kernel.

4.4 Summary

This chapter described the system's hybrid architecture. We take advantage of hybridization to support the coexistence of components with different criticality levels and that provide different guarantees in terms of predictability. Furthermore, hybridization allows us to define functionalities with multiple levels of service, each one relying on different assumptions, using different components and providing a different performance level for the functionality they implement. Since higher levels of service rely on more relaxed assumptions, the performance they are able provide is obviously higher than the ones that rely on tighter assumptions. Due to the uncertainty introduced into the system by unpredictable components, we devise the introduction of a new safe component called Safety Kernel, in charge of checking whether the safety requirements for the current LoSs are

being fulfilled and make the necessary adjustments in the system's configuration to avoid any possible hazard. The next chapter describes in detail the role and architecture of the Safety Kernel, as well as its integration in the existing architecture.

Chapter 5

Safety Kernel

This chapter gives a detailed description of the Safety Kernel component. First, we give a brief overview over the role of the Safety Kernel, its requirements and its relation with the remaining system. Then, we give a detailed description about the multiple components that compose the Safety Kernel and the support required to integrate the Safety Kernel into a system.

5.1 Overview

The main task of the Safety Kernel is to, periodically, evaluate the safety constraints for the current LoS and to make the necessary adjustments on the system's configuration, so that it meets the observed runtime safety constraints. This safety evaluation is done by comparing two different types of information: safety rules — static and defined at design time, based on the safety analysis and assumptions of the components of each LoS — and runtime data — generated in execution time by each of the relevant components in which faults or a variation on the quality or timeliness of the data produced may have impact on safety.

Furthermore, we emphasised the need of using the Safety Kernel as a mediator between components that implement the same function and the components that use the output of that function.

5.2 Safety Kernel design issues

5.2.1 Runtime information

According to the considered system model, we assume that components may be affected by value faults that reflect on the quality of the provided data and by timing faults that affect the timeliness of the data they produce and that flows to other components.

Runtime information refers to the data used to assess and determine the state of the system that must be collected from the system's components whose output quality or

timeliness is subject to changes. As such, every component whose output validity or timeliness is subject to changes shall send the necessary data to be assessed by the Safety Kernel. To collect this information, the Safety Kernel implements a set of interfaces to allow the needed information to be sent by the functional components. These interfaces are known by the designers of functional components and must be used whenever necessary for the sake of collecting data validity or timeliness information. The details on these interfaces are provided ahead in 5.3.3.

The process of collecting this data is continuous and periodic. That is, the Safety Kernel will be periodically collecting this data and analysing it in order to determine violations of the safety requirements both in terms of the validity and timeliness of the data.

5.2.2 Design time information

As explained, based on the safety analysis done for each LoS of each functionality, safety requirements are derived and allocated to system components that explicit the requirements that a given component needs to be executed safely.

As such, for each functionality, the Safety Kernel must be aware of the safety requirements needed for that functionality to be provided safely. Hence, the *Design Time Information* consists on the information that establish the conditions for functional safety assurance in each LoS of each functionality. These safety requirements are stored in the form of rules in a database accessible to the Safety Kernel and refer the bounds of the validity and timeliness data provided, at runtime, by components whose output quality must be monitored by the Safety Kernel.

The Safety Kernel, whose role and operation do not depend on the semantics of functionalities, only evaluates the rules using an appropriate rule evaluator engine, which is generic and not developed for particular rules or functionalities. For example, if a certain LoS of a cooperative functionality requires that variable V1 is lower bounded by some value (e.g., $V1 < 0.9$), then the Safety Kernel will just have to know the bound, the run time value of V1, and the comparison that needs to be done, in order to determine a boolean value indicating if the LoS is sustainable or not. The specific meaning of the bound is irrelevant from the perspective of the Safety Kernel. Nevertheless, the design of the Safety Kernel requires the specification of the rules format and their interdependencies. The complexity of the rules can vary from a collection of independent checks to a sequence of interdependent checks of data validity and timeliness information.

Furthermore, the design time information must provide to the Safety Kernel the information about how to parametrize components whose mode of operation may be externally adjusted and information about which components implement the same function redundantly in a different LoS and to whom one of their results must be forward.

As such, design time information, coupled with the runtime information from each

component, should support the following functions:

- Determine the maximum LoS for each functionality in which the local system is able to operate.
- Determine how to parametrize reconfigurable components according to the current local and cooperative LoS of each functionality
- Know which components implement the same function and to whom their result must be forward.

5.2.3 Adapting the Level of Service

Based on the safety assessment made by comparing the runtime data against the safety requirements, the Safety Kernel must adapt the LoS of the functionalities under its supervision.

When talking about strict local functionalities, which do not depend on any cooperation with external systems, the LoS to be enforced is only dependent on the evaluated local safety requirements.

However, when we shift for cooperative functionalities, the LoS that must be enforced may depend not only on the assessment of local components but also be depend of information received from other systems. Due to the cooperative nature of the performed functionality, the Safety Kernel may have to consider the maximum LoS that is possible to achieve in other cooperating nodes. The LoS that must be actual enforced depends on the specific functionality and on how it is designed. There are basically two options:

- the functionality requires that all involved vehicles are coherently executing the functionality in the same LoS
- the functionality assumes that each vehicle may execute the functionality in a different LoS (possibly knowing in which LoS are the other vehicles executing the functionality).

In the first case, the Safety Kernel must take into account the LoS information received from other vehicles before locally enforcing a LoS. In the second case, the locally enforced LoS will be the one that is determined upon the local assessment of safety requirements.

For example, in the case of a cooperative functionality where its LoS is based on an agreement between the participating nodes, the LoS enforced by the Safety Kernel would be the highest that can be maintained across all the nodes. Whenever the LoS has to be degraded in a certain node, (e.g., because this node is experiencing some failure), this change must be communicated to the other participating nodes and reflected on their

own enforced LoS. Likewise, in the opposite situation, in which a certain node is able to operate at a higher LoS, this information is propagated to the other participating nodes and, if they can all perform in this higher LoS, then the locally enforced LoS is raised.

After determining which LoS adopt, the Safety Kernel must enforce it on the local components used on that specific functionality by means of reconfiguration so that they adapt their behaviour to the new LoS. In other words, upon a LoS change in any functionality, the Safety Kernel must know which components must be adapted and which parameter must be used to enforce that change. In addition, whenever there are multiple components implementing the same function, the Safety Kernel must forward the output that fulfils the requirements corresponding to the chosen LoS.

All these operations must be done in a timely manner. Consequently, a change in the mode of execution of a specific component shall happen within some known timing bounds. If it does not, that shall be detected by the Safety Kernel that, in last case, may continue forcing a downgrade of the LoS, reaching, if needed, the LoS 0. However, how that change is performed is dependent on the applications themselves, being out of the scope of the Safety Kernel.

5.3 Architecture

5.3.1 Components of the Safety Kernel

To perform its role, the Safety Kernel exchanges information with other components of the system. The exchanges with different types of components embody different aspects of the Safety Kernel's operation, like receiving data validity or sending commands for controlling the mode of operation of functional components. For this reason, we see the Safety Kernel as a set of components, with clearly defined and separated concerns, which are combined to verify and guarantee the operational conditions for safety. For the Safety Kernel to be relied upon for the provision of safety-critical functionalities, its components have to be proven to exhibit the necessary reliability and timeliness in design time by means of software verification and validation. Figure 5.1 schematizes these components and their interactions. The figure also includes two example functions with whom the Safety Kernel exchanges information. For the sake of the example, we consider that function A has two implementations and produces an output to function B and that the behaviour of one of those implementations may be adjusted from the outside.

For the safety assessment and system's adaptation, the Safety Kernel collects the data validity or timeliness information made available by the monitored components, assesses it and adapts the LoS of functionalities by reconfiguring components according to predefined rules. The components of the Safety Kernel involved in this control loop are the Rules Database, the Local LoS Evaluator and the Safety Manager.

For every cooperative functionality, since it be may need to agree some LoS with

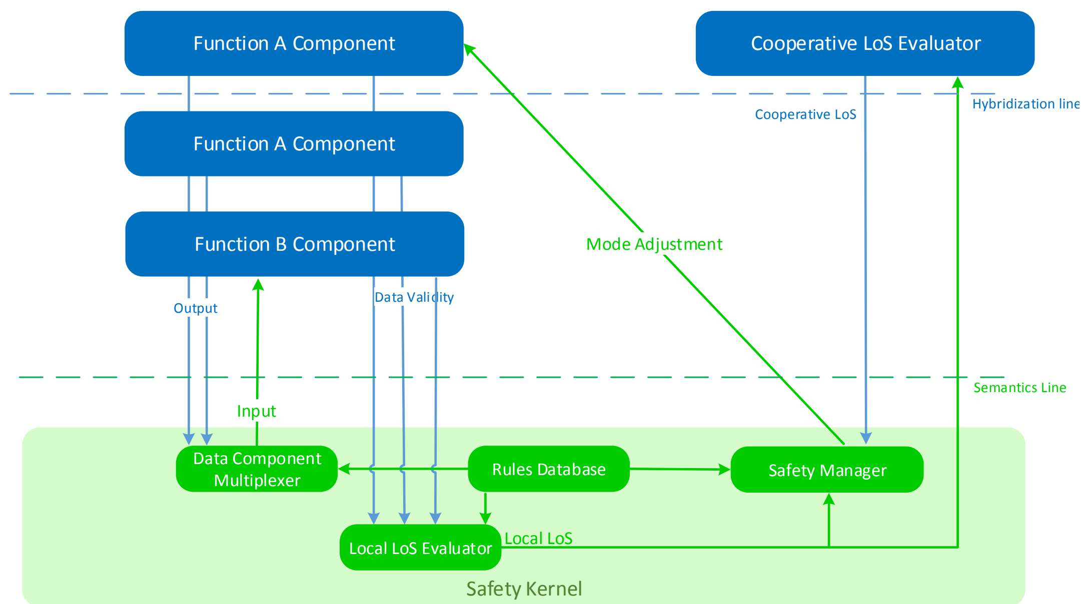


Figure 5.1: System components overview and interaction

other cooperating nodes, the Safety Manager uses the result produced by the Local LoS Evaluator and, possibly, the result from an external component called Cooperative LoS Evaluator, in charge of agree on a LoS with other nodes. The Cooperative LoS Evaluator is, however, a component external do the Safety Kernel, which will be described in section 5.3.2.

For the multiplexing task, the Safety Kernel assess the possible outputs and forwards one to other components. The two component of the Safety Kernel involved in this process are the Rules Database and the Data Component Multiplexer.

This following sections describe in detail these components.

Rules Database

The Rules Database contains all the knowledge the Safety Kernel needs to execute the tasks mentioned before. All this knowledge is derived in design time and loaded in the Safety Kernel's rule database upon the integration of the components in the system. Since the Rules Database gives support to different tasks of the Safety Kernel, we define three categories of rules, each giving support to a different task:

Safety Rules - Rules used to assess, at runtime, under which LoS a specific functionality may operate. The assessment is done by comparing the data validity and timeliness information with respect to the bounds expressed in the safety rules. These rules are accessed by the Local LoS evaluator. As mentioned in 5.2.2, the format of the rules must support the aggregation of individual rules. A set of safety rules must be

defined for each LoS of each functionality. One basic example of a rules grammar that support the aggregation of individual rules, in which each individual rule is a simple comparison between two values could be defined as in Snippet 5.1. Where

$$S \longrightarrow (S)\ op\ (S) \mid s$$
$$op \longrightarrow \wedge \mid \vee \mid \oplus$$
$$s \longrightarrow Value\ comparator\ Bound$$
$$comparator \longrightarrow < \mid > \mid = \mid \neq \mid \leq \mid \geq$$

Snippet 5.1: Safety Rules

$\wedge$, $\vee$ and $\oplus$ represent the binary operators AND, OR and XOR respectively. $Value$ would indicate the value of the data validity to read, and $Bound$ would be a number (e.g., a float or an in), with whom $Value$ would be compared using $comparator$.

Configuration Rules - Rules used to define the parameters that must be set on each reconfigurable components according to the LoS of each functionality. This is, therefore, the set of rules is used by the Safety Manager component to reconfigure the system. As such, a basic configuration rule would have the structure presented in Snippet 5.2.

$$C \longrightarrow c*$$
$$c \longrightarrow component,\ param\ ;$$

Snippet 5.2: Configuration Rules

Where 'param' indicates the parameter (e.g., an integer, float, or string) to be written, 'component' indicates where (e.g., socket or port) that same parameter must be written and '$*$' represents zero or more of the objects to its left. Besides this information, each rule shall include as well the necessary information to infer under which conditions those configurations shall be used (e.g. in which LoS of which functionality). These conditions may be express with simple rules with similar structure as the Safety Rules that are coupled with each defined configuration. This basically allows to, based on the state of the system, reflected by the LoS of each functionality, define which components must be reconfigured and with which parameters.

Multiplexing Rules - Set of rules that give support to the Data Component Multiplexer component, defining which components implement similar functions, and to which components one result must be forward. As such, for each function with multiple implementations, a set of multiplexing rules must be defined. One basic example of a grammar for such rule is represented in Snippet 5.3.

$$M \longrightarrow P;\ C\ ;$$
$$P \longrightarrow p,\ p\ (,p)*$$
$$C \longrightarrow c\ (,c)*$$
$$p \longrightarrow producer$$
$$c \longrightarrow consumer$$

Snippet 5.3: Multiplexing Rules

Where 'producer' and 'consumer' indicate, for instance, a socket or port to read or write respectively, and $*$ represents zero or more of the objects to its left (inside brackets).

Furthermore, for each producer, additional information regarding in which conditions its output may be used may be included. These conditions may be expressed with rules with a similar structure as the Safety Rules.

This static information, defined in design time and stored as simple and generic rules, is the base support for all the operations and decisions made by the Safety Kernel.

Local LoS Evaluator

The Local LoS Evaluator is the first component of the Safety Kernel's control loop. The role of the Local LoS Evaluator is to evaluate and assess the data validity and timeliness information of the monitored components against the Safety Rules. Based on this assessment, the Local LoS Evaluator determines the maximum LoS at which each functionality is able to safely perform from the perspective of the local node. This result is then made available to the Safety Manager and to the Cooperative LoS Evaluator components, discussed later. The basic behaviour of the Local LoS Evaluator is illustrated in Figure 5.2.

Periodically, the Local LoS Evaluator reads the validity and timeliness data sent by the system's components and, for each functionality being executed in the system, compares it against the safety rules defined for the current LoS. If the all the rules are satisfied, the Local LoS evaluator tries to evaluate the safety requirements for the next LoS, repeating this loop until some rule in some LoS is not satisfied, returning the highest LoS found

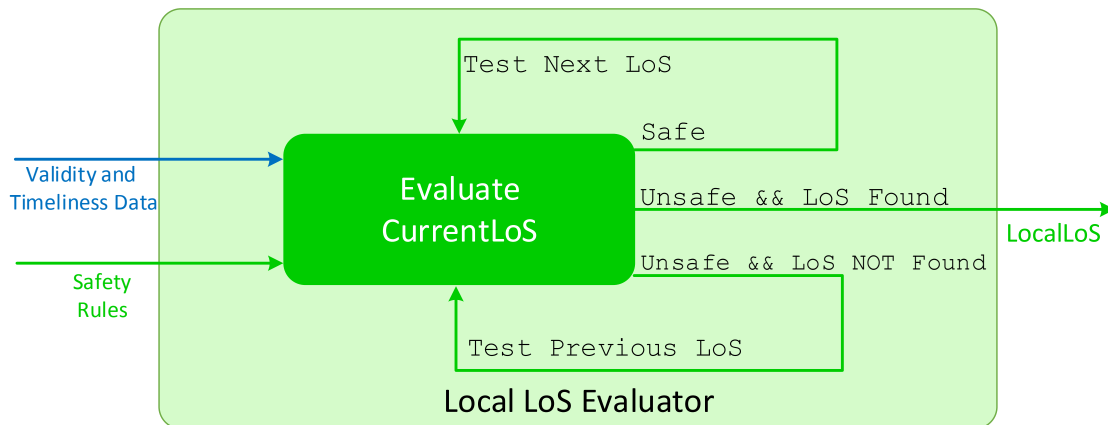


Figure 5.2: Local LoS basic behaviour

whose safety requirements are satisfied. If, in the other hand, the first assessment is negative, the Local LoS evaluator will evaluate the safety requirements of the previous LoS, repeating the loop until finding one LoS whose safety requirements are satisfied (which, in a worst case scenario, will be the LoS 0). Given that the Local LoS Evaluator executes in a timely manner, it is possible to know, in design time, how much it will take to evaluate, in a worst case scenario, from the lowest LoS to the highest (or vice-versa). This amount of time must be taken in account when designing the control algorithms and the safety margins established in their safety rules.

Safety Manager

The Safety Manager is the component that, based on the LoS of each functionality, enforces a reconfiguration on the components of the system. The Safety Manager supplements the operation of the Local LoS Evaluator, by considering not only its output, but also the output of the Cooperative LoS Evaluator in the decision of the effective LoS to adopt.

The effective LoS is then enforced by the Safety Manager by reconfiguring the system's components, making them adapt themselves to the new system's configuration. Figure 5.3 represents the basic behaviour of the Safety Manager.

As pictured, the first operation of the Safety Manager is to, based on the inputs received from the Local and Cooperative LoS Evaluator, determine which configuration should be enforced in the system. The actual configurations are specified and stored in the Configuration Rules that shall define not only the actual configurations, but also under which conditions a given configuration shall be applied.

The reconfiguration of components is necessary to change their execution mode in response to a LoS change of any functionality. As such, after determining the effective

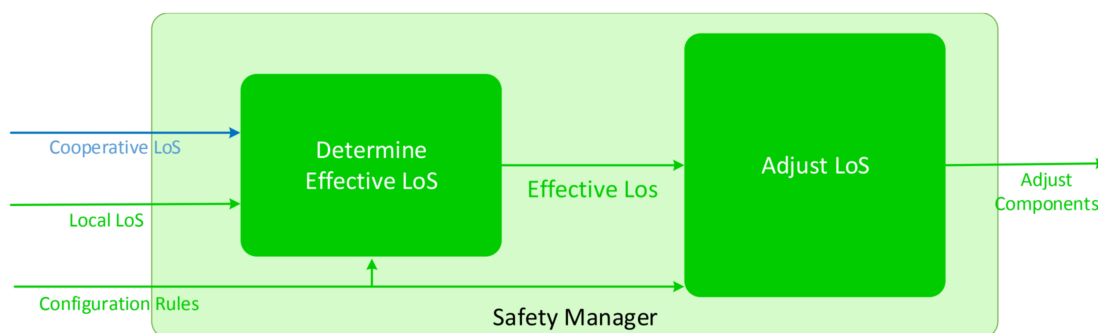


Figure 5.3: Safety Manager basic behaviour

configurations to apply, the Safety Manager propagates the needed reconfigurations by parametrizing the needed components with the new configuration.

The actual reconfiguration and adjustment mechanisms are then executed within each component. The Safety Manager just has the responsibility of triggering these changes on the right components. If, by any reason, some component does not adapt its behaviour that shall be eventually detected by the Safety Kernel and may have as a consequence a new reduction in the LoS of functionalities (in last resort, it may be necessary to decrease the LoS of the affected functionalities to 0).

Data Component Multiplexer

As previously explained, one function may be implemented by more than one component, producing different results in terms of quality and with different timing guarantees. The result of a function with multiple components should always correspond to the output of the component that better satisfies the safety requirements of the LoS of all functionalities that make use of it.

To avoid any time penalty whenever a component misses a deadline or produces a result with low quality, all the implementations of a function are executed simultaneously. For example, considering two components that perform the same function, one with a complex, and unpredictable algorithm and the other with a simple, and therefore predictable, algorithm. When these two implementations are concurrently in execution, both of them produce results with different qualities and timeliness. Whenever possible, the output from the more complex component shall be used, since it provides a better quality and may be used to provide a higher LoS. However, if due to same failure that result is not available, than the system shall use the output produced by the simple component (that never fails). As such, by having both of the components executing, it is possible to ensure that a valid output (produced by the simpler implementation) will always be available for use. If a better and valid result (produced by more complex implementations) exists then this should be used.

The task of the Data Component Multiplexer component is to decide which output will be the result of one given function, discarding the others. Figure 5.4 shows the basic behaviour of the Data Component Multiplexer. As pictured, for each function with

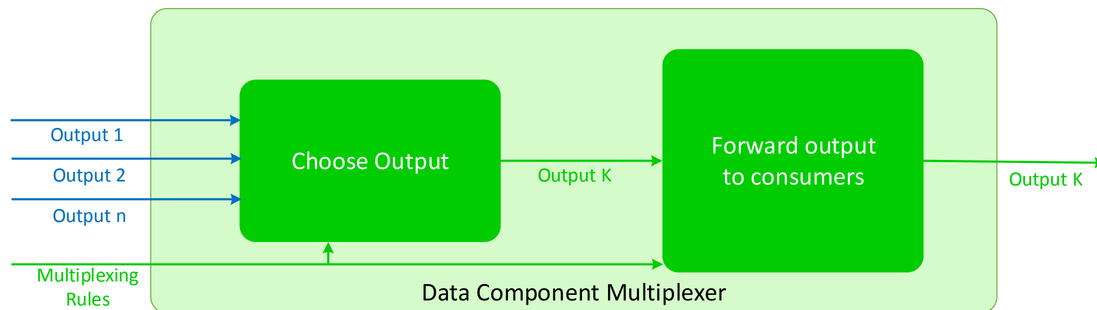


Figure 5.4: Data Component Multiplexer basic behaviour

multiple implementations, the Multiplexer reads all the produced values (represented as Output 1..n). Based on the timeliness and quality of them and on some possible additional conditions stored in the Multiplexing Rules it chooses one of the outputs (represented as Output k). This value is later forward to all the components using this result as input. All the knowledge needed to the operation of the Data Multiplexer (e.g., which components implement the same function or to whom their result must be forward) is stored in the Multiplexing Rules database.

This component plays a crucial role to achieve safety in the described architecture, allowing to mask failures in complex components by using the result of another component. Functions that use the produced output forwarded by a multiplexer are independent from the function before it. The Data Component Multiplexer does not apply to functions with a single implementation.

5.3.2 External Components

This section describes a component external to the Safety Kernel — the Cooperative LoS Evaluator — that plays an important role in its operation when dealing with cooperative functionalities. Since it is external to the Safety Kernel, its description is deliberately not detailed, rather consisting of the knowledge the Safety Kernel has of these components.

Cooperative LoS Evaluator

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

The operation of the Cooperative LoS Evaluator and the algorithms it uses to exchange information with other vehicles is not dealt as part of the Safety Kernel. In fact,

it is possible that a different Cooperative LoS Evaluator is defined for each cooperative functionality. Since this component is defined as a complex component (because it is not possible to guarantee in design time that communication with other vehicles is always possible), the solutions concerning what it does are varied and depend on what may be more desirable for some functionality.

A possible approach for the operation of the Cooperative LoS Evaluator is to have it producing a Cooperative LoS based on an agreement between the participating nodes. This LoS would correspond to the lowest LoS that is possible at every participating node. In this case, this LoS would become the Cooperative LoS that is sent to the Safety Manager and used to determine the necessary system reconfigurations . Nevertheless, when an agreement cannot be achieved, or if a given cooperative functionalities does not require an agreement on the LoS, the Cooperative LoS Evaluator may remain silent and not produce any value.

It can be added that, because the Cooperative LoS Evaluator is application dependent, it could in fact have multiple modes of operation and its complexity can go much further than what is explained here. 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. Figure 5.5 represents a high-level view of a possible Cooperative LoS Evaluator. Since its implementation is application dependent, the scheme does not focus on its internal behaviour, but rather on the interaction it may have with the Safety Kernel components. As pictured, the Cooperative LoS Evaluator uses as input the

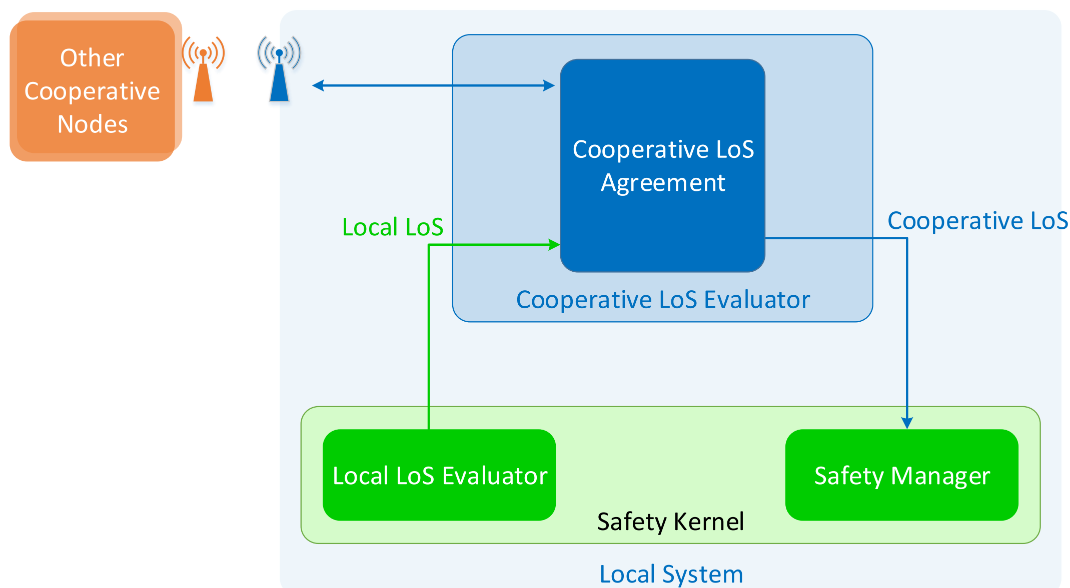


Figure 5.5: Cooperative LoS Evaluator basic behaviour

output of the Local LoS evaluator and, based on some information exchanged with other

cooperating nodes, produces a result sent to the Safety Manager. The algorithm used to decide which LoS is decided outside of the scope of the Safety Kernel.

5.3.3 Interfaces

In order to support the action of the described components, we defined a set of standard interfaces to allow the interaction between the components of the system and the Safety Kernel. The primitives that constitute these interfaces are provided in READ—WRITE pairs, constituting an information flow channel between one writer and one or more readers. Besides being non-blocking and atomic, these primitives must guarantee the following:

READ calls - the value that is read is the one written in the last invocation of the corresponding WRITE call; until overwritten, a value can be read multiple times and/or by multiple readers

WRITE calls - the provided value overwrites the value previously provided by the same writer

In each pair of primitives, one of the primitives shall be used by the Safety Kernel whereas the other shall be used by software components external to the Safety Kernel.

The way these information flow channels are implemented shall be abstracted by an underlying layer in charge of providing communication mechanisms between the components of the system and shall be transparent to them. It is responsibility of this layer to ensure, by whichever means necessary that READs are consistent with the latest WRITE. Furthermore, READ primitives shall allow readers to know if the read value is still timely valid or not (i.e. that the time elapsed since it was written is lower than a predefined delta/timeout). Table 5.1 lists these interfaces, which we describe next.

Data Validity Interface

This is the interface used by functional components to feed the Local LoS Evaluator component with the data validity regarding their output. These components must, therefore, be able to, in runtime, send this data to the Safety Kernel to be checked and assessed so that when it executes it can determine the LoS at functionalities are able to perform. The primitives used to support these operations are the following:

WRITE_DATA_VALIDITY This primitive, to be used by the applications, allows any component to send its data validity to be checked and evaluated by the Safety Kernel;

READ_DATA_VALIDITY This primitive, to be used by the Safety Kernel's Local LoS Evaluator, allows the Local LoS Evaluator to read the data sent by components using the previous primitive

Table 5.1: Interfaces for Safety Kernel support

Interface	Primitive	Involved Components
Data Validity Interface	WRITE_DATA_VALIDITY	Functional Components
	READ_DATA_VALIDITY	Local LoS Evaluator
Cooperative LoS Interface	WRITE_LOCAL_LOS	Local LoS Evaluator
	READ_LOCAL_LOS	Cooperative LoS Evaluator
	WRITE_COOPERATIVE_LOS	Cooperative LoS Evaluator
	READ_COOPERATIVE_LOS	Safety Manager
Mode Switch Interface	WRITE_ENFORCED_MODE	Safety Manager
	READ_ENFORCED_MODE	Functional Components
Data Component Multiplexer Interface	WRITE_APP_OUTPUT	Functional Components
	READ_APP_OUTPUT	Data Component Multiplexer
	WRITE_APP_INPUT	Data Component Multiplexer
	READ_APP_INPUT	Functional Components

One example of a workflow is pictured in Figure 5.6. In this diagram and in those which follow, the dashed arrows inside the Operating System represent the provided communication channel, as described in Section 5.3.4.

Cooperative LoS Interface

This interface is used by 3 components and gives support to the LoS management done by the Safety Manager and by both Local and Cooperative LoS Evaluators. It allows the Local LoS Evaluator to send its output to the Cooperative LoS Evaluator and also for the Cooperative LoS Evaluator to inform the Safety Manager of the Cooperative LoS. The primitives used to support these operations are the following:

WRITE_LOCAL_LOS This primitive is used by the Local LoS Evaluator to make available the information about the maximum Local LoS that is possible to safely perform to the Cooperative LoS Evaluator;

READ_LOCAL_LOS This primitive is used by the Cooperative LoS Evaluator to read the Local LoS written using the previous primitive;

WRITE_COOPERATIVE_LOS This primitive is used by the Cooperative LoS Evaluator to inform the Safety Manager of the Cooperative LoS;

READ_COOPERATIVE_LOS This primitive is used by the Safety Manager to read

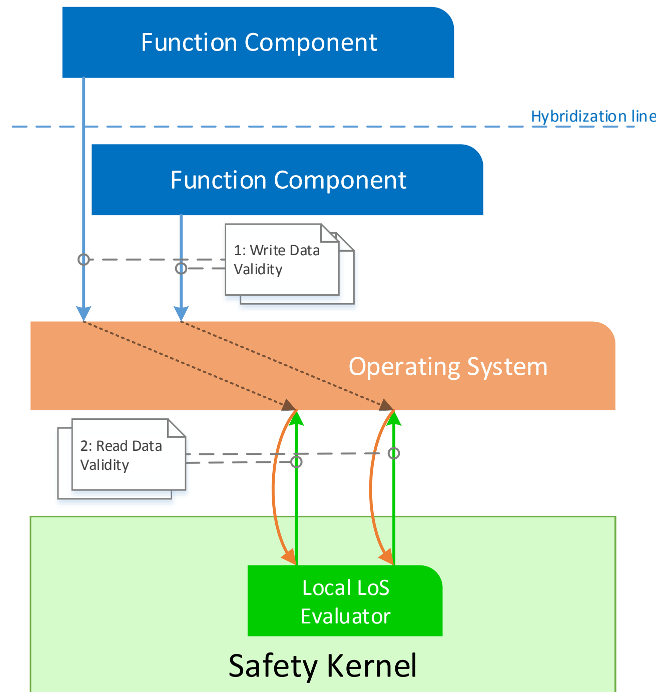


Figure 5.6: Data Validity Interface flow

the Cooperative LoS written by the Cooperative LoS Evaluator using the previous primitive.

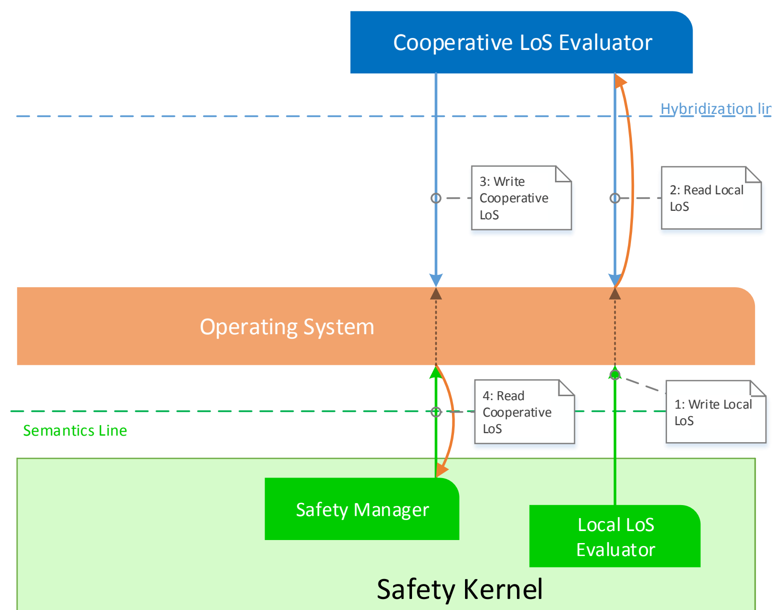


Figure 5.7: Cooperative LoS Interface

In this interaction, pictured in Figure 5.7, the Local LoS Evaluator uses the WRITE_LOCAL_LOS primitive (1) to inform the Cooperative LoS Evaluator of the LoS that is possible to offer by the system. This is read by the Cooperative LoS Evaluator with the READ_LOCAL_LoS (2) primitive.

Upon an agreement with other nodes, the Cooperative LoS Evaluator uses the WRITE_COOPERATIVE_LOS primitive (3) to inform the Safety Manager of the Cooperative LoS, which, in its turn invokes the READ_COOPERATIVE_LOS primitive (4) to read that value.

Mode Switch Interface

This interface implements the mechanisms to allow the Safety Manager to force a functional component to reconfigure its mode of execution to a different one as a consequence to a change in the LoS of a functionality. As such, it is composed by the primitives that allow the Safety Manager to write the mode in which a given component shall execute and the primitive that allow to read that value.

WRITE_ENFORCED_MODE This primitive is used by the Safety Manager to reconfigure a specific component for a certain mode

READ_ENFORCED_MODE This primitive is used by each component to read the mode set by the Safety Kernel using the primitive above.

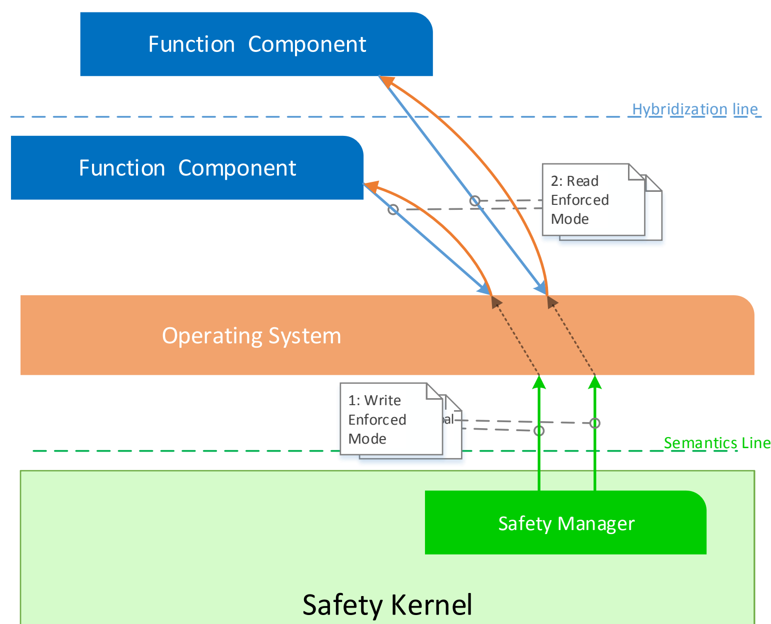


Figure 5.8: Mode Switch Interface

As pictured in Figure 5.8, this interaction is performed between the Safety Manager and the different functional components of the system. The Safety Manager uses the WRITE_ENFORCED_MODE (1) to inform each component of the mode in which they should operate from now on. This value is read by each component with the READ_ENFORCED_MODE (2).

Data Component Multiplexer Interface

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

WRITE_APP_OUTPUT_DATA This primitive, to be used by applications, allows each implementation of Function A to communicate its output value to whichever other functions may need it (including Function B). The value is provided along with a data validity measure, and reflects the output of Function A at a given LoS.

READ_APP_OUTPUT_DATA This primitive, to be used the Safety Kernel's Component Data Multiplexer, allows the Component Data Multiplexer to read the values provided by different implementations of a Function A.

WRITE_APP_INPUT_DATA This primitive, to be used by the Safety Kernel's Component Data Multiplexer, allows the Component Data Multiplexer to communicate (to whichever functions may need, including Function B's components) the appropriate value to be considered as the output Function A. This value is selected by the Component Data Multiplexer among the outputs provided by the implementations of Function A.

READ_APP_INPUT_DATA This primitive, to be used by applications, allows the implementation(s) of Function B to read the output provided by Function A when needed. Through this primitive, an implementation of Function B may read the output from Function A without needing to know about the variety of implementations of Function A.

In this interaction, pictured in Figure 5.9 the Data Component Multiplexer acts a mediator between a function with multiple implementations and other functions. Each component writes its output using the WRITE_APP_OUTPUT primitive (1), which is read

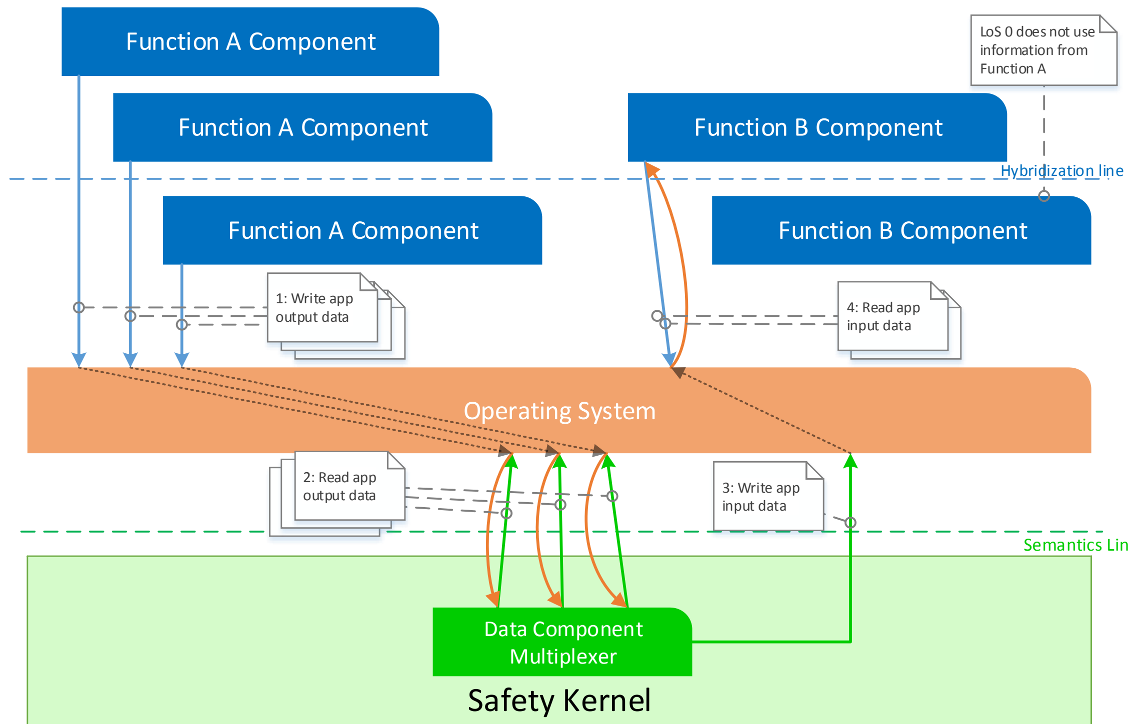


Figure 5.9: Data Component Multiplexer Interface

by the Data Component Multiplexer with the READ_APP_OUTPUT (2). After analysing and deciding which output should be forward, the output is written by calling the WRITE_APP_INPUT primitive (3). All components of that use this this value as input call the READ_APP_INPUT primitive (4) to read it.

5.3.4 Operating System Support

So far, we have only focused on the components of the system, without mentioning the underlying layer — the Operating System (OS) — in charge of managing these components and to provide services such as scheduling. Scheduling shall be done by the OS in such a way that allows temporal predictability in the execution of the multiple components executing in the system and of the interaction flows just described in the previous sections.

Scheduling

As mentioned, the timeliness guarantees provided by each component in the system may vary. However, the safety and timeliness guarantees of components below the hybridization line must hold even in the event of timing faults in the components above the hybridization line. For this reason, the different components must be scheduled in a way

which guarantees that the effects of any timing faults are contained in the scope of their occurrence - i.e., to the component where they happen. The scheduling done by the OS must, therefore, be deterministic and predictable. We achieve this by scheduling components strictly according to a *fixed schedule*, defined at design time, which assigns windows of time to each component, based on the demand expected for each component's workload. Each component may, then, use the allocated time window to schedule its own internal processes according to some local policy.

Since it is impossible to have a schedule that covers the whole system's lifetime, we define a schedule that covers a bounded interval and repeats itself over the time. The length of the schedule — its *period* — must be defined in order to provide a periodic guarantee to each component that its timing requirements are fulfilled. Since different components may require an execution guarantee with a period different to other components, the period of the fixed schedule should be the least common multiple of their periods.

The use of a fixed schedule ensures that, in the event that local scheduling inside a component above the hybridization line diverts, in execution time, from what was assumed in design time, the temporal properties of other components (which get their designated window of activity in any case) are not affected.

5.4 Summary

This section describes in detail the Safety Kernel component, used to manage the configuration of the system. First, it identifies the main issues that the Safety Kernel needs to handle, that is, the embedded design information, the runtime information, and how they are used to assess the LoS in which the system may operate. Besides the local assessment, the LoS of a cooperative functionality may also depend on the safety assessment made in the cooperating nodes. This, coupled with the local assessment, define the LoS to be applied to the system. Another identified issue that the Safety Kernel must foresee is allowing one function to be implemented by different components, acting as intermediary between components of different functions. For that, the Safety Kernel must read the output produced by different implementations of the same function and choose the one with higher validity and that fulfils the requirements for the currents LoSs to be used.

Based on these needs, we detailed the architecture and components that compose the Safety Kernel, the interactions between them as between the components of the system, as well as the support needed for the to operate correctly. The next chapter will detail an implementation example of the Safety Kernel and how it would be integrated within a TSP architecture.

Chapter 6

Safety Kernel Implementation

Based on the Safety Kernel definition described previously, this chapter gives an overview over a preliminary prototype implementation of the Safety Kernel within an AIR based system, detailing all the steps needed from the definition of the system, starting from the definition of the components and functionalities, safety requirements, configuration rules and implementation of the Safety Kernel itself.

6.1 Use Case system description

For the prototype, we started by defining one basic use case system composed by two functionalities; one used for ACC and other for Auto-Steering (AS). The components that compose the system and their interactions are pictured in Figure 6.1, for which we give a brief description next. Here we only focus on the functional components of the system (no Safety Kernel nor Cooperative LoS Calculator are considered yet).

Distance sensor
: Assumed to be reliable. Returns the distance to the nearest object in front of the vehicle with constant precision.

Global Positioning System (GPS) receiver
: Assumed to be unreliable. Returns GPS data regarding the position of the vehicle. Since GPS may not be available at all times, its precision and timeliness may vary.

Basic Speed Calculator (BSC)
: Assumed to be reliable. Calculates the speed to apply on the car based on the distance to the front object.

Smart Speed Calculator (SSC)
: Assumed to be unreliable. Calculates the speed to apply on the car based on the distance to the front object, GPS data and information exchanged with other vehicles (input and output). It has 2 modes of operation (Mode 1 and 2) with different performances and requirements that may be externally adjusted by means of parametrization.

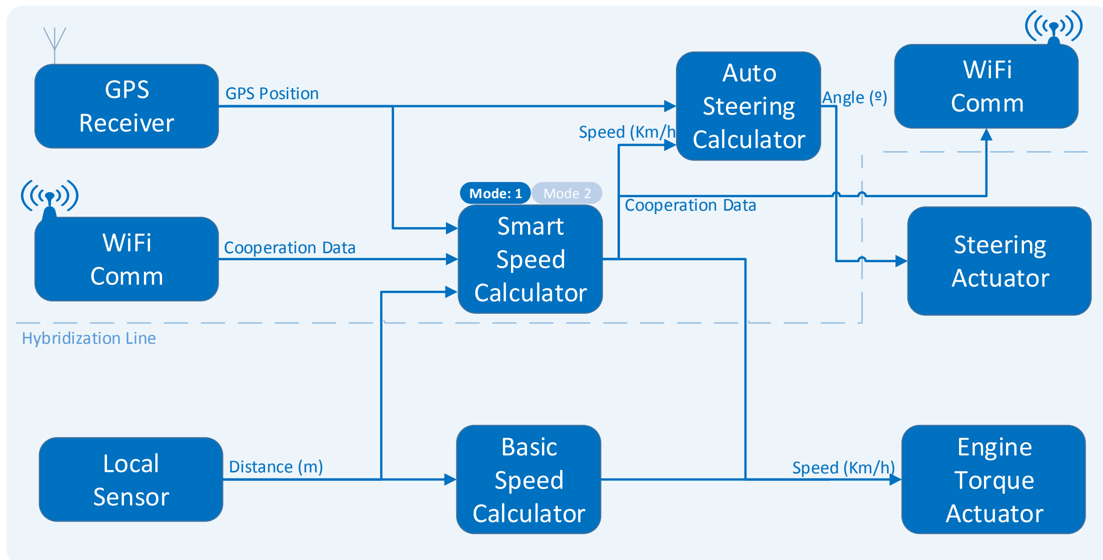


Figure 6.1: Use case system components

Auto Steering Calculator (ASC)
: Assumed to be unreliable. Calculates the angle to apply on the steering wheel based on GPS data and on the speed of the car calculated by the SSC.

Engine Torque Actuator
: Receives one speed, and applies it on the engine of the car.

Steering Actuator
: Receives one steering angle, and applies it on the wheels.

Wireless Communication
: Receives and sends data from or to other vehicles. As described in the fault model, this is an unreliable components, where no guarantees regarding the delivery and timeliness of messages may be given.

In addition to these components, due to the cooperative nature of the SSC, we define add an extra component to the system:

Cooperative LoS Evaluator
: Assumed to be unreliable. Uses wireless communication to communicate with other cooperative vehicles and reach an agreement about the LoS to use by the ACC functionality. When an agreement is reached, it outputs the result of that agreement. Otherwise, remains silent.

6.1.1 Functionalities Definition

Based on these components, we now define each functionality, their available LoSs and the set of safety requirements for each one.

Auto Steering Functionality

The main components involved in the Auto-Steering functionality are the ASC and the SSC, where the former receives data from the latter and from the GPS to calculate its output.

The functionality has two different LoSs available, henceforth called LoS 1 and 2, whose difference is related with the GPS validity requirements and with the execution mode of the SSC: when the GPS error increases, to guarantee safety, the functionality requires the use of a more conservative mode by SSC component.

If GPS is unavailable, if the GPS precision is higher than the required bounds or if the SSC does not compute a timely output, the Auto-Steering functionality shall not be used. In this case, we assume that a manual steering would be used in that situation, which would correspond to the LoS 0. However, for the sake of simplicity, we will not include this LoS mode in this example.

Table 6.1 describes the requirements and configurations of the two LoSs, where GPS_Validity is the validity of the GPS output (lower is better), SSC_Output_Age is the time elapsed since the last calculated speed by the SSC component and deadline is the maximum amount of the time for which that output will be valid.

So, as detailed in Table 6.1, LoS 2 requires a GPS error lower than 5 meters and a valid output from the SSC. In this level, the SSC should be executing in the mode 2. On the other hand, the LoS 1 requires a GPS error lower than 10 meters and the SSC should be parametrized to execute in mode 1.

Table 6.1: Auto-Steering functionality safety requirements

LoS	Requirements	Configurations
LoS 2	GPS_Validity $< 5 \wedge$ SSC_Output_Age $<$ Deadline	SSC Mode 2
LoS 1	GPS_Validity $< 10 \wedge$ SSC_Output_Age $<$ Deadline	SSC Mode 1

Adaptive Cruise Control Functionality

The ACC functionality uses all the components of the system except the ASC and the Steering actuator, having a total of three LoSs available:

The two higher LoSs, henceforth called LoS 1 and 2, use the output produced by the SSC component and, besides having requirements in terms of GPS validity, also require an agreement about which LoS to use with neighbour systems. That agreement is done by a Cooperative LoS calculator component, located above the hybridization line, whose task is to agree a common LoS for the ACC functionality with the remaining cooperative vehicles.

The LoS 0 use the BSC component, and should be used whenever it is not possible to use the LoSs 1 or 2. Since the BSC is proven safe and only relies on local sensors, its safety assumptions will always hold and its result may always be used. Table 6.2 describes the requirements and configurations of the three LoSs, where Cooperative_LoS represents the LoS agreed with the surrounding vehicles that are cooperating for the functionality, GPS_Validity is the validity of the GPS output (lower is better).

So, as detailed in Table 6.2, LoS 2 requires a GPS error lower than 10 meters and a valid Cooperative_LoS of 2. In this level, the SSC should be configured in the mode 2. For the LoS 0, since it is assumed safe and all its assumptions are assumed to always hold, no requirements in terms of safety need to be checked, reason why there are no requirements defined for this LoS in the table.

Table 6.2: Adaptive Cruise Control functionality safety requirements

LoS	**Requirements**	**Configurations**
LoS 2	GPS_Validity $< 10 \wedge$ Cooperative_LoS $= 2$	Use SSC Mode 2
LoS 1	GPS_Validity $< 20 \wedge$ Cooperative_LoS $= 1$	Use SSC Mode 1
LoS 0	-	Use BSC

The requirements defined in Tables 6.2 and 6.1 are the ones that originate the rules stored in the Safety Rules database, used by the Local LoS Calculator to assess the LoS of each functionality.

6.1.2 Configuration Rules

With the requirements of each functionality LoS defined, we defined the configuration rules used by the Safety Manager to configure the system. The configuration rules are derived from the safety rules of each functionality and, for each possible configuration, we define a set of conditions that define whether that configuration may be safety used or not. Upon the definition of these conditions it is need to take in account any possible dependency or incompatibility between functionalities.

In this example, the configuration rules define how shall the SSC be configured by the Safety Manager and the conditions in which that configuration shall be used. Table 6.3 schematizes the result of the analysis of the safety rules of each functionality, defining the conditions that must be met in runtime (1st column) and which configurations to apply (2nd column). ACC_LoOS and AS_LoS represent, respectively, the LoS calculated by the LoS manager for the ACC and AS functionalities and Cooperative_LoS represents the LoS calculated by the Cooperative LoS Calculator for the ACC functionality. The rules are defined to be evaluated in order from the first to the last one (from top to the bottom).

As such, the Safety Manager simply checks these conditions and when finds one condition that is verified, applies the required configurations.

Table 6.3: Configuration Rules

Conditions	**Configuration**
ACC_LoS = 2 $\wedge$ Cooperative_LoS = 2 $\wedge$ AS_LoS = 2	Use SSC Mode: 2
ACC_LoS $\geq$ 1 $\wedge$ Cooperative_LoS $\geq$ 1	Use SSC Mode: 1
Otherwise	Use BSC

As it possible to notice, the SSC component will only execute in mode 2 (the best one) if the LoS of the ACC (local and cooperative) and the AS is equal to 2. In other cases, the SSC performance needs to be reduced to meet the safety requirements of one of the functionalities. As specified in Table 6.3, if at a given moment the GPS precision is of 9 meters, then the local and cooperative LoS of the ACC would be 2 and of the AS would be 1. In this case, the SSC should be configured to execute in mode 1, which is the only safe configuration possible for the AS functionality, even if the ACC could be executed in LoS 2.

For the Steering Actuator, since it only receives input from one component (the ASC), the communication between them may be done in a direct way, without passing through the Data Component Multiplexer. If, however, we wanted to add a new component to provide the same input to the Steering Actuator, that would be done by placing the Data Component Multiplexer as intermediary between them, which could be easily done just by changing the origin and destination of the communication channels, without requiring any change to the application themselves.

6.1.3 Data Multiplexing

In this example the task of the Data Multiplexer and the definition of the multiplexing rules is pretty straightforward: both SSC and BSC are executed and implement the same function (calculate a speed) that is used as input of the Engine Torque Actuator. As such, the task of the Multiplexer is to decide which of the outputs forward to the actuator, based on the LoS of the ACC functionality (ACC_LoS) and on the timeliness of the last output wrote by the SSC component. Table 6.4 represents the rules used by the Data Multiplexer Component to choose the correct output.

As schematized, the Data Multiplexer makes its decision based on the LoS active for the ACC functionality and whether the output from the SSC is timely valid or not (i.e., if the time elapsed since the output was written is lower than the given deadline). Based on this simple condition, the Data Multiplexer would just need to forward the chosen output to the Engine Torque Actuator, which, in is turn, would just read the forwarded value,

Table 6.4: Speed Calculator Multiplexing Rules

Conditions	**Chosen output**
ACC_LoS $\geq 1 \wedge$ Cooperative_LoS $\geq 1 \wedge$ SSC_Output_Age $<$ Deadline	SSC Output
Otherwise	BSC Output

unaware of its original source.

6.2 Prototype

Aiming to demonstrate the feasibility and test the working of the Safety Kernel, we provided its engineering base on an AIR-based. We composed the system is in four partitions (P_1,..P_4), each one running a Real-Time Executive for Multiprocessor Systems (RTEMS)-based application [31], a real-time operating system for embedded systems that hosts the described components. As pictured in Figure 6.2, the first three partitions host mock-up applications of the described functional components of the system and the last one hosts the Safety Kernel application; We take advantage of partitioning to isolate the unreliable components from those we want to ensure execute correctly. For that reason, we isolate the BSC component in partition P_2 and the Safety Kernel in partition P_4. P_3 hosts the SSC and ASC and the Cooperative LoS evaluator that we know that are not timely safe. This way, we ensure that eventual timing faults occurring in P_3 do not propagate to other partitions. Finally, we used partition P_1 to host one mock up application that simulates the sensor components (distance and GPS).

The visualization and interaction of the prototype is made using VITRAL, a tex-mode windows manager for RTEMS [13] that allows us to include a different monitoring unity for each partition (P_1,..P_4) and two windows used to monitor the execution of AIR components. For demonstration purposes, we interact with the prototype using the keyboard. We take advantage of this interaction to inject faults in the system, allowing to simulate variations in the GPS Data Validity and to make the GPS available or unavailable, cause delays in the SSC execution and change the output of the Cooperative LoS Evaluator. The demonstration was implemented for an Intel IA-32 target platform and executed in QEMU, an open source machine emulator and virtualizer [8]

6.2.1 Communication and timing failures detection

Applications hosted in one partition may communicate with each other using the appropriate interpartition communication services provided by AIR, already described in 2. We implement the interfaces described 5.3.3 using sampling ports defined at integration time. Upon their creation we define the source port, the destiny port and one timeout period.

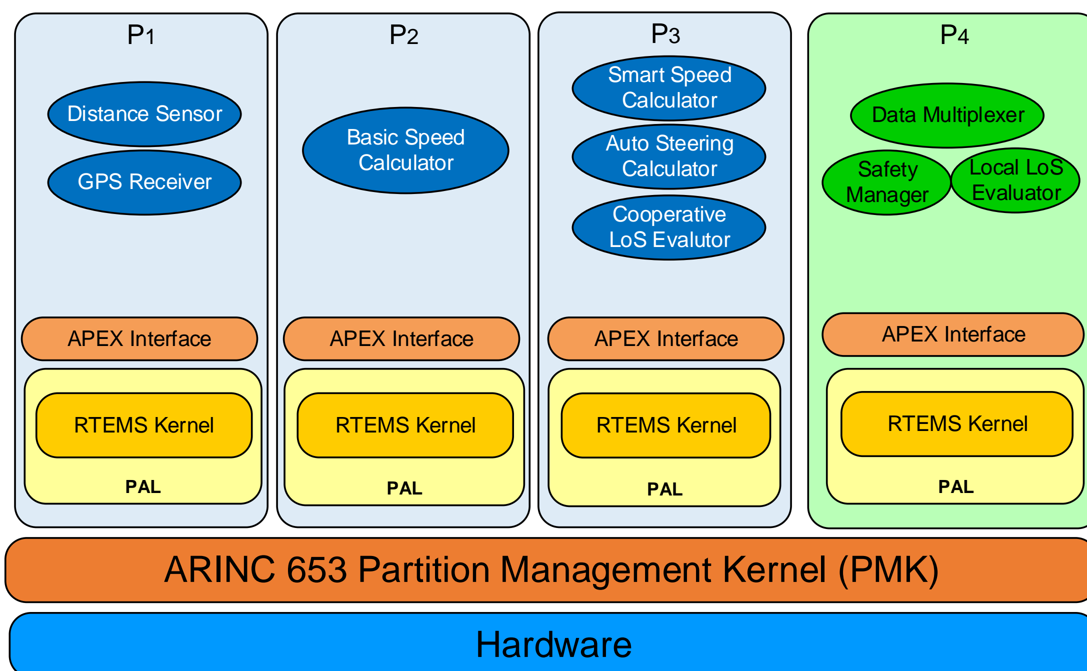


Figure 6.2: System implementation over AIR

We take advantage of this timeout period to detect timing failures in the execution of uncertain components. Every application, upon reading the value on a sampling port value knows whether that value is still valid or not. This allows the Safety Kernel to know if some component is executing in a timely manner or not and also allows components to know if the inputs they are using are valid or not. For instance, in the considered example, if the SSC gets delayed and does not produce a timely output, the LoS calculator and Data Multiplexer will detect that fact and take it into in their execution, which would have impact in the calculated LoS of the LoS Calculator and in the output forward by the Data Multiplexer.

6.2.2 Rules Implementation

One important step for the implementation of the Safety Kernel was the definition of the data structures to support the different types of rules; first, we started by defining a simple data structure that is used to store a simple comparison between two values. The snippet of the code used to define this struct is shown in Listing 6.1. As it is possible to notice, the structure stores two values and an enumerated type that represents the operator used to compare the two values, being this the support for the definition of the rules and conditions that will be checked and evaluated by the Safety Kernel.

All the remaining rules used by the Safety Kernel are constructed based on the *simple_rule_t* structure. For instance, the Listing 6.2, shows the definition of the safety rules, used

Listing 6.1: Simple comparison struct

```
/* Defines comparators to be used by rules */
typedef enum {
  LOWER, GREATER, EQUAL, GREQUAL, LOWEQUAL, NEQUAL
} comparador_t;

 /* Defines a simple rule, that compares two values */
typedef struct simple_rule {
  int *value_port;
  int bound;
  comparador_t comparador;
} simple_rule_t;
```

by the LoS evaluator. We first define a struct, *los_reqs_t* that store a set of *simple_rule_t*. This struct is used to store the requirements of each LoS of a functionality. Furthermore, we define the *los_calculator_rule_t* that stores a set of *los_reqs_t*, representing the multiple LoS of a functionality. We followed a similar approach for the definition of the conditions

Listing 6.2: Safety Rules struct

```
/* Defines the set of requirements of a given LoS.
The set of requirements is a set of simple rules */
typedef struct los_reqs {
  int n_of_reqs;
  simple_rule_t *requirements;
} los_reqs_t;


/* Stores the rules of the LoSs of one functionality */
typedef struct los_calculator_rule {
  int n_of_los;
  los_reqs_t **levels;
} los_calculator_rule_t;
```

used in the configuration and multiplexing rules.

In addition to this, we defined a simple function that evaluates one rule and returns whether that rule is verified or not. The signature of the function is listed in Listening 6.3. This simple function is used by all the components of the Safety Kernel when are iterating their rules and with it the remaining development of the Safety Kernel is very straightforward.

Listing 6.3: Evalulation function

```
/* Evaluates if given rule checks or not */
int evaluate (simple_rule_t);
```

6.2.3 Execution Samples

To better understand the operation of the Safety Kernel, we now present some execution samples of the prototype. Figure 6.3 shows the overall graphical aspect of the prototype with VITRAL, where it is possible to see the execution of the multiple partitions.

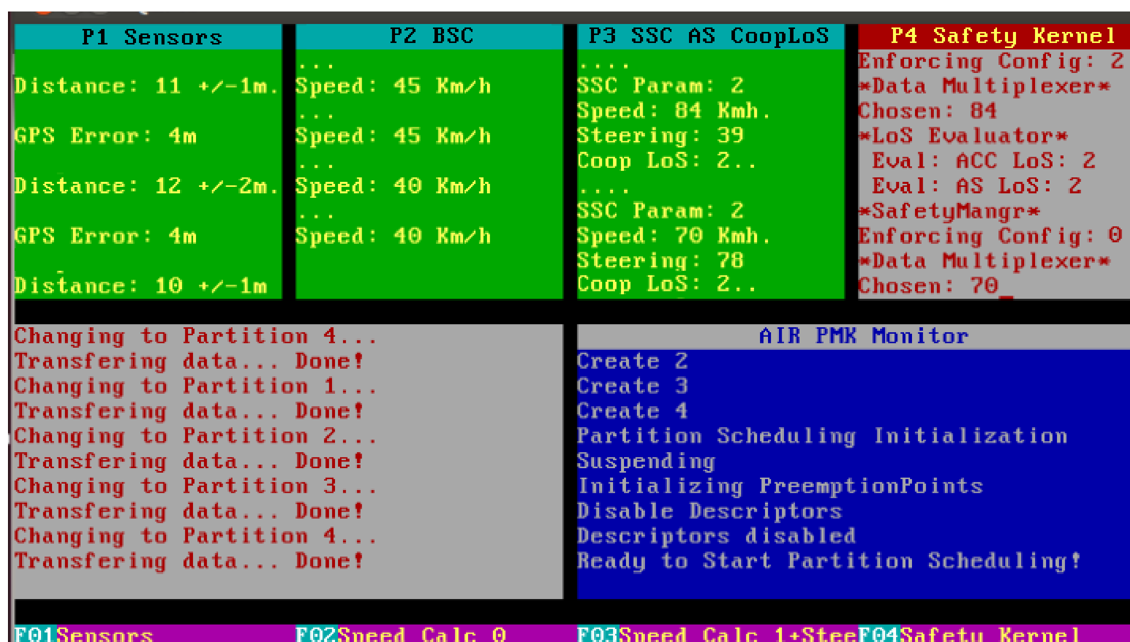


Figure 6.3: Prototype Sample Execution

In the moment captured by the Figure 6.3, under the Sensors partition (P_1) it is possible to notice the current GPS error (4 m) and the output of the Distance Sensors (10 m). In P_2, it is possible to see the output of the BSC component (40 km/h), whose calculation is based on the output of the Distance Sensor. P_3 shows of the SSC component which reads its parameter (2, writen by the Safety Manager) and based on that parameter outputs its result (70 km/h). Furthermore it is possible to se the execution of the ASC, whose output are just random values and the execution the output of the Cooperative LoS (*Coop Los: 2*) In the Safety Kernel partition, P_4, is done the evaluation of the local LoS for both functionalities (which in this case is LoS 2 for both). Furthermore, based on this evaluation and on the output of the Cooperative LoS the Safety Manager enforces one configuration (Enforcing config: 0, where 0 means that the first configuration of the Configuration Rules, defined in Table 6.3 was chosen). In addition to it, the Data Multiplexer reads the outputs of both BSC and SSC and forwards the latter (70 Km/h). The two lower

output windows show information related to AIR, such as the partition scheduling and dispatching and data transfers that support interpartition communication.

In Figure 6.4, we use the fault injectors to increase the GPS error to 24 m (see partition P_1). This, according to the rules limits has impact in the LoS of both functionalities that decreases to 0 and forces the Data Multiplexer to use the output produced by the BSC (50 km/h) (see P_4).

P1 Sensors
Distance: 12 +/-1m.
GPS Error: 24m
Distance: 12 +/-1m.
GPS Error: 24m
Distance: 12 +/-1m

P2 BSC
Speed: 50 Km/h
...
Speed: 50 Km/h
...
Speed: 50 Km/h
...
Speed: 50 Km/h
...
Speed: 50 Km/h

P3 SSC AS CoopLoS
....
SSC Param: 1
Speed: 72 Km/h
Steering: 23.
Coop LoS: 2.
....
SSC Param: 0
Speed: 0 Km/h
Steering: 62
Coop LoS: 2.

P4 Safety Kernel
Enforcing Config: 0
Data Multiplexer
Chosen: 72
LoS Evaluator
Eval: ACC LoS: 0
Eval: ASC LoS: 0
SafetyMangr
Enforcing Config: 2
Data Multiplexer
Chosen: 50

Figure 6.4: Prototype - Reduced GPS validity example

In Figure 6.5, we inject a timing fault in the SSC function that stops sending output (see partition P_3) This fault is, eventually, detected by the Local LoS evaluator that reduces the LoS of the AS functionality to 0. The fault is also detected by the Data Multiplexer because, since last output of the SSC is no longer valid, starts forwarding the result produced by the BSC (40 km/h).

P1 Sensors
Distance: 12 +/-1m.
GPS Error: 4m
Distance: 10 +/-1m.
GPS Error: 4m
Distance: 10 +/-1m

P2 BSC
Speed: 40 Km/h
...
Speed: 50 Km/h
...
Speed: 50 Km/h
...
Speed: 40 Km/h
...
Speed: 40 Km/h

P3 SSC AS CoopLoS
Steering: 29
Coop LoS: 2..
.....
Steering: 16
Coop LoS: 2..
.....
Steering: 55
Coop LoS: 2..
.....
Steering: 5

P4 Safety Kernel
Enforcing Config: 2
Data Multiplexer
Chosen: 84
LoS Evaluator
Eval: ACC LoS: 2
Eval: AS LoS: 0
SafetyMangr
Enforcing Config: 1
Data Multiplexer
Chosen: 40

Figure 6.5: Prototype - Timing fault example

6.3 Summary

This chapter gives a practical example of one simple system composed by two functionalities that interact with each other and that may be provided in different LoS. We described in detail all the steps needed to develop the Safety Kernel, starting from the definition of the safety requirements for each individual component of the system, the definition of the rules used to configure the system according to the LoS of each functionality. We gave a glance of how the Safety Kernel can be integrated on AIR-based platforms, and on the definition that we use to implement the different types of rules of the Safety Kernel.

To end, we gave some execution samples of the Safety Kernel implemented over the AIR architecture, demonstrating its feasibility and showing of its execution under the injection of faults.

Chapter 7

Conclusion

This thesis addressed the problematic of providing safety and predictability guarantees in safety-critical systems, despite the introduction of sources of uncertainty in the system.

We achieve this task with the definition of multiple Levels of Service for each functionality of the system, based on different safety assumptions and providing different guarantees. We manage these LoSs in runtime to ensure that the system adapts its behaviour and configuration to the observed constraints and avoiding any major hazard. If some safety constraint is violated, we reduce the LoS of the affected functionalities, trading the provided performance for safety. The definition of multiple LoSs opens the possibility to include more complex functionalities, with more relaxed safety constraints, able to provide features that were hard or impossible to provide with state-of-the-art solutions.

The contributions of this work, described through this document:

(i) The definition of a hybrid and modular architecture for safety-critical that hosts components whose development was based on different assumptions, enabling the definition of multiple functionalities and LoSs;

(ii) Support for advanced adaptation to faults that rely on reconfigurable components that are adjusted in runtime to meet the current LoS;

(iii) The definition of a generic component, the Safety Kernel, implemented as a rule checker, that provides the mechanisms and support for monitoring safety constraints and triggering the necessary reconfigurations of the system when any is violated;

(iv) A prototype implementation of Safety Kernel within an AIR-based system.

7.1 Future Work

For future work, we envision to implement the Safety Kernel in a Field-programmable gate array (FPGA),a programmable integrated circuit hosting a LEON microprocessors

executing an AIR-based system. The FPGA may then be integrated and tested into exiting systems.

Furthermore, it should be tested the impact of the solution in terms of performance, real-time and safety guarantees, analysing the possible pros and cons of the solution in terms of the desired capabilities and scope of use. Further studies should include the evaluation of the safety assurance according to the industries guidelines such as the ISO 26262 or the RTCA-O-178 for the verification, validation and certification of the software and test its results under realistic scenarios for the automotive and civil aviation industries.

Furthermore, a new trend related with the a new kind of system architecture starts to emerge is the concept of Distributed Integrated Modular Avionics (DIMA) [47]. DIMA combines the advantages of IMA and federated architectures, by defining a distributed system composed by multiple IMA systems that, despite being physically separated, interact with each other as in a regular system. When considering TSPs architectures, this model corresponds to the concept of distributed TSP systems. By themselves, distributed TSP represent a new research field, with new challenges and new considerations, implying the development and study of new solutions able to cope with the same concerns if terms of safety, robustness and security. For instance, regarding the AIR architecture, to cope with this new tendency, a new architecture is emerging — the Distributed AIR (DAIR) — corresponding to the concept of system of systems, in which a set of multiple distributed AIR-based systems may communicate and interact with each other to perform their tasks.

Bibliography

[1] AEEC. Design guidance for integrated modular avionics. ARINC Specification 651, Airlines Electronic Engineering Committee (AEEC), 1991.

[2] AEEC. Avionics application software standard interface. ARINC Specification 653, Airlines Electronic Engineering Committee (AEEC), January 1997, rev. Nov 2010.

[3] AEEC. Avionics application software standard interface part 2 - extended services. ARINC Specification 653, Airlines Electronic Engineering Committee (AEEC), 2007.

[4] AUTOSAR. Requirements on operating system, v3.1.0, release 4.1, revision 1, January 2013.

[5] AUTOSAR. Specification of operating system, v5.1.0, release 4.1, revision 1, February 2013.

[6] Algirdas A. Avižienis. The methodology of n-version programming. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 2, pages 23–46. John Wiley & Sons, New York, 1995.

[7] J. Barhorst, T. Belote, P. Binns, P. Hoffman, P. Paunicka, J. Sarathy, J. S. P. Stanfill, J. S. P. Stuart, and R. Urzi. A research agenda for mixed-criticality systems, 2009. White paper.

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[9] Tino Brade, Sebastian Zug, and Joerg Kaiser. A validity-based failure algebra for distributed sensor systems. In *SRDS 2013, Symposium on Reliable Distributed Systems*, Braga Portugal, set 2013.

[10] Mário Calha, João Craveiro, Pedro Nóbrega da Costa, and António Casimiro. First report on safety kernel definition. Technical report, KARYON Deliverable WP4.2, FCUL, 2013.

[11] California Parterns for Advanced Transit and Highways. PATH Project, 2006.

[12] António Casimiro, José Rufino, Luis Marques, Mário Calha, and Paulo Verissimo. Applying architectural hybridization in networked embedded systems. In Sunggu Lee and Priya Narasimhan, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 5860 of *LNCS*, pages 264–275. Springer Berlin Heidelberg, 2009.

[13] M. Coutinho, C. Almeida, and J. Rufino. Vitral - a text mode window manager for real-time embedded kernels. In *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pages 1254–1260, 2006.

[14] João Craveiro, Joaquim Rosa, and José Rufino. Towards self-adaptive scheduling in time- and space-partitioned systems. In *The 32nd IEEE Real-Time Systems Symposium WiP session*, Vienna, Austria, November 2011.

[15] João Craveiro and José Rufino. Adaptability support in time- and space-partitioned aerospace systems. In *Proceedings of the Second International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE 2010)*, Lisbon, Portugal, November 2010.

[16] João Craveiro, José Rufino, Tobias Schoofs, and James Windsor. Flexible operating system integration in partitioned aerospace systems,. In *Actas do INForum - Simpósio de Informática 2009, Lisbon, Portugal, Sep. 2009*, September 2009.

[17] João Craveiro, José Rufino, and Frank Singhoff. Architecture, mechanisms and scheduling analysis tool for multicore time- and space-partitioned systems. *ACM SIGBED Review*, 8(3), September 2011.

[18] João Pedro Craveiro and José Rufino. Schedulability analysis in partitioned systems for aerospace avionics. In *Proceedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2010)*, September 2010.

[19] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[20] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, IIES '08, pages 11–16, New York, NY, USA, 2008. ACM.

[21] ISO. ISO/IEC Guide 51:1999, Safety aspects — Guidelines for their inclusion in standards. Technical report.

[22] ISO. ISO 11898-1: Controller area network (CAN) – Part 1: Data link layer and physical signalling. Int'l Standard ISO 11898, 2003.

[23] ISO. ISO 26262: Road vehicles - functional safety. Int'l Standard ISO/FDIS 26262, 2011.

[24] David Jensen. B787 Cockpit: Boeing's Bold Move. Avionics Magazine, nov 2005.

[25] J. Kaiser and S. Zug. A fault-aware sensor architecture for cooperative mobile applications. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1512–1519, 2012.

[26] K. H. Kim. The distributed recovery block scheme. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 8, pages 189–209. John Wiley Sons, 1995.

[27] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[28] Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, Washington, USA, 1995.

[29] Richard M. Murray. Recent research in cooperative control of multi-vehicle systems. *ASME Journal of Dynamic Systems, Measurement, and Control*, 2006.

[30] Pedro Nóbrega da Costa, João Pedro Craveiro, António Casimiro, and José Rufino. Safety kernel for cooperative sensor-based systems. In *Safecomp 2013 Workshop on Architecting Safety in Collaborative Mobile Systems (ASCoMS)*, Toulouse, France, September 2013.

[31] On-line Applications Research Corporation. RTEMS C User's Guide, Edition 4.8.1 for RTEMS 4.8.1, aug 2008.

[32] D. Powell. Failure mode assumptions and assumption coverage. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 386–395, 1992.

[33] James Ramsey. Integrated Modular Avionics: Less is More. Avionics Magazine, feb 2007.

[34] Joaquim Rosa, João Craveiro, and José Rufino. Exploiting AIR Composability towards Spacecraft Onboard Software Update. In *Actas do INForum - Simpósio de Informática 2010, Braga, Portugal, Sep. 2010.*, September 2010.

[35] RTCA and EUROCAE. RTC-DO-178B/C - Software Considerations in Airborne Systems and Equipment Certification. Int'l Standard DO-178B/C, 1992.

[36] J. Rufino, C. Almeida, P. Verissimo, and G. Arroz. Enforcing dependability and timeliness in controller area networks. In *IEEE Industrial Electronics, IECON 2006 - 32nd Annual Conference on*, pages 3755–3760, 2006.

[37] José Rufino, João Craveiro, and Paulo Verissimo. Architecting robustness and timeliness in a new generation of aerospace systems. In António Casimiro, Rogério de Lemos, and Cristina Gacek, editors, *Architecting Dependable Systems VII*, volume 6420 of *LNCS*, pages 146–170. Springer Berlin Heidelberg, 2010.

[38] D. Seto, B. Krogh, L. Sha, and A. Chutinan. The Simplex architecture for safe online control system upgrades. In *American Control Conference, 1998. Proceedings of the 1998*, volume 6, pages 3504–3508, 1998.

[39] Lui Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, July/August 2001.

[40] S. Shladover and The Exploratory Advanced Research Program (U.S.). *Recent International Activity in Cooperative Vehicle-highway Automation Systems*. Office of Operations Research and Development, Turner-Fairban Highway Research Center, dec 2012.

[41] J.A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[42] Eushiuan Tran. Verification/Validation/Certification. Technical report, Carnegie Mellon University, 1999.

[43] Paulo Verissimo. Uncertainty and predictability: Can they be reconciled? In André Schiper, Alex A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *LNCS*, pages 108–113. Springer Berlin Heidelberg, 2003.

[44] Paulo Verissimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[45] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[46] James Windsor, K. Eckstein, P. Mendham, and T. Pareaud. Time and space partitioning security components for spacecraft flight software. In *30th IEEE/AIAA Digital Avionics Systems Conference (DASC 2011)*, October 2011.

[47] R. Wolfig and M. Jakovljevic. 3Distributed IMA and DO-297: Architectural, communication and certification attributes. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1.E.4–1–1.E.4–10, 2008.

[48] J. David Zook, Ulrich Bonne, and Tariq Samad. Automation, control and complexity. chapter Sensors in control systems, pages 131–150. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[49] Karl Johan Åström and Richard M. Murray. *Feedback systems : an introduction for scientists and engineers*. Princeton university press, Princeton, Oxford, 2008.