A Kernel-based Architecture for Safe Cooperative Vehicular Functions

(Invited Paper)

António Casimiro, José Rufino,
Ricardo C. Pinto, Eric Vial
Departamento de Informática
Faculdade de Ciências, Universidade de Lisboa
1749-016 Lisboa, Portugal
{casim, ruf}@di.fc.ul.pt
{rcp, evial}@lasige.di.fc.ul.pt

Elad M. Schiller, Oscar Morales-Ponce, Thomas Petig
Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
{elad, mooscar, petig}@chalmers.se

***Abstract*—Future vehicular systems will be able to cooperate in order to perform many functions in a more effective and efficient way. However, achieving predictable and safe coordination of vehicles that autonomously cooperate in open and uncertain environments is a challenging task. Traditional solutions for achieving safety either impose restrictions on performance or require costly resources to deal with the worst case situations. In this paper, we describe a generic architectural pattern that addresses this problem. We consider that cooperative functions can be executed with multiple levels of service, and we rely on a safety kernel to manage the service level in run-time. A set of safety rules defined in design-time determine conditions under which the cooperative function can be performed safely in each level of service. The paper provides details of our implementation of this safety kernel, covering both hardware and software aspects. It also presents an example application of the proposed solutions in the development of a demonstrator using scaled vehicles.**

I. Introduction

The increasing variety and quality of sensor technology, faster and more powerful embedded processors and new wireless communication standards, can be seen as a technological incentive towards the development of control systems and cooperative applications with increased functionality and capable of delivering higher levels of performance. In general, this should lead to improvements in terms of traffic flow, energy consumption and passenger safety. However, this also means that systems will become more complex and will have to rely on wireless communication networks, thus becoming more prone to failures and subject to uncertainties affecting the timeliness and sensor data quality. Being safety-critical systems, these uncertainties cannot be ignored.

The KARYON project addresses the problems caused by uncertainty, aiming at providing solutions to improve the performance of cooperative systems, without the need to incur in higher costs to prove safety in all situations. In contrast with existing approaches for architecting safe systems, which, for proving safety, usually require all sources of uncertainty to be eliminated, or bounds on these uncertainties to be established and taken into account in design-time, the approach taken in KARYON leaves part of the safety assessment for run-time. The key concept is to consider that functions can be performed in several levels of service (LoS), using a safety kernel to select the highest possible, but still safe, LoS. When timeliness and information quality are deemed to be good, which should happen most of the time, it becomes possible to operate in a higher LoS and offer better performance (e.g., fuel consumption, traveling time). The LoS of cooperative functions will be adjusted in run-time, depending on the health of the system and sensor data. In design-time, it is necessary to determine the conditions under which each cooperative function will be safely executed in each LoS. This is expressed by means of safety rules, against which the observed heath state is matched in run-time to determine the appropriate LoS.

In this paper we cover several aspects of developing safe cooperative systems, from the system architecture to the concrete implementation of fundamental components, also including an example that illustrate how these components are used in a realistic setup and for realistic cooperative functions. We use the Gulliver test-bed [3], [17] to demonstrate the proposed concepts. The test-bed consists of several scaled vehicles that realize cooperative functions. The software architecture within each vehicle follows the proposed architectural pattern and, in particular, uses a safety kernel for safety management. For that, the hardware and software solution presented in this paper for implementing the safety kernel was integrated in Gulliver vehicles. The test-bed is thus adequate to demonstrate the architectural concept, and to show that it is possible to manage the performance level depending on the operational conditions, while ensuring that the functions are always performed safely.

This paper provides the following specific contributions:

- A set of concepts and definitions that is fundamental for achieving safe cooperative systems in environments subject to faults and temporal uncertainties. These concepts set the stage for a better understanding of the problems under consideration, and particularly for understanding the proposed solutions.
- The generic KARYON architectural pattern for safe cooperative systems. The architectural pattern is at a sufficiently high level of abstraction to allow varied instantiations, depending on the concrete system that is to be developed. On the other hand, it provides

the necessary guidelines for system designers, by explaining how to structure the system and by defining the fundamental architectural components that must be present in any concrete system.

- Descriptions of the concrete hardware and software solutions that were used to implement the safety kernel.
- An example application using the Gulliver test-bed. The application demonstrates the architectural and safety kernel concepts, through the implementation of Cooperative Advanced Driver Assistance Systems (CADAS) in scaled vehicles that are safe in the presence of uncertainty.

The paper is organized as follows. In Section II we discuss how the presented solutions related to major approaches existing in the literature. In Section III we introduce fundamental concepts for developing safe cooperative systems. Then, Section IV is devoted to the description of the KARYON architectural pattern and Section V to the implementation of the safety kernel. The example application is presented in Section VI and the paper is concluded in Section VII

II. Related work

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

The approach we describe in this paper, which is followed in the KARYON project, is to consider a hybrid distributed system model [23] and explore the concept of architectural hybridization as a baseline design principle [24]. Based on that, we define a generic architecture that accommodates both complex functions that might be subject to temporal uncertainties, and simple and well-defined functions, with predictable temporal behaviour, which are fundamental to address safety concerns. In this way, the resulting systems will be able to incorporate different kinds of components with respect to exhibited properties, namely concerning timeliness and reliability. In fact, our approach has some similarities to approaches used to address problems found in systems of *mixed criticality* [2], in particular when it comes to enforce assumed properties through partitioning for fault confinement. However, while in these systems the objective is to allow applications with different levels of criticality to coexist, our focus is on functions that need to be always performed safely while using components with heterogeneous properties. Interestingly, the hybrid nature of systems was also acknowledged in the GENESYS project [16], which developed a component-based generic platform for embedded real-time system. However, GENESYS is significantly focused on the problems related to composition and component interfaces, whereas our interest is on understanding how uncertainty can be characterized and how the performance can be managed while making sure that safety requirements are always satisfied.

One important aspect in our approach is that we need to manage the service level of cooperative functions in run-time, reacting to changes in the integrity of data or the timeliness of components to ensure that the functions are executed safely. This has some relations with the recovery block concept [11], where multiple versions for the same function are developed. The more complex version of the function (with extra features and more prone to errors) is executed by default. If an error is detected, then a simpler implementation is executed. In fact, Simplex [21] follows a similar approach by defining an architecture composed of two alternative control functions, where one is designed to achieve improved control at the expense of increased complexity and potential unreliability of the control algorithm, while the other uses a simple and proven safe control algorithm. It tolerates faults in the unreliable controller using a decision module that observes the plant to verify if the the controller is being able to keep the controlled system within the desired operational envelope. If not, it switches the execution to the reliable controller, trading off performance for safety. The solution is thus designed by assuming that faults are ultimately reflected on some undesired external behaviour, which can be reliably observed through the existing sensors. Our approach is different in the sense that we use the reliable part of the system to evaluate the integrity of control data (e.g. sensor data that might be affected by sensor faults) and the timeliness of specific activities (e.g., consensus reaching, which might be affected by communication faults). Given that, the solutions for deciding when to change the control algorithm, or when to perform some system reconfiguration, are done in a different way than it is done in Simplex.

In previous papers we have presented different aspects of our solutions. In [6] we provided an overview of the overall work being developed in the KARYON project, where we included a brief description of the architectural concept. The safety kernel internal architecture was previously detailed in [7]. Therefore, here we just provide an overview of the safety kernel architecture, for self-containment reasons. On the other hand, we provide an extensive description of its implementation. The Gulliver test-bed, which we used to develop our example application, was previously presented in [17] and [3]. In [5] we present the details of the algorithms for the cooperative functions and, finally, in [9] we address the aspects of safety assurance with the proposed architectural approach.

III. Concepts and Definitions

In this section we introduce important concepts that need to be considered when defining solutions for safe cooperative vehicular functions. First, we provide our perspective on what is a cooperative system. Then we explain in detail the concept of level of service in the context of a cooperative function and, finally, we introduce some key ideas concerning safety analysis.

A. Cooperative Systems

In KARYON we focus on cooperative systems that include autonomous cars, robots, airplanes or Remote Piloted Vehicles

(RPVs). The meaning we give to *cooperation* is that systems actively help each other in order to achieve some common goal, or to realize some cooperative functionality. Cooperation can be used to improve coordination among vehicles. The concept of *coordination* is more general in the sense that vehicles can be coordinated by other means, for instance by following pre-established rules embedded in control algorithms. Vehicles can thus coordinate their behavior without the need to communicate with each other. Communication must be seen as an enabler for cooperation, for agreements to be reached in run-time, which will allow improving the traffic flow, the safety and the overall energy consumption.

When considering cooperative vehicles, *communication* between vehicles is fundamental. This is in contrast with non cooperating vehicles, even autonomous ones. Autonomous vehicles operate in a fully independent way, without the need to actively interact with other vehicles or entities and hence without communication needs. Since communication is absolutely required for cooperation, its quality and reliability dictates how well cooperation may be achieved. The problem is that wireless networks are prone to interferences and much less reliable than wired networks, making it harder to build cooperative vehicular applications and, in particular, to make them safe. In fact, cooperation may not be achievable at all times.

Because of that, any reasonable and practical solution (at least using the existing communication technology) *must allow individual vehicles to operate autonomously* when communication is not possible. The overall system performance (considering all vehicles) will be smaller, but safety will be preserved. On the other hand, and hopefully most of the time, communication will be good enough to support cooperation and thus achieve performance improvements. The implication on the system design is that it will be necessary to consider different modes of operations, for different situations concerning the communication quality, and devise architectural solutions to support system changes in run-time.

B. Level of Service

Achieving higher functional performance while excluding the additional safety risks this this could bring, either requires all risks to be excluded at design time, which might be costly due to the uncertainties affecting the system operation, or requires a risks to be managed in run-time. The concept of level of service (LoS) is introduced in this context and is explained next.

When designing the nominal control system that performs the cooperative function several assumptions have to be made. These are about the possible contexts in which the function will be performed (e.g., weather conditions, potential obstacles, traffic regulations) and also about system properties (sensor faults, communication delays, task execution jitter). Our focus is on functional safety (for instance, according to ISO 26262 [8] for the automotive domain) and thus we consider that, provided system properties hold true, the function will be performed safety for whatever contexts it has been designed for. Given that it is not possible, or very expensive, to enforce the best system properties at all time (e.g., wireless communication quality varies), it is necessary to design multiple version of the nominal control system, each of which which assuming different system properties and being proven safe in design time for that set of assumptions. The consequence is that the function may be performed with different levels of service, depending on the nominal system configuration.

In run-time the nominal control system will thus be adjusted to execute the function with the level of service that is safe with respect to the observed system state. For instance, if the quality of the information provided by a sensor distance degrades, then it becomes necessary to exclude the additional risk brought by this degradation which can be done by adjusting the operation of the control system to a mode in which higher safety margins are assumed. A similar adjustment of the control algorithm has to can be made when it becomes impossible to achieve timely communication or timely consensus among cooperative entities (within assumed bounds). The KARYON project also aims at defining methods and solutions for evaluating the quality of sensor data [4], but this is outside the scope of this paper.

It is fundamental to note that we are considering cooperative functions, which means that the level of service is common across all cooperative vehicles and is thus often referred to as the *cooperative level of service*. If a single vehicle is not able to perform the functions in a certain LoS, then all the vehicles have to adjust their behavior accordingly. At worse, if they cannot reach agreement on the cooperative LoS (which may happen if communication becomes untimely), then they will all adjust the operation to a previously agreed LoS (one in which the risks associated to lack of communication are excluded).

Finally, it is necessary to define one mode of operation that will be always safe, which we consider to provide the lowest LoS. In this mode of operation all possible system faults and uncertainties have been excluded by design.

C. Safety analysis

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

- Part of the overall safety of a system or piece of equipment that depends on the system or equipment operating correctly in response to its inputs, including the safe management of likely operator errors, hardware failures and environmental changes.
- For the automotive domain, ISO 26262 defines it as absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems [8].
- In the avionics domain, the RTCA-DO178B/C's Design Assurance Level (DAL), defines the safety level through the effect a failure will have on an aircraft.

The safety analysis is always static and today's approach is to take the worst case assumptions and make the system safe according to those assumptions. A vehicle is a static entity and this approach is viable for today's situation and operating modes. Moving to fully autonomous, coordinated and cooperative mode, the controllability factor for involved persons will become much degraded. The vehicle will change from a closed

system to an open system, i.e. a system that interacts with its environment. The system must also be adaptive to handle a dynamic environment. The safety analysis must shift from a self-view to a relational-view. Today's safety standards do not address these new situations.

To be able to take full benefit from coordinated and cooperative systems it seems necessary to move from a purely static view on safety integrity levels during design time into partly dynamic safety assertions in run time. By doing so, the actual run time operational situation will dictate the current need for some of the safety integrity levels. This implies that the safety analysis has to be stored in the vehicles as safety rules which map operational situations to safety integrity levels. Hence the safety analysis is still static but it is no longer the worst case scenario that dictates the needed safety integrity level. The selection of some of the safety integrity levels becomes a dynamic run time process as well as the assertion that those integrity levels are achieved.

IV. The KARYON Architectural Pattern

The proposed architectural solution enables the development of systems in which cooperative functions can be executed with multiple levels of service, and where components can exhibit heterogeneous properties concerning reliability and timeliness. It relies on the *architectural hybridization* paradigm [24] to match the necessarily hybrid nature of the system, in which some components have to be made predictable at design time, and other cannot be so. The following section describe the system architecture, including an overview of the safety kernel components, introduce the considered fault model and discuss some relevant timeliness issues.

A. System Architecture

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

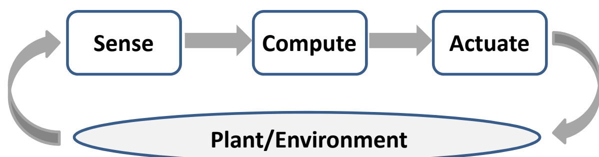

Fig. 1. Basic control loop.

A basic control system involves sensing, processing and actuation. We say that this is the nominal control system, because it includes the strictly necessary components to realize the desired control activities. It is well-known that it is easier to ensure control stability, in which controlled variables are kept within desired bounds, when the system and the environment are well defined and all variables can be characterized in a precise way. From a modeling perspective, this translates into considering synchronous models, well-defined failure modes, known event patterns, etc. Control stability is fundamental to meet safety requirements, whenever these are defined. Solutions for stable and safe control under precisely defined conditions are well known in fully described in the literature. They imply a detailed system analysis at design time (i.e., statically), to prove that the necessary safety conditions are met.

However, this simple model is not adequate to our purposes because: a) we need to consider communication components, for a distributed operation; b) we must handle several components of a kind (e.g., several different sensors); c) we want to support systems that execute several cooperative functions at the same time, sharing components among them; d) we have to acknowledge the fact that some components may behave in a non-deterministic way, and that they have to be considered separately from parts of the system that are made predictable by design.

Therefore, we model the nominal control system as shown in Figure 2.

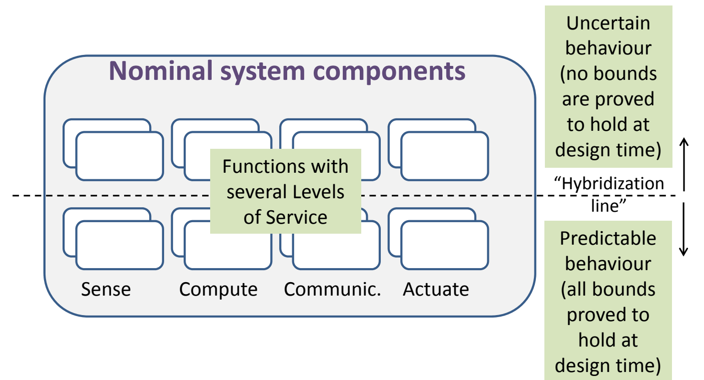

Fig. 2. Nominal system for improved performance.

Besides exploiting cooperation, the objective is also to exploit the possible use of more complex control algorithms and of a richer variety of sensors. As we have discussed previously, this implies that functionality will be provided with more than one LoS, which must be reflected in the system architecture. In particular, and given that we introduce complex components whose behavior might not be proven in design time to satisfy some bounds, the general pattern is to allow for heterogeneity of assumed properties. The hybridization line explicitly separates the system in two parts, following the architectural hybridization paradigm. We note that although actuation components are also represented above the hybridization line, for generality, special special solutions based on interposition or on the use of redundancy would be required to deal with uncertain actuation.

In order to deal with the possible faults affecting the timeliness of computing components or the quality of sensor data, that is, to adjust the system configuration in run-time so that all the functions are executed in the LoS that secures safety, additional components need to be included in the architectural pattern. In addition to the nominal system we add a *Safety Manager* component and associated *Design Time Safety Information* and *Run Time Safety Information* components. These are the fundamental components of the *Safety Kernel*. We also highlight the fact that these components are located below the hybridization line. This is necessary because it is fundamental to make sure that the safety kernel components properties (e.g., timeliness) are proven in design-time to always hold true. The complete architectural pattern is shown in Figure 3.

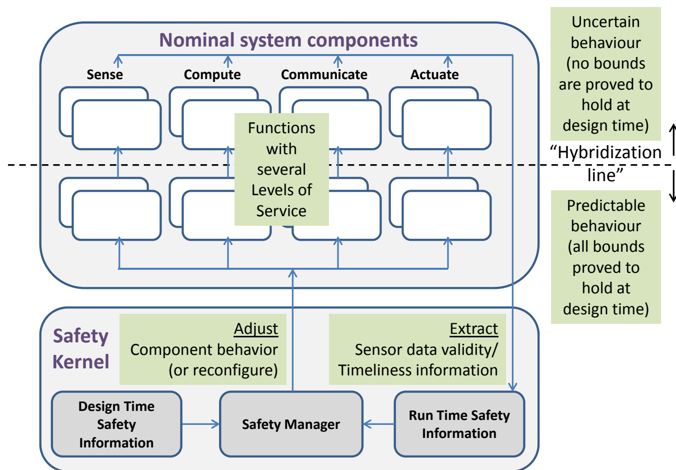

Fig. 3. Complete KARYON architectural pattern.

The safety kernel is separated from the nominal system in order to make it clear that the components belonging to the safety kernel are generic, i.e., they can serve any particular application. On the contrary, each component of the nominal control part is application-dependent.

The design time safety information consists in safety rules expressing the required sensor data validity and component timeliness that are needed to ensure that a cooperative function can safely performed with a certain LoS. Therefore, there are safety rules for each LoS of each cooperative function (excluding the lowest LoS). In run-time, the safety kernel collects information about data validity and execution timeliness, i.e., run time safety information. Then, the safety manager is the component in charge of matching this run time safety information against safety rules, in order to determine the appropriate LoS for each function. Depending on that, it also determines if it is necessary to make any adjustment concerning the operation or the configuration of the nominal system components. For this purpose, the design time safety information also includes specifications of the concrete system configurations for each combination of functional LoS. A detailed description of the safety kernel internal architecture and operation can be found in [7]. In Section V we describe a hardware and software solution for implementing these safety kernel components.

B. Fault Model

Sensor components can experience various faults affecting their output. When sensors are above the hybridization line, these faults can be both in the time and in the value domain. Otherwise they can be only in the value domain. Faults in the time domain include crash faults, i.e., when a sensor does not provide further output, and timing faults, when a sensor produces a late output. A subset of sensors must behave in a predictable way. This is necessary to ensure that functions relying on those sensors can be executed at least with a the baseline LoS.

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

Communication components can be modeled just like sensor components. In fact, it is possible to consider that communication components provide local interfaces to remote sensors. Above the hybridization line, communication components can either crash of produce timing faults. In principle, all communication components will be above the hybridization line, because it is hard to achieve timely communication in wireless networks with sufficiently high probability, as usually necessary in the considered applications.

Actuator components are assumed not to fail. We are essentially worried with faults affecting the correctness of perception, rather than the correctness of actuation. Failures in the mechanical system including the mechanical parts of the actuators are outside the scope of our the work being developed in the KARYON project and are not considered.

C. Timeliness Issues

Timeliness issues are important when it comes to safety assurance. Vehicle control functions, for all LoS, are designed based on timing assumptions about task execution times, communication delays, etc. In addition to the typical assumptions, it is also necessary to take into account the time that it may take to perform adjustments in the system configuration, when the LoS of a function has to change. Given that these adjustments are done by the safety kernel, it is fundamental that the safety kernel operation is temporally predictable. And this is one of the reasons why the safety kernel has to be implemented below the hybridization line.

V. Safety Kernel Implementation

A. Hardware Platform

The implementation of the safety kernel components is done in software, supported by a suitable computing platform for its execution. The functional elements to be provided by the hardware platform are: **Processing Unit**, providing the computing resources; **Read-only Memory** (ROM), to store the safety kernel software code and the safety rules; **Random Access Memory** (RAM), supporting the safety kernel execution; **Input/Output Interface**, to enable the exchange of data between the safety kernel and the nominal system components.

The fulfillment of the requirements is achieved through the usage of a development board containing a reconfigurable logic device (FPGA), together with Intellectual Property (IP) cores part of a System-on-a-Chip (SoC) library, mapping the functional elements such as Input/Output Interface into the reconfigurable logic device.

The selected development board is a Trenz TE-0600, comprised of: Xilinx Spartan-6 FPGA; 256 MB of RAM memory; Ethernet physical interface; Flash ROM and an Secure Digital (SD) card physical interface. (*see* Figure 4).

The Flash ROM (not shown in Figure 4) serves as non-volatile storage for the safety kernel, whilst the SD Card

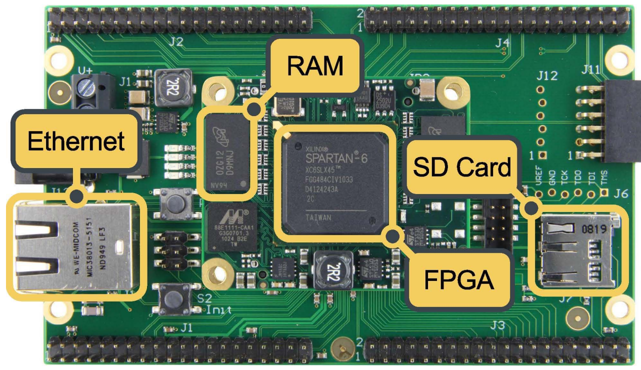

Fig. 4. Safety Kernel Hardware Platform

interface supports the Safety Rules, written *offline* to an SD card. The FPGA supports the mapping of the controller mechanisms for these memory interfaces, together with the processing unit and Ethernet controller.

The elements to be implemented in the FPGA are provided by the GRLIB SoC library [1], which encompasses IP cores providing I/O functions, such as Ethernet and UART, together with the remaining components needed to implement a computer, e.g. memory and interrupt controllers. The processing unit is implemented by the LEON3 soft-processor, a SPARCv8 architecture used in the Space domain by the European industry. A diagram with the SoC elements is shown in Figure 5.

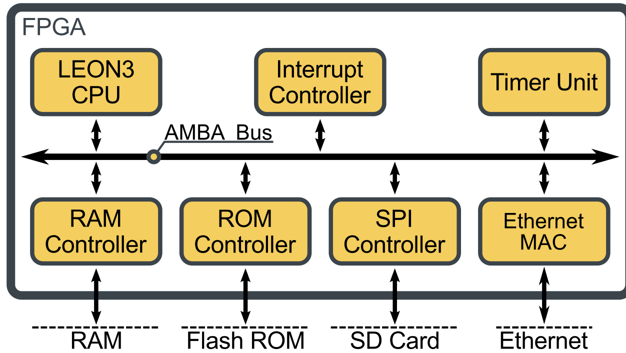

Fig. 5. Safety Kernel Hardware Platform

The elements are interconnected through AMBA, a SoC bus specified by ARM. The system presented in Figure 5 enables the usage of the development board as a fully-fledged embedded computer, suitable to run a real-time operating system.

B. Software Architecture

We developed the safety kernel using the C language, and we made it available in two Posix-compliant versions. The first can be compiled on any Unix/Linux platform, but due to the lack of real-time support in these platforms, this version is reserved for testing purposes. In contrast, the second version is aimed to the RTEMS real-time operating system that is installed on the hardware board described in the previous section.

The safety kernel can be used as an independent and self-contained subsystem. Communication with the external components is point-to-point, and can be implemented on top of the existing I/O interfaces provided by the underlying board. Details about the communication API are provided in the next chapter.

The safety kernel operation (e.g., execution period of the safety manager) can be configured by means of a configuration file in XML format, which is loaded during the system bootstrap.

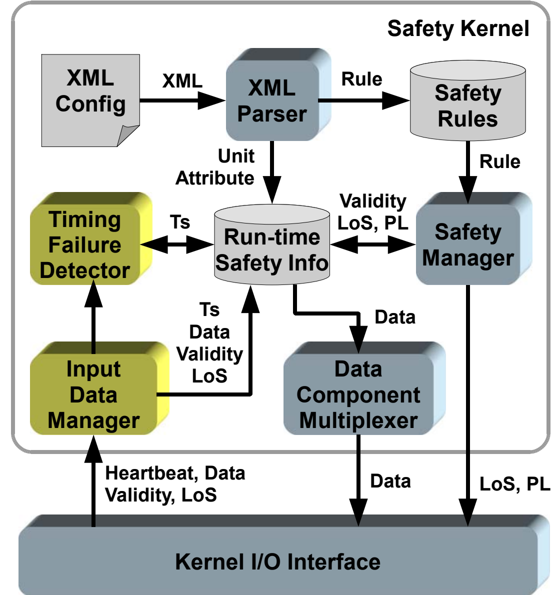

Fig. 6. Functional architecture

Figure 6 presents the functional architecture and the data flows exchanged between the main function modules. The safety kernel uses two data structures to respectively store the safety rules parsed from the XML configuration file and the run-time safety information that will be collected from specific nominal system components. In fact, we define two safety kernel components that are able to provide run-time safety information: the *Input Data Manager* and the *Timing Failure Detector*. Part of this information can also be locally calculated by the *Safety Manager* (e.g., the acceptable LoS for some function). The *Input Freshness Manager* receives validity, LoS values from remote vehicles, and heartbeats for timing failure detection, and uses them to update the *Run-time Safety Info*. The *Timing Failure Detector* checks the timeliness of the potentially non-timely units, based on received heartbeats and on configured (in design-time) timeout attributes. Finally the *Safety Manager* evaluates if safety rules are satisfied by existing run-time safety data and updates, if necessary, LoS information also within the *Run-time Safety Info* repository. The safety manager is also in charge of checking if some LoS occurred, and if so, it will be able to determine adjustments on the performance level (PL) of specific nominal system components, or changes in the system configuration, which are achieved controlling the flows of application data between nominal system components. In fact, the *Data Component Multiplexer* is specifically meant to multiplex, among different sources of data, the one that should be forwarded to some component.

The safety kernel is powered by two concurrent Posix threads depicted in yellow in figure 6: the *Input Data Manager*, which analyzes the incoming information, and the *Timing Failure Detector*. This latter thread is triggered every X milliseconds, where X is the safety kernel execution period. This value can be changed in the XML configuration file. When the timeliness checking phase is completed, the *Timing Failure Detector* calls sequentially both the *Safety Manager* and *Data Component Multiplexer* functions.

C. Interfaces

The safety kernel environment provides a simple API library written in C, designed to easily send and receive information to/from the safety kernel board. The connection with the safety kernel can be respectively opened and closed by the **sk_init** and **sk_close** primitives. The API provides a set of writing primitives to send data to the safety kernel: **sk_write_heartbeat**, for a simple heartbeat message, **sk_write_validity**, for a component data validity, **sk_write_data**, for a data value to be multiplexed within the safety kernel, and **sk_write_level**, to send the LoS received from another vehicle. To get a response from the safety kernel, information must be first received by calling **sk_receive**, and then some primitives can be used to handle received information: **sk_read_type**, to identify the type of received information, **sk_read_id**, for the retrieve an identification of the nominal system component to which the information is directed, **sk_read_data**, for obtain a data value resulting from a multiplexing operation performed within the safety kernel, **sk_read_level**, for obtain LoS or PL information. This LoS information (from a certain function) can be send to other vehicles.

The information sending frequency depends on the nature on the source component. For instance, all nominal system components below the hybridization line can send data validity information just when it changes changed. On the other hand, components above the hybridization line (which might not be always timely), must periodically send either data validity values or heartbeats, so that the safety kernel is able to monitor how timely they are.

VI. Example Application

A. The Gulliver Test-bed

Vehicular systems require extensive validation of new ideas and an iterative upgrade of the technology readiness level (TRL) of the system. The designers of vehicular systems often use simulation tools for validating and testing systems behavior. Today, a number of large-scale testing facilities that use real vehicles exists. These facilities and the vehicular prototypes are very costly.

We use the Gulliver test-bed that combines digital and physical vehicular stimulations. The test-bed uses physical miniature vehicles to facilitate extensive testing in a representative environment and by that aim for TRL-5. Our system validation integrates simulated and physical miniature vehicles in a common environment. The presented system provides flexibility, cost efficiency and broad testing for future vehicular systems.

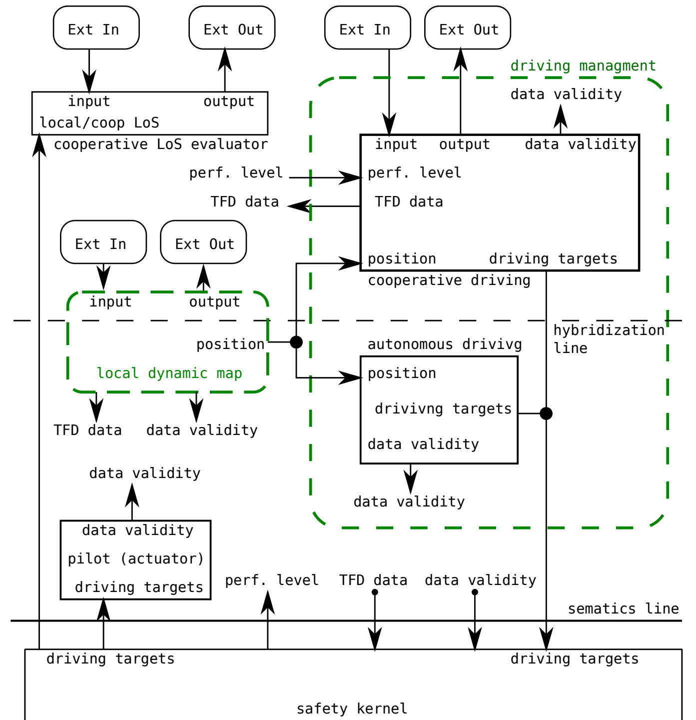

Fig. 7. System architecture for vehicular coordination with a safety kernel in the Gulliver test-bed.

We illustrate the test-bed software components and their interaction in Figure 7. The primary components are the safety kernel, which monitors data validity flows (Section V), and the cooperative evaluator of service level that allows the vehicles to agree on the performing service level (Section VI-C2). Two other important components are the Local Dynamic Map (LDM) (Section VI-C1), which provides location information concerning all vehicles, and the Driving Manager (Section VI-C3) that determines the cooperative and autonomous trajectories. The driving manager encompasses the cooperative driving and the autonomous driving components, which used alternatively (only the outputs of one of them are used at a time), depending on the current service level.

The algorithms that we use for implementing the system in Figure 7, consider a system that includes a well-known set, $members$, of n identical vehicles with unique identifiers (network addresses) that are known to all. Each vehicle is an autonomous mobile entity with a set of capabilities that includes sensing, communicating, computing and controlling its motion speed and steering. Vehicles have access to a common clock. The time is divided into *rounds* of exactly $ROUND$ time units. We simple denote the present round as $round_{now}$. We assume that all onboard processors do not crash.

B. Vehicular Functions

1) Adaptive Cruise Control and Vehicle Platooning: The application adjusts the headway (inter-vehicle distance) according to the data validity and the performance level. Namely, the safety kernel adjusts the application so that it enforces a larger inter-vehicle distance when the performance level deteriorates. However, when the performance level is high

enough, it allows the vehicles to maintain shorter inter-vehicle distance. The information that the vehicles require to perform this application is the relative distance to, and the speed of the vehicle ahead which is wirelessly exchanged. It also requires vehicle to vehicle communication for agreeing on the joint (cooperative) cruising speed.

In this functionality, the highest service level is, indeed, the cooperative application of vehicular platooning and the lowest service is the autonomous application of adaptive cruise control. The first is based on [22] and the second on [10]; see the papers for formal proofs. We do not consider external factors, such as friction, mass, and road condition. Thus, the vehicular platooning depends only on the speed and relative position of each other vehicle. Meanwhile, the adaptive cruise control depends only on the speed and relative position of the vehicle in front.

On a broader scope than KARYON, cooperative vehicle systems have the task of determining the participating vehicle that are forming the platoon in order to avoid conflicting trajectories due to different views. The algorithm in details appears in [5].

2) Intersection Crossing: This application coordinates how the vehicles cross the intersections. In the highest level of service the vehicles cross the intersection with minimal waiting time while in the lowest level of service vehicles behave conservatively and cross the intersection given the right to the vehicle in the right lane. In this case there is no explicit cooperation, but coordination based on predefined rules. The safety kernel enforces the level of service according to the data validity and ability to timely communicate. For the highest level of service we consider the first-come first served approach to decide who crosses the intersection first. The symmetry is broken using the right hand rule, i.e., roads have distinct priority. Indeed, such symmetry breaking techniques are essential especially at low service levels for which there are no communication-based coordination guarantees. We based the highest service level on [20].

For this test case, we consider an intersection of the roads that have a single pair of conflicting directions. We divide the area in two regions, *critical zone* and *negotiating zone*. The critical zone determines the area of high risk of collision. Therefore, it corresponds to the intersection, as well as, the road sections next to the intersection. The road sections of the critical zone are at least the distance that every vehicle requires to stop completely without entering the intersection. The negotiating zone determines the road sectors next to the critical zone that are in the communication range. We refer to these two zones as the vital zone. The algorithm details appear in [5].

3) Coordinated Lane-Change: This application coordinates a lane-change maneuver by adjusting the inter-vehicle distances in the subject lane. The highest service level maintains short inter-vehicle distance while the maneuver is performed meanwhile the lowest service level keep longer inter-vehicle distance. The safety kernel enforces the service level according to the data validity.

We assume that the vehicles are able to follow the paths to change lanes and, when it needs to abort the maneuver, return to the original lane [18]. The algorithm details appear in [5] and it considers the accessibility of (primitive) applications for maintaining appropriate inter-vehicle distance, e.g., adaptive cruise control and vehicular platooning. The algorithm considers a vehicle that is changing lane to a *subject (target) lane*, as well as the vehicle projection on the subject lane. We define the *projection* as the intersection between the bisector ray of the vehicle position and the subject lane.

The main idea of the algorithm is that every vehicle projects the vehicle that is aiming to change lanes into the target lane. Then, all the vehicles involved in the maneuver perform an adaptive cruise control or platooning on the target lane with the projected vehicle according to the information quality and level of service that the safety kernel determines. Thus, they safely prepare for the maneuver by opening space. In the highest level of service, the aiming vehicle starts changing lanes if the inter-vehicle distance between the virtual vehicle and the vehicles in the target lane is at least two times the headway that corresponds to the highest level of service. In the lowest level of service, the aiming vehicle only starts changing lanes if the inter-vehicle distance between the virtual vehicle and the vehicles in the target lane is at least two times the headway that corresponds to the lowest level of service. While they are performing the maneuver, they check for hazardous situation. If they detect one that can avoid completing the maneuver, they abort it. Otherwise, they will complete it.

C. Functional components

In this work, we base vehicular cooperation on the ability to exchange remote sensory information and create a Local Dynamic Map (LDM), as well as agreeing on the cooperative service level according to which the driver manager sets the functionalities' performance level.

1) Positioning via a Local Dynamic Map (LDM): This component provides location information about the system objects, such as vehicles and roads. The position estimator implementation is divided into two parts: the local and the global position estimator. The local position estimator the position of the vehicle using only onboard sensors and thus it can be implemented in a real-time manner. The abstract sensor combines the data of an onboard range sensor and an onboard odometry sensor. An ultra-wide-band radio communication is used for indoor and outdoor localization. Such a radio can determine the distance to any other radio the range with a ranging request. The test-bed uses three radios, called *anchors*, with known fixed positions. The onboard range sensor sends ranging requests to the anchors in a periodic manner and returns the result, i.e., anchor id and distance, expected error and measurement time. Vehicles exchange their localization information, and use that for discovering the location of all nearby nodes. We control the access of these radio units using protocols that appears in [12]–[15], [19]. The position database contains position estimation for all vehicles from which sensor data has been received. The position database uses a Bayesian filter for each vehicle $v_i \in V$ to estimate the probability density function of v_i's position, speed and heading. The onboard odometry sensor facilitates the motor controller to get information about speed and steering angle.

2) Cooperative Evaluator of Service Level: The *cooperative evaluator of service level* is a fault tolerant distributed

component that allows the vehicles to agree on a join cooperative service level according to which they are to operate with respect to their cooperative vehicular applications. At each round, vehicles propose their maximum service level that they support and let the cooperative evaluator of service level to jointly decide on the performance service level that they have to apply during the next round. The implementation therefore has to reach such a decision within a bounded time over failure-prone communication links.

At the beginning of every round, the maximum local level of service in each vehicle is the lowest if it did not receive all votes from each other vehicle in the previous round. Then, it broadcast k times its vote (its maximum local service level) where k is a parameter that determines the number of mini-rounds. If the vehicle does not receive all votes from each other during the round, it sets its performance service level to the lowest, i.e., abort. Otherwise, it sets its performance service level to the minimum value among the received values in the current round, i.e., the cooperative service level.

The cooperative evaluator of service level has the following properties: (Agreement) all $members$ decide on the same value $CLoS = \min_{\forall v \in member}\{\max\{\mathcal{MLOS}_v\}\}$ with a probability that depends on the message delivery probability, (Validity) the value $CLoS$ was proposed by at least one vehicle v, and (Termination) all nodes decide (or abort) within a bounded time.

3) The Driving Manager: This component computes the instructions for the pilot actuator. It determines whether the vehicle follows autonomous or cooperative driving. The component consists in two sub components (cooperative driving and the autonomous driving) that are separated by the hybridization line.

The safety kernel determines the performance level and selects accordingly which output should be direct to the pilot actuator (which is dome using the data component multiplexer function of the safety kernel). When cooperation is not possible, e.g., due to communication problems, the safety kernel lowers the level of service to the lowest. In consequence, the selected driving function will be autonomous driving.

The driving manager deterministically determines the plans for $round_{now}+1$ with the information collected by the current round. The plans include the target (or trajectory) for all service levels that the vehicle supports. We introduce a special trajectory, called *transition trajectory*, that guarantees a safe transition from a higher service level to the lowest service level in a $transitionTime$ number of rounds. This trajectory is used in case that either the system fails due to communication, lack of agreements and/or low data validity, in the next round and, the safety kernel changes the performance level to the lowest one. To obtain the transition trajectory for the $round_{now} + 2$ in the worst case scenario, we assume that the vehicles will perform the highest service level in $round_{now} + 1$.

By the round phase, the vehicles exchange the local maximum level of service and agreement on the cooperative LoS before writing all the trajectories (driving targets, in Figure 7) to the *pilot multiplexer* within the safety kernel. The safety kernel controls which of the vlues sent to the pilot multiplexer is forwarded to the pilot (actuators), according to the data validity. We require to the *autonomous driving* that when the performance level is lowered to the lowest, it sends to the *pilot multiplexer* the transition trajectory for $transitionTime$ rounds.

D. Safety Rules

In this section we demonstrate the ability of the safety kernel to facilitate the implementation of cooperative advance driver assistance systems without compromising safety. Our intention is not to design new cooperative advance driver assistance systems, but show that the safety kernel can be used as a mechanism to achieve safe cooperative operation in spite of communication failures. We consider two service levels (an autonomous level of service and a fully cooperative service level) in three cooperative functionalities presented in Section VI-A.

The algorithm for each test case takes the input from the LDM and returns the trajectory plan for each service level. The algorithms use constants listed in Table I and assume that the errors in the LDM are bounded while the safety kernel uses the safety rules depicted in Table II:

Parameter	Description
$maxAcceleration$	Maximum vehicles' acceleration capability (bound)
$maxDeceleration$	Maximum vehicles' deceleration capability (bound)
$maxSpeed$	The road speed limit
$cruisingSpeed$	The cruising speed that all vehicles are aiming at. Vehicle's speeds may temporarily exceed their $cruisingSpeed$ but never $maxSpeed$
$length$	Vehicle length
$autonomousLoSHeadway$	Minimum headway required in the autonomous level of service
$cooperativeLoSHeadway$	Minimum headway required in the highest level of service
$transitionTime$	The transition time (number of rounds) from the highest service level to the lowest one

TABLE I. ALGORITHM CONSTANTS

Level of Service	Conditions
Highest $Local_LoS$	(1) Onboard localization received timely, (2) Remote localization received timely, (3) Cooperative evaluator receives timely
Highest $Cooperative_LoS$	(1) $Local_LoS$ is the Highest, and (2)Agreement reached timely

TABLE II. SAFETY RULES

E. Demonstration

The validation of our applications (Section VI-A) requires at least three vehicles. For the functionalities of adaptive cruise control and vehicular platooning as well as lane change we consider a scaled test road in the shape of a loop (without intersection). For the coordination intersection crossing, we consider a scaled test road with a single intersection in a shape of an eight. One of the main challenges that we face is dealing with the inherent nature of communication that is prone to ambient noise and interferences. Although these challenges become even more evident as the number of vehicles increases, we were able to implement the system design presented in this paper. We observed that the vehicles adapt their manoeuvres and operation according to the joint level of information quality.[1] We plan to include in our demonstrations different

[1]Demonstration videos at www.chalmers.se/hosted/gulliver-en/documents

fault injection techniques for manipulating the validity of the sensory information by causing increased packet drop and communication delays. Thus, we plan to complete the demonstrations that the scaled vehicles can safely perform the applications (Section VI-A).

VII. Conclusion

This paper describes an architectural pattern that may be generally applied for developing safe cooperative systems, addressing the problem on how to exploit cooperation for increased performance while dealing with the negative impacts that uncertainty could cause to safety assurance. The approach relies on the possibility of adjusting how cooperative functions are executed, so that the service level for each functions will always be adequate with respect to the uncertain operational conditions, namely timeliness and validity of sensor data. Safety is achieved because in design-time each function is proved to be safely executed in each level of service, when a certain set of safety conditions is met. And in run-time, a safety kernel adjusts the system operation as necessary to ensure that each function is executed with the service level for which the safety conditions are satisfied.

The approach is being implemented and demonstrated in the scope of the KARYON project. The paper described the solution that was developed to implement the safety kernel, which makes use of a dedicated hardware board on which safety kernel components execute. This modular approach was adopted for allowing a simpler integration of the safety kernel in the Gulliver test-bed. The paper provides an example application of the architectural pattern using this Gulliver test-bed. It is used to demonstrate some of the concepts and solutions developed in the project, in particular the architectural approach and the benefits in terms of overall functional performance.

Acknowledgment

This work was partially supported by the European Unions Seventh Programme for research, technological development and demonstration, through project KARYON, under grant agreement No. 288195, and by FCT, through the Multiannual program.

References

[1] Aeroflex Gaisler A.B. *GRLIB IP Library User's Manual*, April 2014.

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

[3] Christian Berger, Erik Dahlgren, Johan Grunden, Daniel Gunnarsson, Nadia Holtryd, Anmar Khazal, Mohamed Mustafa, Marina Papatriantafilou, Elad M Schiller, Christoph Steup, et al. Bridging physical and digital traffic system simulations with the gulliver test-bed. In *Communication Technologies for Vehicles*, pages 169–184. Springer, 2013.

[4] Tino Brade, Sebastian Zug, and Jörg Kaiser. Validity-based failure algebra for distributed sensor systems. In *SRDS*, pages 143–152, 2013.

[5] Antnio Casimiro, Oscar Morales-Ponce, Thomas Petig, and Elad M. Schiller. Vehicular coordination via a safety kernel in the gulliver test-bed. In *13th Int. Workshop on Assurance in Distributed Systems and Networks (ADSN'14)*. IEEE, 2014. to appear.

[6] António Casimiro, Jorg Kaiser, Elad M Schiller, Pedro Costa, José Parizi, Rolf Johansson, and Renato Librino. The karyon project: Predictable and safe coordination in cooperative vehicular systems. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–12. IEEE, 2013.

[7] Pedro Nóbrega Da Costa, João Craveiro, Antonio Casimiro, and José Rufino. Safety kernel for cooperative sensor-based systems. In Henrik Lönn and Elad Michael Schiller, editors, *ASCoMS@SAFECOMP*. HAL, 2013.

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

[9] Rolf Johansson, Jörg Kaiser, António Casimiro, Renato Librino, Kenneth Östberg, José Rufino, and Pedro Costa. An architecture pattern enabling safety at lower cost and with higher performance. In *Embedded Real Time Software and Systems Conference, Toulouse, France*, 2014.

[10] Maziar E Khatir and Edward J Davison. Decentralized control of a large platoon of vehicles using non-identical controllers. In *American Control Conference, 2004. Proceedings of the 2004*, volume 3, pages 2769–2776. IEEE, 2004.

[11] 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.

[12] Pierre Leone, Marina Papatriantafilou, and Elad Michael Schiller. Relocation analysis of stabilizing mac algorithms for large-scale mobile ad hoc networks. In Shlomi Dolev, editor, *ALGOSENSORS*, volume 5804 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2009.

[13] Pierre Leone, Marina Papatriantafilou, Elad Michael Schiller, and Gongxi Zhu. Chameleon-mac: Adaptive and self-* algorithms for media access control in mobile ad hoc networks. In Shlomi Dolev, Jorge Arturo Cobb, Michael J. Fischer, and Moti Yung, editors, *SSS*, volume 6366 of *Lecture Notes in Computer Science*, pages 468–488. Springer, 2010.

[14] Pierre Leone and Elad Michael Schiller. Self-stabilizing TDMA algorithms for dynamic wireless ad-hoc networks. In Marten van Sinderen, Octavian Postolache, and César Benavente-Peces, editors, *SENSORNETS*, pages 119–124. SciTePress, 2013.

[15] Mohamed Mustafa, Marina Papatriantafilou, Elad Michael Schiller, Amir Tohidi, and Philippas Tsigas. Autonomous TDMA alignment for vanets. In *VTC Fall*, pages 1–5. IEEE, 2012.

[16] Roman Obermaisser and Hermann Kopetz, editors. *GENESYS: A Candidate for an ARTEMIS Cross-Domain Reference Architecture for Embedded Systems*. September 2009.

[17] Mitra Pahlavan, Marina Papatriantafilou, and Elad M Schiller. Gulliver: a test-bed for developing, demonstrating and prototyping vehicular systems. In *Proceedings of the 9th ACM international symposium on Mobility management and wireless access*, pages 1–8. ACM, 2011.

[18] Iakovos Papadimitriou and Masayoshi Tomizuka. Fast lane changing computations using polynomials. In *American Control Conference, 2003. Proceedings of the 2003*, volume 1, pages 48–53. IEEE, 2003.

[19] Thomas Petig, Elad Michael Schiller, and Philippas Tsigas. Self-stabilizing TDMA algorithms for wireless ad-hoc networks without external reference. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *SSS*, volume 8255 of *Lecture Notes in Computer Science*, pages 354–356. Springer, 2013.

[20] Gabriel Rodrigues de Campos, Paolo Falcone, and Jonas Sjöberg. Autonomous cooperative driving: a velocity-based negotiation approach for intersections crossing. In *16th International IEEE Conference on Intelligent Transportation Systems*, 2013.

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

[22] Steven E Shladover, Charles A Desoer, J Karl Hedrick, Masayoshi Tomizuka, Jean Walrand, W-B Zhang, Donn H McMahon, Huei Peng, Shahab Sheikholeslam, and Nick McKeown. Automated vehicle control developments in the path program. *Vehicular Technology, IEEE Transactions on*, 40(1):114–130, 1991.

[23] Paulo Verissimo and António Casimiro. The timely computing base model and architecture. *Computers, IEEE Transactions on*, 51(8):916–930, 2002.

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