A reference architecture for cooperative driving

Sagar Behere[a,*], Martin Törngren[a], De-Jiu Chen[a]

[a] *Brinellvägen 85, Stockholm 10044, Sweden*

Abstract

Cooperative driving systems enable vehicles to adapt their motion to the surrounding traffic situation by utilizing information communicated by other vehicles and infrastructure in the vicinity. How should these systems be designed and integrated into the modern automobile? What are the needed functions, key architectural elements and their relationships? We created a reference architecture that systematically answers these questions and validated it in real world usage scenarios. Key findings concern required services and enabling them via the architecture. We present the reference architecture and discuss how it can influence the design and implementation of such features in automotive systems.

Keywords: Reference architecture, cooperative driving, autonomous systems, automotive embedded application, intelligent transportation systems

1. Introduction

The road transportation system today faces several challenges. The demand for road transportation is increasing, while there is a simultaneous need to reduce its environmental impact and achieve better control over road congestion. The use of Information and Communication Technology (ICT) presents an excellent opportunity to tackle these challenges through the use of novel Intelligent Transportation Systems (ITS) solutions. Advanced ITS solutions act as enablers for technologies that can play a key role in solving transportation problems. For example, enabling a vehicle or road infrastructure to communicate its location, intention or other information to its surrounding vehicles or nearby infrastructure, using wireless media, would allow innovative features like cooperative driving. With cooperative driving facilities, a vehicle can use the communicated information for adapting its own motion to the current traffic situation. Benefits of cooperative driving include improvements in the efficiency and safety of traffic flow, reduction of traffic congestion, reduction of fuel consumption and associated positive environmental and economic impacts. Additionally, cooperative driving systems can be used to enforce legal requirements

*Corresponding author

Email addresses: behere@kth.se (Sagar Behere), martin@md.kth.se (Martin Törngren), chen@md.kth.se (De-Jiu Chen)

like speed limits, obedience of traffic lights as well as for active safety functions like collision avoidance via trajectory control.

1.1. Research motivation

The potential benefits of ITS solutions in general and cooperative driving systems in particular, are strong motivators for research in this field. In order to reach sustainable solutions, a systematic approach is required for the design and realization of system functionality. This involves, among other things, the integration of new features into existing or legacy vehicle architectures and the guarantee of robustness and performance. One approach to this is through reference architectures. A reference architecture is described in [1] as "*.. in essence, a predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed, and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use. Often, these artifacts are harvested from previous projects.*" A reference architecture specifies the services and key architectural elements for achieving needed functionality and quality goals. It is a standard conceptualization, which can be instantiated in different ways to create a specific architecture for a particular system. Reference architectures are commonly being adopted for enabling better control of system integration and achieving non-functional goals such as reusability, understandability, etc. [2]

1.2. Contribution of this paper

In this paper, we present a reference architecture for implementing a cooperative driving feature in a modern automobile. The architecture fits into and extends the existing vehicle architecture in a minimally invasive manner. It provides a clear definition of needed services and the architectural elements that realize them. There is a good separation of concerns through modularization, enabling the compartmentalization of related data handling and control functions into hardware and software modules. This permits domain experts to focus narrowly on their specific part. Furthermore, possible errors are isolated and contained within respective architectural modules. The reference architecture can be used to implement various cooperative driving applications like vehicle platooning or convoying. An example instantiation of the reference architecture on a heavy duty commercial truck is also presented in this paper. The instantiation was used in the Grand Cooperative Driving Challenge (GCDC) 2011 [3]. Thus, the reference architecture has been validated in real world conditions.

1.3. Related work

Research in cooperative driving can be broadly split into two categories. The first category comprises of focused research primarily within the areas of automatic control, wireless communication and smart transport infrastructures. This research aims to develop the key knowledge and technologies needed to make cooperative driving possible. The second category consists of the research

needed to integrate the individual technologies and develop comprehensive cooperative driving solutions. Big research projects may include or combine both categories. This section points to relevant research in roughly the following order: impact of intelligent cooperative driving systems on traffic flow, selected results from individual technology areas (i.e. the first category, as discussed above) and finally, integration efforts relevant for entire systems (i.e. the second category, as discussed above).

A literature survey on traffic effects of automated vehicle guidance (AVG) systems is presented in [4]. Three different stages of development for AVG systems are described and the impact of each is presented. Of particular relevance are the so-called stage 3 AVG systems, which include intelligent infrastructure, for example, road-vehicle communication. For such systems, the estimated improvements in traffic capacity are mentioned to be 100 to 200%, which is claimed to correspond to a lane capacity of 4000 to 6000 vehicles/hour/lane. The effects of cooperative adaptive cruise control (CACC) on traffic flow characteristics are presented in [5]. One of their conclusions is that a high CACC penetration rate ($> 60\%$) appears to improve traffic stability and throughput in high traffic volume conditions. There is a reduction in the number of shock waves and standard deviation of speed on a traffic link. This is attributed to the reduced time gaps and improved string stability due to CACC.

A survey of inter-vehicle communication (IVC) systems is presented in [6]. Different applications of IVC are presented (e.g. collision avoidance) and options for short range communication technologies and transport and security layers are presented. Another survey of IVC and its application to intelligent vehicles is given in [7]. It presents the use of radio and infrared waves as the communication media and also looks at some inter-vehicle communications protocols. A categorization of IVC applications and comparison of their communication requirements as well as IVC protocols are covered in [8]. Some communication system requirements and related scenario specific challenges for autonomous and cognitive automobiles are given in [9].

A historical review on the longitudinal and lateral control of autonomous vehicle motion is given in [10]. The motions covered are car following, lane keeping, lane changing and the subsequent longitudinal and lateral controls and their integration. An overview of issues related to communication and control in networked multi-vehicle systems is presented in [11]. The identified issues are related to coordination and control strategies, formalization of control architectures and real time scheduling. Algorithms for lateral and longitudinal vehicle control in a cooperative driving scenario with intervehicle communication are described in [12]. The lateral control algorithm described there is based on dead reckoning functions with a differential GPS. The longitudinal control algorithm is based on the distance to the vehicle ahead, as measured with a laser radar and also calculated from the localization data received from the vehicle. Some practical results of a longitudinal control concept for truck platooning are provided in [13]. The concept uses a two stage controller, where the first stage consists of an acceleration control loop that linearizes the nonlinearities of the vehicle drivetrain. The second stage controller is a platooning controller designed with

sliding mode techniques. A theoretical framework for analysis and design of a CACC system is presented in [14]. In this paper, it is shown how feedforward control enables small inter-vehicle distances while maintaining string stability. Similar results are also presented in [15]. Yet another description of control structures, lateral and longitudinal control is given in [16]. Details of the development of an autonomous, intelligent cruise control system and as well as its comparison with human driver models is given in [17]. The transient response of the system was found to be superion to the human models, leading to smoother and faster traffic flow. A comprehensive description of an automated Intelligent Vehicle/Highway System (IVHS) is given in [18]. The paper addresses the range of driving functions that can be automated and the degree of automation, the decomposition of these functions into control tasks and the division of intelligence between the vehicle and highway infrastructure. A control architecture of an Automated Highway System (AHS) is presented in [19]. It also discusses the design and safety verification of the on-board vehicle control systems.

A survey and tutorial on requirements, architectures, challenges and standards related to vehicular networking is given presented in [20]. The paper also provides an overview of current and past major ITS efforts in the USA, Japan and Europe. An overview of adaptive and cooperative vehicles, their main applications, pros and cons is provided in [21]. The paper looks at the SAFESPOT project [22] as a case study. A somewhat dated, but still relevant survey of intelligent vehicle applications is presented in [23]. Challenges of operating vehicle platoons on unmodified public motorways are described in [24]. New developments and trends for intelligent vehicles are identified in [25]. It is concluded there, that information sharing between vehicles and between vehicles and the infrastructure is the most attractive trend in intelligent vehicle research.

A three layered architecture for cooperative driving of automated vehicles is presented in [26]. The architecture includes two layers in the vehicle and one layer in the infrastructure. An overview of the cooperative driving system for automated vehicles in the Demo 2000 project together with the driving scenario is given in [27].

Several large European projects have been directly or indirectly involved in cooperative driving. The Cooperative Vehicle Infrastructure Systems (CVIS) [28] is a big project that attempted to create a unified technical solution allowing all vehicles and infrastructure elements to communicate with each other. The project defined and validated an open architecture and system concept for a number of cooperative system applications and created an open application framework that can run on the vehicle and roadside equipment. A closely related project to CVIS is SAFESPOT [22, 29]. This project shares the CVIS infrastructure and some of its technical solutions, but has a greater emphasis on safety and safety related applications. Among other results, the project has developed key enabling technologies for ad-hoc dynamic networking, accurate relative localisation and dynamic local traffic maps. The CO-OPerative SystEms for Intelligent Road Safety (COOPERS) project [30] focused on the development of telematics applications intended to provide vehicles and drivers with real time local situation based, safety related status and infrastructure status

information. It aimed to extend the concepts of in-vehicle autonomous systems and vehicle-to-vehicle communication (V2V) with tactical and strategic traffic information. A comparative synthesis of CVIS, SAFESPOT and COOPERS is given in [31]. The Highly Automated Vehicles for Intelligent Transport (HAVE-IT) project [32] had a twin focus on highly automated driving and developing a safety architecture. The project results include development and validation of next generation advanced driver assistance systems (ADAS), an electronic "co-pilot", as well as a scalabe and safe architecture with advanced redundancy management. The Safe Road Trains for the Environment (SARTRE) project [33] has developed strategies and technologies that enable vehicle platoons to operate on normal public highways. These large European projects have developed domain expertise related to sensors, algorithms and control, and also architectures for integrating it. The architectures vary widely, with some being technology prototypes only while others are relatively closer to production, because they utilize the standard tools and components available in the automotive industry. The individual project websites contain a lot of information, however many architectural details are often not openly published. Support for formal verification and validation of cooperative driving systems are usually compiled as annexes, in a project's publications.

Compared to the progress in domain knowledge, there is a relative lack of research on architectural aspects, especially in the context of extending and integrating into existing vehicle architectures. The aim of the current work is to address this deficit. The reference architecture presented here provides a research perspective, while the instantiated architecture provides a practical use case of the research results.

1.4. Organization of this paper

This paper is organized as follows: Section 2 presents some technical considerations in the design of cooperative driving systems. Section 3 describes the reference architecture. The description commences by clarifying the context and scope of the reference architecture, followed by a description of the services the architecture must provide. Then, the main architectural elements are introduced via a conceptual view, followed by an elaboration on each element. Section 5 presents an instantiated example of the reference architecture. The instantiated example is described with the aid of two different architectural views. Finally, section 6 contains a discussion of the reference architecture. It is shown how the reference architecture takes the technical considerations of section 2 into account and how it can be used to provide the functionality needed to fulfill cooperative driving scenarios. Next, it is shown how the reference architecture relates to the AUTOSAR standardized software architecture. This is followed by a comparison with some autonomous system architectures, where it is shown that the reference architecture follows established principles of constructing autonomous systems. The discussion concludes by summarizing the architectural highlights and future work.

2. Technical considerations

2.1. Characteristics of cooperative driving systems

A cooperative driving system has little, no or mostly unpredictable control over the behavior of other vehicles in the vicinity. Expected behavior may be indicated by regulations and legislation, but from a purely technical point of view, it is a safety hazard to assume that all other systems will conform to and abide by the rules. The only certainty is the ability to control the ego vehicle and broadcast information about it. Despite this, it is the responsibility of all cooperative driving systems to maintain the integrity of the traffic flow in the vicinity.

A cooperative driving system inherits general characteristics and software specific challenges of all automotive systems. Such characteristics and challenges are related to distributed and heterogenous software, middleware, error diagnosis and recovery, software reuse, resource mangement, complexity and model based development and are described in more detail in [34, 35, 36, 37]. Additional characteristics specific to cooperative driving or those having a direct impact on architecture are listed in this section. Some of the characteristics are more relevant to an instantiated architecture, rather than a reference architecture.

2.1.1. Functional characteristics

These characteristics are related to the specific behavior or functions of the cooperative driving system

1. A cooperative driving system is characterized by distributed, hierarchical control. The driving controller takes decisions and creates set-points for high-level motion variables, like speed, acceleration etc. However, the actual regulation of these variables occurs in controllers that are completely distinct from the platooning controller. For example, the cooperative driving controller may demand a certain amount of deceleration. It is the task of the brake controller to actually decelerate the vehicle.
2. A cooperative driving system needs to perceive the environment and form a world-view from fairly limited sensor input[1]. Therefore, it is necessary to have a lot of trust in sensor data (e.g.: There really is no vehicle ahead).
3. A cooperative driving system is closely related in concept to an autonomous system. Therefore, the presence of a system ego is needed in some form or the other. In other words, there must be a system component that knows what functions the system-as-a-whole is supposed to perform, and how those functions are done by the system.
4. A cooperative driving system needs a human machine interface (HMI) that can be used to make the human driver aware of what the system is doing.

[1] as compared to the perception abilities of a human driver.

2.1.2. Extra-functional characteristics

These characteristics are more about the qualities of the system and are related to overall system operation, rather than specific behaviors.

1. Most sensors used by a cooperative driving system provide reliable data only under a subset of circumstances that occur while driving cooperatively. Therefore, redundancy, multiple sensors based on different principles, fusion and above all, reasoning on received data is unavoidable.
2. Cooperative driving involve an inherent safety paradox. Ideally, a system should not depend on external input to assure its own safety. However, a cooperative driving system must do so. It is a design challenge to make the system as safe as possible, without making assumptions about the quality and trustworthiness of incoming data.
3. In addition to the accuracy of input data, it is also necessary at all times to maintain the accuracy of the ego vehicle data being broadcast by the cooperative driving system to the surroundings, because this is used by other vehicles in the vicinity.
4. The cooperative driving system should ideally be able to determine which of its constituent components are working properly and use that as a basis for system behavior. Thus, it should be possible to gracefully degrade functionality based on the health of monitored components
5. A cooperative driving system is made up of a mixture of tasks. Some tasks have hard real-time constraints, some are soft realtime, while others are not generally time critical. Time criticality is sometimes directly linked to safety criticality.
6. A cooperative driving system mixes safety critical and non-safety related functions. A strict awareness needs to be maintained of this fact when developing the architecture, because the impact goes beyond obvious technical matters and affects processes like certifications.

2.1.3. Miscellaneous characteristics

1. Cooperative driving is developed as an optional feature to an existing vehicle platform. Therefore, the onus on the architect is two-fold. The cooperative driving system
   (a) should conform to the existing architecture, its possibilities and limitations as far as possible
   (b) should minimize required changes to the existing system. If any changes are made, their long term impact must be clearly understood
2. The design of a cooperative driving system is significantly affected by the sensors used to obtain a "world-view". For example, a system using a camera would differ from a system using a radar, which in turn would be different from a system using a bank of lasers. It is therefore important to have a flexible data fusion component, yet keep the rest of the system invariant to its changes.

3. Since the vehicle already has an HMI, which is part of its existing architecture, decisions need to be made on the HMI specification of the cooperative driving system and whether it can fit into the existing HMI scheme
4. Cooperative systems integrate domain expertise. Therefore, the architect needs to have sufficient knowledge of all domain functions; mere knowledge of functional and/or behavioral requirements is not sufficient to create the architecture.
5. The development, testing and verification activities for a cooperative driving system are entirely governed by the datalogging and visualization infrastructure. Although not a part of the final product, good datalogging tools and frameworks are crucial during the development phase. As such, the architecture must have native support for datalogging.
6. Data, control and computation need to be handled separately within the architecture. A cooperative driving system is a mix of time and event triggered control. The specification and design of inter-component interfaces is driven primarily by data, while the scheduling of the components and their priorities are driven primarily by control and computation requirements
7. A cooperative driving system requires heavy parametrization and the architecture should support live calibration methods so that parameter values can be correctly decided during live tests of the system
8. The development of cooperative driving systems is practically always a mixture of visual programming methods (Simulink [38] etc.) and hand coding. The architecture should enable smooth integration of the artifacts produced by both methods.
9. The functionalities required can be classified as low level (e.g.: input/output functions) and high level (e.g.: algorithms, mode management etc.). This has an impact on the execution environments and therefore the architecture should segregate them into distinct blocks which can be executed on different execution environments.
10. A cooperative driving system's functions can also be classified as relatively static or dynamic in nature. Static elements (e.g.: i/o blocks) are relatively fixed in the sense that their inputs, outputs, content and behavior are well-defined. Dynamic elements have greater dependence on the nature and type of their inputs and their behavior cannot be rigidly predicted in advance (e.g.: world models, control states). The architecture should avoid mixing together static and dynamic functions in the same architectural element.

2.2. Execution environments

Software execution environments and associated development processes for embedded systems of the scale and magnitude of cooperative driving systems can be broadly divided into two categories.

The first category is that of dedicated microcontrollers executing code that is auto-generated from models like those made in Simulink. This is the de facto standard in the automotive industry, since the complete solution is created in the domain of the problem (control engineering, signal processing etc.), using the skills available to the engineers with domain expertise. Occasionally, the coding may be done by hand, but the key point and distinction here is that the dedicated microcontroller has limited requirements on general purpose computations.

The second category is that of general purpose computer hardware, executing general purpose operating systems like Microsoft Windows, or Linux. Using such execution environments is a relatively recent trend, and they are generally restricted to use cases where the former category is sub-optimal, or simply can not be used. When using general purpose operating systems, the code is often hand-written (although there is an increasing trend of integrating auto-generated code as well). Hand-written code necessarily implies that a solution developed in the problem domain must be manually transferred to the computer programming domain and this transformation in itself is often a cause of errors.

It is the responsibility of the architect to decide the appropriate environment or mixture of environments for the specific system under design. This decision can have a lasting impact on the development process as well as the product being developed. The first aid to decision making should be the nature of the problem being solved. Certain problems, like the development of control components is more naturally done in a model based development environment, like Simulink. The runtime requirements of control components are also very strict with regards to real-time capabilities, scheduling, computation times etc. Therefore, it is often best to execute such components on dedicated microcontrollers, using code auto-generation. This assures that the model will execute exactly as intended and there will be little unanticipated interference from other software tasks. Components with more lenient runtime requirements, graphical displays, databases, datalogging and background tasks, TCP/IP or internet communication etc. can be more easily coded up in a general purpose programming language like C/C++ and this is the argument for usage of more general purpose computing. Hybrid solutions can also be considered where applicable. For example, it is possible to include hand-written code in Simulink blocks, and at the same time, it is possible to use auto-generated code from Simulink in a large, hand-coded software framework.

Very often, the choice is not made on technical merits alone. The programming skills in the team can dictate the development method. A good architect can take team skills into consideration and design solutions that the team can efficiently deliver. In other scenarios, it is possible to use one approach for rapid prototyping and proof of concepts. Later, other approaches can be used for production level systems.

3. Reference architecture

This section describes the reference architecture, beginning with its scope within the larger automobile architecture. Architectural details are then presented in gradual increments, to aid easy comprehension. The reference architecture is independent of particular implementation technologies, tools and frameworks. It is an abstract set of solution considerations, requirements, patterns and guidelines, which are then concretized during an individual instantiation.

3.1. Context and scope

The electric architecture of a modern automobile can be logically seen as a network of sensors, actuators and electronic control units (ECUs) [39]. The network backbone is typically a bus like CAN, or for upcoming architectures, FlexRAY. The sensors and actuators may either be directly connected to the network buses, or they are connected to the ECUs and their associated data is available as messages on the network.

Physically, the sensors, actuators and ECUs are geographically distributed around the automobile. The locations are constrained by various factors including function, operating environment constraints, EMC regulations, accessibility etc. A large amount of cabling distributes power and network signals from the sources to the sinks.

Features and functions of the automobile are then realized via a combination of sensors, actuators and ECUs. The ECUs execute software which uses sensor data, performs computation and control tasks and influences the actuators. For example, a simple cruise control system reads information about the current vehicle speed and tries to maintain a set-point speed via control of the engine.

Addition of simple new features to the automobile is done via addition of software code to existing ECUs. More complicated features, which require additional sensors/actuators or more computational resources than are available in the existing ECUs, are implemented via the addition of one or more ECUs to the vehicle network. The new sensors/actuators are then either directly connected to the network or to the newly added ECUs. Often, the newly added ECUs simply control existing actuators in novel ways, via actuation messages sent over the vehicle bus.

Within this physical (hardware+software) context, a cooperative driving feature can be realized by adding a system of sensors, actuators and ECUs to an existing vehicle network (Figure 1). The reference architecture described in this paper is constrained in scope to such an additional system. Note however, that the reference architecture is described in terms of logical architecture (elements, information flows, etc.) because reference architectures, by their very nature, refrain from dictating physical implementations.

3.2. Services needed in a cooperative driving system

This subsection describes the services that need to be present in a cooperative driving system. These services are either already present in the vehicle, or they need to be introduced by the cooperative driving system. In both cases, it

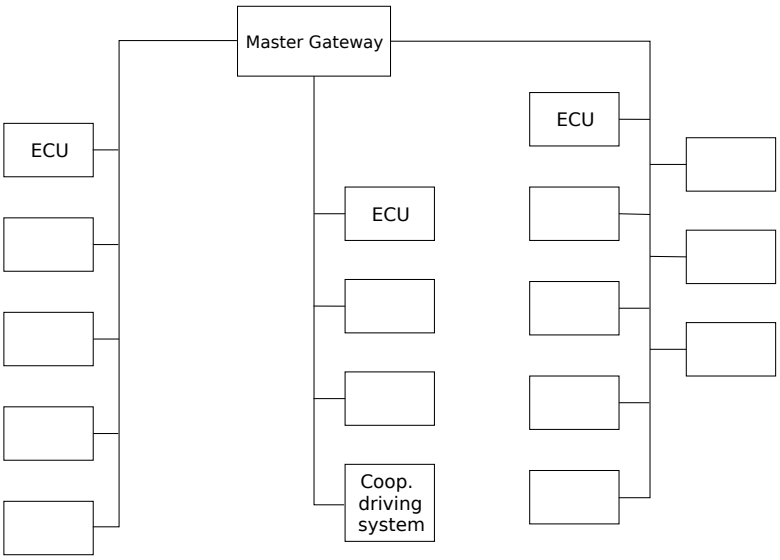

Figure 1: Vehicle network as a context for the reference architecture

is the task of the architecture to enable smooth integration/introduction of the services and associated dataflows. The services need not be explicit architectural artifacts, nevertheless their existence is still essential in an implied form.

Positioning is at the core of a cooperative driving system. The positioning service provides the location of the ego vehicle and other objects in the vicinity with the required accuracy. Ego vehicle position can be obtained using GPS technology and the GPS signals are often fused with data from inertial sensors in the vehicle and other positioning information obtained over the wireless. Positions of other vehicles and road objects can be obtained via wireless communication and sensors like radar, cameras, lidar etc. It is also often necessary to provide all positions as coordinates on a map, and when this is done, the service is called 'map matched positioning'.

Clock synchronization services are needed because the data packets exchanged between the cooperative driving systems need to be timestamped. The timestamps need to be synchronized across all vehicles. For the timestamps to be synchronized, all the system clocks should be synchronized to a common clock source. Typically, the clock source is the GPS time signal. Periodic synchronization with the GPS time is necessary due to the inevitable drift in clock mechanisms of any electronic device.

World modeling is the creation and updates of the state of the world in an area of interest around the ego vehicle. This includes other vehicles and

their positions and intentions as well as the state of the infrastructure like current speed limits, lanes, surface conditions, traffic signals and so on. Strictly speaking, the cooperative driving system operates not in the world as the human driver sees it, but in a virtual world synthesized from the sensor data stream a.k.a the world model. The world model needs to match reality as closely as possible, for the cooperative driving system's actions to be acceptable. Further details of world modeling for cooperative intelligent vehicles is given in [40].

Wireless communication is necessary when at least one of the communicating devices is mobile. This is the case with cooperative driving systems. The term, 'wireless communication' encompasses a broad range of technologies and protocols. Some of these are in the process of becoming standards in the field of Intelligent Transport Systems (ITS) [41].Wireless communication services should provide abstractions via which the cooperative driving system can send and receive data with the outside world.

Vehicle interface is the mechanism via which the cooperative driving system exerts influence on the vehicle motion. This is the interface via which data can be read from and sent to other ECUs. It is the gateway to all other vehicle functions.

Control services are needed to obtain set-point values for the vehicle motion actuators. These services enable the cooperative driving system to answer the all important question, "What should the system be doing right now?" In addition to managing the current vehicle motion, the control services may also be responsible for future planning and alternative behavior in case of errors and emergencies. All the other services described so far are essentially helpers to the control services.

Supervision services are those that are ultimately responsible for the correct functioning of the cooperative driving system. They start and stop the other services as necessary, monitor the system for signs of unusual behavior and perform other supervisory tasks. They are needed because (semi) autonomous systems today are not at a stage where perfect behavior emerges out of the interaction of constituent functional components. Informally, it is the supervisory services that maintain the controlling stake in the system.

HMI services are needed to provide an insight into the system, as it operates. Especially for critical systems like cooperative driving, it can be unnerving for the human driver to not know what is going on. HMI services are generally always present in the architecture of a modern automobile. The cooperative driving architecture needs to integrate and non-destructively extend the available services in the vehicle.

Diagnostics, error handling and recovery services are needed in any large and complex system. Systems fail, software has bugs and faults occur.

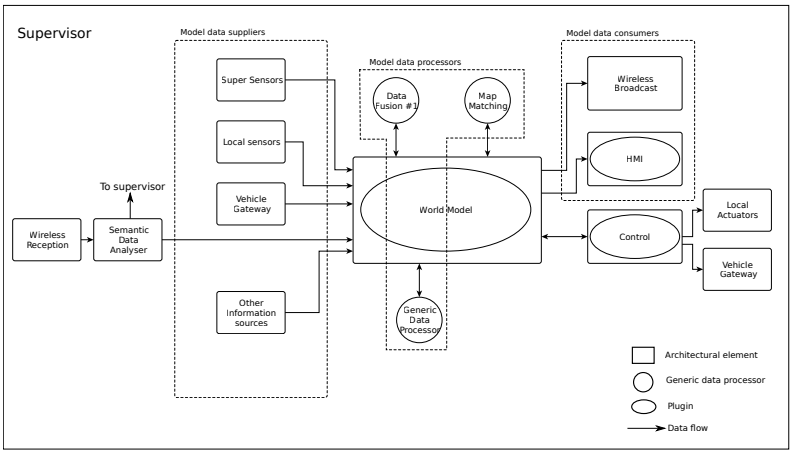

Figure 2: Conceptual view of the reference architecture

These are facts and it is necessary to have well-defined means to recover from error conditions and diagnose/troubleshoot the system in event of failure. These service assume greater importance during the system development and calibration phase and should therefore be built right into the architecture, rather than being added as an afterthought.

All of the services above are enabled by the reference architecture.

3.3. Conceptual view of the reference architecture

An architecture can be described with the aid of views. A view is defined by ISO 42010 as "A representation of a whole system from the perspective of a related set of concerns." There are several types of views which can be collectively used for a comprehensive architecture description [42, 43]. We use the conceptual view described in [42] to describe the reference architecture, since it corresponds well to the abstraction level at which the reference architecture exists. The conceptual view describes the system in terms of its major design elements and the relationships among them. It is independent of implementation decisions. This is a reasonable view for our reference architecture, since it captures the essence of the architecture at a high level of abstraction. Implementation specific architecture is created during instantiation of the reference architecture and an instantiated example is described in section 5. The conceptual reference architecture is shown in Figure 2.

The architecture centers around a so-called world model and some elements which are categorized based on their interaction with the world model. There are also additional elements that do not directly interact with the world model, but nevertheless have specific functional roles. The entire architecture and its functionality is ultimately supervised by a single, special element that constitutes the identity of the system, from the viewpoint of the system's context.

In Figure 2, the squares indicate architectural elements with specific roles. The circles indicate elements capable of running generic data processing algorithms (e.g. band pass filtering) on the data in the world model. The arrows indicate data flow. Bi-directional arrows indicate bi-directional dataflow between the connected elements. The ellipses indicate plugins (placeholders), which can be executed by the architectural elements for fulfilling their functional role. Plugins enable a clean separation of algorithms and structure, since different plugins implementing different algorithms can be swapped in and out of the same architectural element.

A detailed description of the architectural elements now follows.

3.3.1. Key properties of all architectural elements

Before presenting the individual architectural elements, it is useful to understand key properties common to all architectural elements. These properties are independent of the actual role and behavior of each architectural element.

An element may support multiple modes of operation and can switch between these modes during operation. The transitions can occur either as a result of a request from another element, or it could be self-dictated.

Each element contains diagnostic facilities and interfaces. The element can either self-diagnose its errors and status, or can permit other elements to do so via a diagnostic interface. In all cases, information about the current health of the element is accessible on demand.

The elements support lifecycle management operations like starting, stopping, pausing, restarting and other introspective functions. Elements also have various types of data interfaces which can be connected to other elements to form a dataflow network among the elements.

In addition to data flow, all elements support synchronous and asynchronous command interfaces. These interfaces permit the elements to access and utilize functionality present in other elements.

3.3.2. The world model

The world model is a dynamic, formal representation of the states of the ego vehicle and the vehicles of interest in the vicinity of the ego vehicle. The state, in the context of cooperative driving, consists of those variables, the knowledge of which is required for controlling the current and future vehicle motion. This includes values of current physical variables like vehicle velocity, acceleration, position etc. Also required would be variables indicating future intent of a vehicle. For example, intention to decelerate, change lanes or initiate an overtaking maneuver. Besides vehicles, the world model also stores information about states of road infrastructure. This could be information about traffic lights, which includes their position, current color and time to next color change. Other infrastructure information includes applicable speed limits, information about lane closures and hazard warnings. In addition to state information, the world model may also contain data and metadata associated with, or derived from the sensed/received information. Examples of metadata are identification numbers of detected vehicles, confidence estimates of received information etc.

The data in the world model can be extended to include not only the information about the external world, but also the internal "world" of the vehicle and the cooperative driving system. This would be the operating modes of the system, state of health of individual components, active errors etc. Finally, the model can be extended to include historical data as well as short term predictions of the current variable values. Predictions are necessary because information received over wireless media is susceptible to intermittent disruptions, or certain data sources may simply be unavailable (GPS reception, for example, gets disrupted under bridges or in tunnels). The exact information that needs to be present in the world model, as well as the required quality of that information, depends on the specific application for which the system is being designed. A good example of developing a world model can be found in [40], which describes a "Local Dynamic Map" developed for the SAFESPOT project[22].

The information in the world model could be hierarchically layered, with the raw sensor readings at the lower layers, which are combined and fused along successive upper layers to get high-confidence information about the world objects. It is important to point out that the world model need not perform the actual processing (combination, fusion etc.) of the data it contains. That can be done by external model data processors that interact with the world model. For example, a specific model data processor instance could contain a vehicle motion model which it uses to predict future values of vehicle velocity. The world model only needs to be a structured data pool or repository that performs the functions of holding data and providing access to the data it holds.

The world model assures the consistency, reliability and time related characteristics of the data it holds. Access to the data and metadata of the world model is via a rich variety of interfaces. The world model supports concurrent access, thereby enabling a multitude to external entities to simultaneously insert, modify and extract its contents.

The world model is implemented via plugins, which could use different mechanisms to store the information. For example, the data could be held in a distributed database while the world model would be in the form of a database schema. Alternatively, a geographical or topological map could be the basis, which is populated with objects and their attributes.

3.3.3. World model data processors, suppliers and consumers

Model data processors are generic elements that work with the data in the world model. They are responsible for processing, modifying, combining or otherwise transforming the information in the world model. They read input data from the world model and their output data is fed back into the world model. Examples of model data processors would be entities that fuse the raw sensor information in the lower layers of a hierarchical world model and write it into the higher layers. An example of this would be a model data processor that takes current velocity readings as provided by the GPS and sensors in the wheel, axle, transmission and fuses them to provide a single, high-confidence velocity estimate. Another example of a model data processor would be a function that performs "map-matching". Such a function would perform the task of

maintaining the geographic location attributes of world model objects, with reference to a specific map. This would involve reading the latitude/longitude information of each vehicle and transforming it into coordinates on the particular map. Model data processors are represented by circles in Figure 2, and the associated bidirectional arrows indicate that the world model is their primary source and sink.

Model data suppliers are elements that feed data to the world model. This could be local sensors, entities that extract data from ego vehicle gateways, entities that supply data obtained over the wireless interface as well as so-called super sensors. The local sensors are those sensors connected directly to the cooperative driving system and whose output is not available on the vehicle bus. For example, a sweeping laser rangefinder installed together with the cooperative driving system package. The ego vehicle gateway provides information about the ego vehicle, which is accessible over the vehicle's data network, like the CAN bus. This information is generally related to the vehicle speed, acceleration, engine torque, status of accelerator and brake pedals, steering wheel angle etc. Super sensors are those elements that preprocess information from sensors like camera, radar, lidar etc. and provide higher level information directly to the world model with high confidence estimates. An example of such information is the location of lane markings, which are extracted from the camera images. The model data suppliers generally have a write-only access to the world model.

Model data consumers are elements that behave as sinks to the world model. These elements generally have read-only access to the world model. Two examples of model data consumers shown in Figure 2 are the wireless broadcast and the HMI elements. The wireless broadcast element periodically reads ego vehicle state data (position, velocity, acceleration etc.) from the world model and sends it out over the wireless interface. Similarly, the HMI element reads data from the world model, and sends it to the vehicle HMI system, perhaps after some pre-processing steps. This data includes the current state of the cooperative driving system (active/inactive), detected traffic lights and speed limits, status of maneuvers currently being executed, detected vehicles etc.

In summary, data flows into the world model through the model data suppliers. It is transformed within the world model by the model data processors and it flows out of the world model and into the model data consumers.

3.3.4. Wireless and the semantic data analyzer

The wireless element is the abstraction for the physical wireless interface(s), together with all the associated transmission, routing and interaction protocols. The wireless interface assumes a role of great significance in a cooperative driving system, because it is the primary means of communicating with other vehicles and the infrastructure. This information includes the status data broadcasted periodically by the vehicles (speed, position etc.) and the infrastructure(speed limits, traffic lights etc.). It can also contain service announcements and other ITS scenario specific information. The information is categorized according to a taxonomy and only certain categories of information need to go to the world

model. Therefore, there is a need for investigating, classifying and appropriately routing the received wireless data content. This task is conceptually distinct from the task of receiving the data and is therefore fulfilled by a separate architectural element, the semantic data analyzer.

The semantic data analyzer peeks into the incoming data stream and is equipped with the resources needed to categorize the content according to predefined application specific taxonomies. This task necessarily involves a degree of semantic understanding of the data, and hence the name. After categorizing the incoming data, the semantic data analyzer then routes the data to the relevant architectural elements. For example, data related to the acceleration of the vehicle ahead will go to the world model. A request to join platoon, when the ego vehicle is functioning as platoon leader might be routed to the supervisor. In the reference architecture, it is anticipated that the incoming data will be routed either to the world model, or to the supervisor. Further details of what the supervisor does and why data should be routed there are provided in subsection 3.3.6.

3.3.5. Control

The control element in the architecture is responsible for the motion of the ego vehicle and all the decision making related to that process. The specific tasks of the control element are dependent on the nature of the cooperative driving task(s) the system is designed for. Consequently, the information needed by the control element is also task specific. It will however almost definitely include the variables reflecting the current motion of the vehicle (velocity, acceleration, braking torque, demand torque etc.) because there are needed for any closed loop control of the vehicle motion, which is the ultimate goal of the cooperative driving system. As an application specific addition, the controller for a platooning system may require some motion variables of the vehicles ahead, because taking these into consideration can lead to better control of phenomenon like string effects in the platoon. It will also require information about the infrastructure states (Red light, don't move. Or, max speed 50km/h). Regardless of the specific control tasks, it is reasonable to assume that the control system will influence actuators directly connected to the system, as well as those present in other ego vehicle subsystems. A trivial example of local actuation is the operation of lamps (During system development and demonstrations, there is often a requirement to operate blinking lights mounted on the vehicle when driving in automatic mode). More typically, the cooperative driving control element will send messages to other vehicle subsystems over the vehicle's data bus. These are speed, engine torque, vehicle acceleration and braking setpoints, which the existing subsystems are already equipped to respond to. The control system will also need access to the world model, in order to perform its function.

It is relevant to question why the control element isn't simply classified as a consumer of the world model. In a sense, it is. It can also influence the content of the world model and in that sense, it is a model supplier too. The reason it is listed separately in the description of this reference architecture is to highlight the crucial functional role it plays in the cooperative driving system.

It is not a simple data consumer (compare to HMI), nor is it a classical data source (compare to a radar sensor). The world model may be considered as the primary enabler in the system and the control is what it primarily enables. Furthermore, it is very often the case that the control element is implemented in distinct ways in the implementation architecture and having it as a distinct element in the conceptual architecture leads to a better mapping between the conceptual and implementation architectures. Beyond these arguments, the actual classification of the control element is of importance only to compulsive categorizers, and adds no value to the architectural description.

3.3.6. Supervisor

The supervisor is one of the most special parts of the reference architecture. Given the description of the architecture as a set of connected elements, the supervisor is akin to the canvas the elements are drawn upon. The following paragraphs describe the supervisor in different ways, in order to convey a sense of the supervisor's role.

It is the supervisor that encodes an understanding of the various architectural elements, their capabilities and limitations. Thus, it is the supervisor that is aware of the presence of the world model, the control and other elements and how they must function in order to generate specific behaviors of the cooperative driving system. The supervisor "knows" what behavior is expected of the cooperative driving system in a given context and uses them to achieve the expected behavior. The elements in turn pass on all unknown inputs, locally unresolvable errors and requests to the supervisor and expect instructions on how they should proceed.

The cooperative driving system can be considered as an autonomous system, within some tightly constrained definition of autonomy. Regardless of the constraints, all autonomous systems need a single ego, or Self which represents the entire system as a cohesive unit. The supervisor performs this role, and provides the system identity to its context.

The supervisor can query the other architectural elements for specific inputs and handle queries generated by the elements. Let us illustrate this with an example from the cooperative platooning system described in section 5. When the system receives a request from another vehicle to join the platoon, the semantic data analyzer marks this request as something that is not destined for the world model. It then gets routed to the supervisor, who in turn passes it to the control, because the supervisor is aware that the control can answer such a request. The control returns a yes or no response, which the supervisor sends off to the wireless broadcast element. Another example scenario where the supervisor is involved in the imposition of radio silence is described later in section 6.2.1.

From a connections perspective, the supervisor has data and control connections with all other architectural elements. It uses these connections to exchange data with the elements, and perform control actions like setting modes in the elements. Depending on the implementation technologies, the supervisor to element connections may have to be statically specified, or they may be

dynamically generated as needed.

In addition to arbitration tasks, the supervisor is also responsible for system level error management, diagnostics and health monitoring of individual architectural elements. It handles lifecycle issues like starting and stopping elements and the encoded knowledge of the system and its functioning can be utilized for graceful degradation of system functionality, as and when needed. The supervisor also performs mode management functions for the entire system and can cascade the system modes into appropriate modes for the individual elements. This refers to both system specific modes (e.g.: state of a processing node) as well as application specific modes.

In its role as a "catch all" for the system, the supervisor also performs an important practical architectural function. It is the place to implement all unforeseen system functionality. The need for such functionality arises in almost all practical projects and the solution is often to employ 'quick-and-dirty local hacks' that break the otherwise clean design. Our reference architecture acknowledges that such hacks are practically unavoidable and instead suggests the supervisor as a natural location to code them. Thus, by preventing dirty hacks from popping up all over the architecture, the supervisor provides a clean way to do dirty hacks. In due time, they can be merged into the appropriate architectural elements.

Examples of information used by the supervisor would be: Periodic health status of components, information about failures of specific sensors or entire elements, current operating mode of the controller, information about external demands like imposition of radio silence etc.

4. Guidelines for instantiation of the reference architecture

This section presents some guidelines for instantiating a *prototype* of the reference architecture. It attempts to answer the question, "What should a system architect think about, before attempting an instantiation of the reference architecture?". It is important to emphasize the prototyping aspect, wherein the instantiation is likely to revolve around the capabilites of available tools, libraries and frameworks. For production systems, there are a large number of techno-commercial, platform and supply chain considerations which often outweigh purely technical considerations.

4.1. Minimum data set

Before considering any instantiation, due regard must obviously be given to the availability of the minimum set of information needed for such a system to work. What is the minimum information set? The exact answer is of course application specific, but the minimum set should be evolved by assessing the bare functionality needed to make the application work. For example, in order to determine setpoints for the vehicle motion, is the required information about the current vehicle state available over the vehicle data bus? Next, the minimum requirement in a cooperative driving scenario is that the vehicle should

not collide with a vehicle in front of it. In order to do this, it is necessary to detect a vehicle ahead and have distance related information for it. Whether this is obtained via a local radar sensor or over the wireless is secondary (although still important). Next, the vehicle should respect prevalent speed limits and not jump red lights. This necessitates information about detected infrastructure states. If the cooperative driving scenario requires shifting of lanes, then information regarding the lane the ego vehicle is in, as well as the presence and availability of other lanes will be required. Thus, the minimum set of required information is necessarily dependant on the scenario requirements and it is not possible to arbitrarily specify a minimum set.

4.2. Mapping to physical components

How should the elements of the reference architecture be mapped to physical components? In the first step, the computing platforms and execution environments for implementing the elements should be determined. Based on the element implementation, the inter-element communication mechanisms can be chosen in the second step. Several iterations of these two steps may be necessary. All the inter-element communication mechanisms need not be of the same type.

4.2.1. Element implementation

We suggest that the following aspects should be considered for implementation of the architectural elements:

1. **Computational capacity:** This is the primary consideration when selecting a computing platform. Especially in the case of the world model and assorted model data processors, the computational capacity required can vary considerably depending on the the complexity of the algorithms being implemented. A simple in-memory key-value store has vastly different minimum requirements compared to a distributed, relational database with an SQL query interface. Similarly, some filtering algorithms can require heavy number crunching capabilities. A typical example is feature extraction from digital images. Such algorithms are also common in super sensors that require heavy signal processing(e.g. radar). The HMI can consume a surprising amount of processing power, especially if utilizing graphical widgets or tables with continuously updating rows.
2. **Execution environment:** Should the element be designed in a tool like Simulink and executed in an environment like dSpace[44] or xPC Target[45] using auto code generation and execution? Should it be designed in Simulink and auto generated code from the model be hand massaged and executed on a generic computer? Or should the element be hand coded using standard C/C++/Java libraries and executed on a generic computer+operating system? Should the element be a hard real-time task executing in a Xenomai thread on Linux? There are numerous such choices here, each with its pros and cons. We suggest that the wireless and world model elements should run on general purposed computers

(with either stock or realtime operating system kernels) as these will most likely require hand coding. The control element will probably be designed using a tool like Simulink or Labview and it is most conveniently executed on vendor supported hardware that can directly execute the models in realtime i.e. "Push a button and watch the model execute". When assigning elements to physical computers, the concept of "mixed criticality" should also be considered. Is a safety critical function running on the same computer as a less critical function? Are the executions partitioned sufficiently so that they will not interfere with each other? Typically the HMI element is not as critical as the control or world model and it might make sense to isolate it completely on another computer, together with the data logging functions (which are not realtime or safety critical anyway).

SIDE NOTE: While it is theoretically possible to execute the control element in a hard realtime task on a general purpose RTOS, practical considerations often make it difficult. For example, the control element generally requires to talk to the vehicle network over CAN. However, most CAN device drivers for general purpose OSes are not designed with realtime performance in mind. On a dual kernel RTOS system like Linux+Xenomai, the moment the CAN drivers make a syscall, the thread will be dropped from realtime. This necessitates separation of the i/o and the control calculations into different threads, with the requirement of realtime data transfer between them. Situations like this can get messy rapidly and always require high technical competence. Mostly, it is just vastly simpler to run the control on a system like dSpace.

3. **Vendor solutions:** Some elements of the architecture may come as prepackaged vendor black boxes. In such cases, there is little choice but to use whatever technologies these boxes support. For example, a super sensor like radar, or a local sensor like GPS or the wireless may come packaged from a vendor as a black box that outputs data in a certain format on the CAN bus, or over a serial connection or ethernet. In such a case, the only option is to integrate the box into the architecture using whatever means available.
4. **Testing/calibration needs:** Does an element contain many internal variables that need to be constantly adjusted while tests are being conducted in the actual vehicle? Does element data need to be logged? Will it be required to replay the logged data through the element for offline testing of that element, or of a combination of elements? Such considerations have an impact on the tools and execution environments used to develop the elements. Certain development tools or frameworks offer facilities that enable testing/calibration/logging to be done with ease. This can often tip the balance in favor of the tool/framework. And the chosen tool/framework usually works only on specific computing platforms or execution environments.

4.2.2. Communication mechanisms

The choice of communication mechanisms depends on the bandwith requirements, required quality of service, communication pattern and finally (and very importantly) on the technical implementation of the components. In the context of this discussion, 'communication mechanisms' includes everything from communication between two threads running in the same process to communication between distinct computers connected via some form of networking protocol.

1. **Bandwidth requirements:** Different architectural elements have different data transfer requirements and these must be accounted for when selecting the communication mechanism. In the reference architecture, the world model element requires maximum bandwidth, since it continuously receives and serves data. The data transfer requirements for sensors depend to some extent on the local processing done before the sensor output. For example, in the case of a camera sensor, the bandwidth required for transferring even 320x240 images at 30 frames per second is considerably higher than the bandwidth required for transferring information about extracted features from the image. The HMI is another element that can require potentially high bandwidth. For the control elements, the bandwidth requirements are not as high as that for the world model, but the quality of service requirements are more stringent. Typically, communication via shared memory between two processes on the same computer has the highest bandwidth while a CAN bus will have the lowest bandwidth for the communication scenarios likely to occur in an instantiation of the reference architecture. Streaming data using UDP/TCP packets over ethernet lies somewhere in between.
2. **Quality of Service(QoS):** The temporal and spatial requirements of data distribution are specified by QoS policies. The policies can specify requirements for durability, reliability, deadlines, latency budgets, transport priorities, lifespan, partitioning etc. of the transmitted data. It must be considered whether the selected means of communication permits the tweaking of desired QoS parameters and in case it doesn't, the impact this will have on the correctness of the system behavior must be deterministically known in advance. The selection of QoS parameters depends largely on the algorithms being used. For example, some algorithms will require the newest sensor values known at any instant, and delayed historical values may be of little importance. In such a case, it is not necessary to implement a buffered data connection to the relevant element.
3. **Communication pattern:** Certain communication mechanisms (for example, ZeroMQ[46], DDS[47]) impose a pattern on top of the communication. A pattern can be publish/subscribe, push/pull, request/reply, pipeline, exclusive pair etc. If the selected communication mechanism permits it, we recommend a publish/subscribe pattern for continuous dataflows between data producers and data consumers. This is because the publish/subscribe pattern enables good decoupling of the communication partners (the publisher need not know which or how many subscribers

are present and vice-versa) while satisfying functional requirements. For asynchronous data requests(typically involving the supervisor), we recommend the request/reply pattern. The pipeline pattern is not needed in this architecture.

4. **Component implementation:** When prototyping, it is not always possible to have a component implementation that supports all kinds of communication mechanisms. For example, an autogenerated component from Simulink executing on a rapid prototyping setup like xPC target, implies that something like DDS together with all its Qos goodies, can not be used. (This is because the xPC setup supports an extremely limited set of communication possibilities: serial, CAN, basic udp packets). Similarly, usage of specific programming languages, libraries or component frameworks puts rather strong limits on what communication mechanisms can be used. As another example, the AUTOSAR platform does not have built-in support for the publish/subscribe pattern. The point being made here is that any practical implementation narrows down the set of usable communication mechanisms.

5. Instantiated example

In this section, we present an instantiation i.e. a concrete realization of the reference architecture for a cooperative adaptive cruise control (CACC) system. This CACC system can control the longitudinal motion of a vehicle and is activated in a platooning scenario. A platooning scenario is one possible scenario in cooperative driving, in which vehicles drive one behind the other. The first vehicle is denoted as the 'lead vehicle' and it is generally driven manually. The rest of the vehicles follow the lead vehicle autonomously. The lead vehicle and the following vehicles together form a 'platoon'. It is useful in a rough way to think of a platoon of vehicles as a 'road train'. A platoon on a road can split into multiple platoons, merge with other platoons etc. The platoon must obey the local road laws, respect speed limits, react appropriately to traffic lights and so on.

The instantiated architecture was developed and tested on a heavy duty commercial truck, within the Scoop project[48]. The Scoop project was formed in order to create a participating entry in the Grand Cooperative Driving Challenge(GCDC) 2011 [3]. The GCDC 2011 involved several competing teams in urban and highway platooning scenarios. It was held at Helmond, the Netherlands in May 2011. The Scoop team performed well in the challenge, thereby inspiring confidence both in the reference architecture as well as this particular instantiation. Interestingly, the addition of the cooperative driving system required no significant changes to the existing vehicle systems.

The instantiated architecture is described in the following subsections using a hardware view and the so-called module architecture view [42] since these views correspond well to the implementation level detail we wish to describe. Detailed views such as the execution view [42], which shows runtime properties

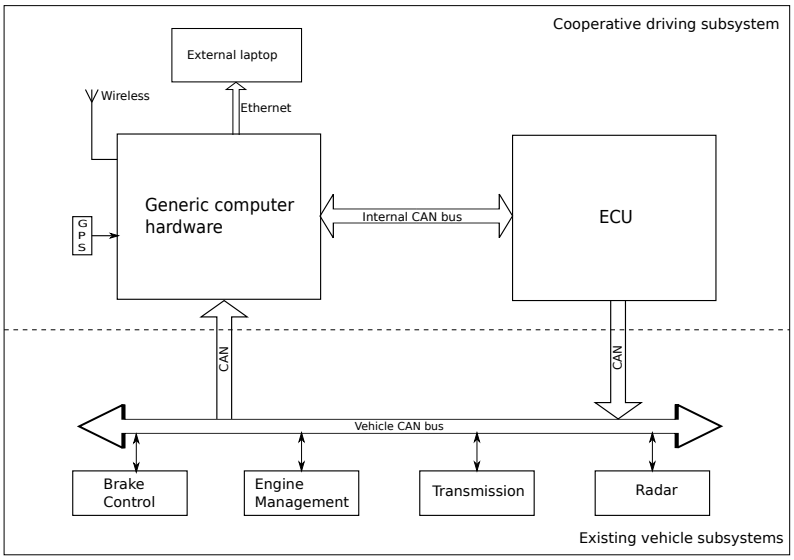

Figure 3: Hardware view of instantiated architecture

and connections, are excluded since they are beyond the scope of this paper. They can be found in the project's technical report[49].

5.1. Hardware view

This description uses the hardware view primarily to establish the context of the instantiated architecture. It shows how the developed system fits in within the rest of the vehicle architecture. The hardware view is shown in Figure 3.

The system consists of a generic computer and an automotive grade Electronic Control Unit (ECU) interconnected via a 250kbps CAN bus. A wireless mini-pci card is installed on the generic computer and functions as the wireless interface. A realtime kinematic (RTK) gps is also connected to the generic computer, acting as a local sensor to the system. There is a CAN bus link between the generic computer and the vehicle network, via which messages on the vehicle CAN bus can be read by the generic computer. This link forms one part of the vehicle gateway. There is another CAN link between the ECU and the vehicle CAN bus, via which the controller in the ECU can send actuation messages to the various vehicle subsystems. This link forms the other part of the vehicle gateway.

The vehicle subsystems in use are primarily the brake, engine and transmission controllers. The existing vehicle radar sensor messages are also read from the CAN bus, and the radar sensor acts like a super sensor in the sense that it directly sends information about detected vehicles ahead, rather than raw radar readings.

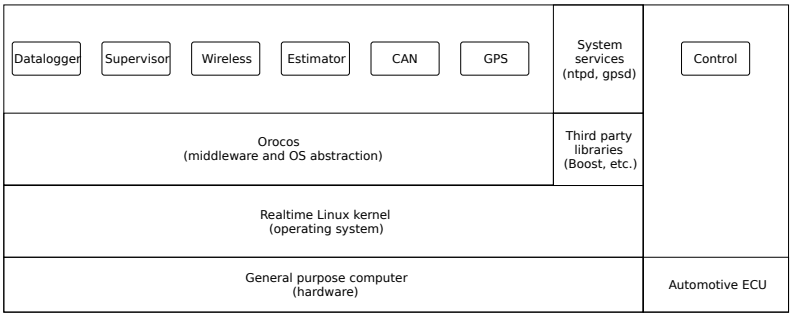

Figure 4: Module view of instantiated architecture

It is possible to optionally connect an external computer to the generic computer via a 10/100mbps ethernet link. The external computer receives and stores log messages and can function like an HMI.

The entire cooperative driving subsystem is thus implemented as two additional nodes on the vehicle CAN bus. The links from the subsystem to the vehicle CAN bus can be physically severed via an emergency switch (not shown in Figure 3). Thus, it is possible to easily connect and disconnect the subsystem from the vehicle.

5.2. Module architecture view

The module architecture view (as described in [42]) encompasses two orthogonal structures: functional decomposition and layers. Functional decomposition of a system captures the way the system is logically decomposed into subsystems, modules and abstract program units. Layers reflect design decisions based on interfacing constraints. They reduce and isolate dependencies and facilitate bottom-up building and testing of the various subsystems. The module view is shown in Figure 4. Note that although [42] describe a module view as an abstract, programming language independent implementation structure, Figure 4 shows the concrete choices we made for implementing the various layers. Therefore, it is no longer abstract and programming language independent, but reflects the actual technologies used by our instantiation.

In Figure 4, the little "boxes" in the topmost layer ("Datalogger", "Supervisor" etc.) represent software components that implement elements of the reference architecture. The Figure also shows that from the viewpoints of execution environments and hardware, the instantiated architecture is split into a two parts. The first part is an automotive grade ECU and the second part is a generic computer running a Xenomai [50] based realtime Linux kernel as the operating system.

The ECU can directly execute Simulink models using auto-generated code. Thus, there is no consideration of operating system, manual task scheduling and priorities, memory management etc. in the ECU part. This is so because the ECU provides a light-weight real-time executive that transparently provides

Orocos component	Corresponding reference architecture element(s)
Supervisor	Supervisor
Estimator	World model
GPS	Local sensor
CAN	Vehicle gateway
Wireless	Wireless, Semantic Data Analyzer
Datalogger	HMI

Table 1: Orocos components and corresponding elements of the reference architecture

mechanisms for dealing with this. The control part of the conceptual architecture executes within the ECU. The controller exchanges control and status data with the generic computer over an internal CAN bus and outputs actuation messages to the various vehicle subsystems over another CAN link, as shown in Figure 3.

The generic computer executes an Intel x86 port of the Linux kernel, patched with the Xenomai realtime framework. On top of the operating system, is the Orocos [51] middleware layer. Orocos contains a realtime toolkit which provides operating system abstraction services as well as facilities to create and execute realtime software components in a time and event triggered fashion. It provides a variety of inter-component communication methods for data and control flow. Components can be distributed across multiple computers and Orocos provides all the middleware and glue for constructing a system out of a set of interacting components. In addition to Orocos, there are two important system services which provide gps data and clock synchronization. The gps data service uses the UNIX gpsd [52] daemon for complete abstraction and management of the gps hardware. The clock synchronization service is provided by the UNIX ntpd [53] daemon, which can interact with gpsd in order to synchronize the system clock with the gps time in Coordinated Universal Time (UTC) format.

The remaining system functionality is implemented using multiple periodic Orocos components. These are the Supervisor, Wireless, Estimator, GPS and CAN. There is also a datalogging component. The CAN component serves a dual purpose. It is the vehicle gateway, as well as the means of communication with the control component, which executes in the ECU. The snapshot of the logical connections of these components are shown in Figure 5. The takeaway for the reader from Figure 5 is that the Wireless, GPS and CAN components act as model data suppliers to the Estimator, which acts as the World Model. Each arrow in the figure indicates the flow of specific data structures. For example, the 'wtm' arrow from the Wireless to the Estimator, is the 'Wireless Trafficlight Message' which sends information about any detected traffic lights. It is neither possible nor necessary for this discussion to explain each connection or dataflow in the figure. All those details are extremely implementation specific and can be found in the project's technical report[49].

The components are summarized in Table 1, together with the elements of the conceptual architecture which they implement. Note that the components

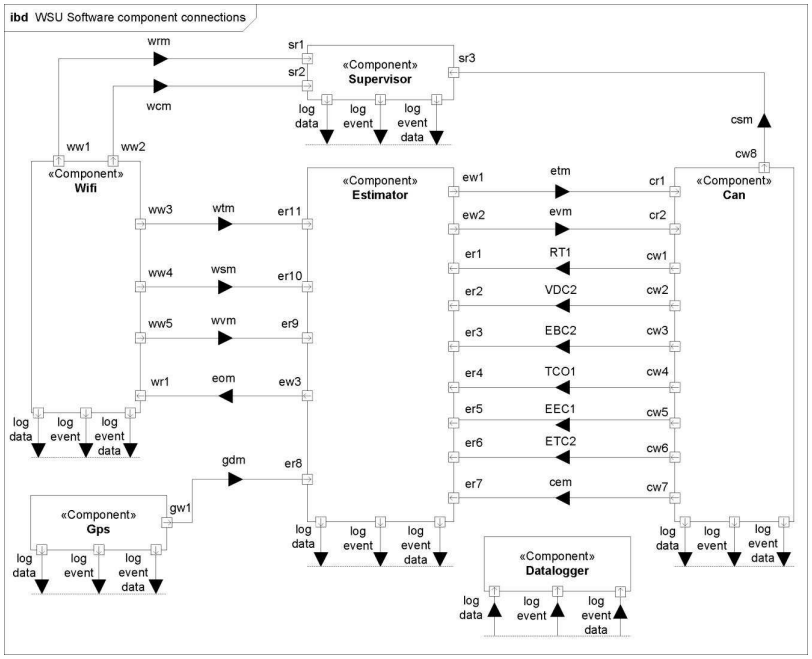

Figure 5: Logical view of instantiated architecture within the general purpose computer. See accompanying text for description.

do not have a one-to-one mapping with the conceptual architecture. In this particular instantiation, the HMI was created by post-processing specific info from the datalogger. While the datalogger indirectly serves as an HMI component it is more an artifact of the specific implementation. It is good to have from a system development and testing viewpoint, but datalogging as a functionality is not needed in the cooperative driving system and therefore is not present in the conceptual architecture. That said, a subset of datalogging facilities may be retained in the architecture for dealing with operational situations like diagnostics and recovery.

The control element of the conceptual architecture is implemented by the control models executing in the ECU.

5.3. Experience with the instantiation

In the instantiated system, the main feeling about the architecture was that it 'just worked'. Once it was developed and debugged, the architecture related code kept functioning quietly in the background. It was possible to shift focus to the development and testing of the actual algorithms, without paying much attention to the architecture. We believe that it reflects positively on the architecture when it does not keep popping up during the project's prototyping phase, by limiting what can be done, or by imposing weird constraints on how certain functionality can be implemented.

During the Grand Cooperative Driving Challenge (GCDC), quite a few changes to the system were required to be made 'on the fly' to accommodate changing competition rules and requirements during the preliminary heats. For example, during tests with the other teams, a system mode had to be introduced where all wireless transmission by the ego vehicle should be stopped, turning it into a 'silent observer'. At other times, the Control element had to be activated/deactivated depending on specific data packets broadcast by the organizers over the wireless. In all cases, it was easy to adapt the system behavior to meet the new requirements. The adaptation was mostly done via the Supervisor, since it had data and control oriented connections to all other architectural elements and could independently trigger mode changes within them.

The load on the vehicle CAN bus underwent no practically significant increase. This is because the Control component's output consisted of only two periodic CAN messages which contained information intended for the vehicle's existing cruise control and braking systems. There was no change to the real time properties of any existing vehicle systems.

There was a strong temptation among the developers to add any newly required functionality within the Estimator component. This was because the Estimator already contained all relevant system data and it was considered easy to add in a new function there, which could fetch needed data using simple and quick local access methods. It was considered to be comparatively more work to add the new functions in some other component and setup the necessary intercomponent data flow mechanisms to fetch/push data from/to the Estimator. There had to be a degree of insistence from the system architects to prevent this from happening. Similarly, there would also be a temptation to bypass the

Supervisor and directly open a control/data connection from one component to another. The architects had to actively squash any 'shortcut' ideas because permitting them implied that components may require knowledge of each other's internal states or ways of functioning, which violates the reference architecture principles. An example of 'quick-and-easy but bad' is: In the GCDC, a request by another vehicle to 'join the platoon' is always accepted positively when the ego vehicle is the platoon leader. Therefore, it is really easy to detect such an incoming request in the Wireless component, and send a positive accept response from the Wireless component itself. This is bad, however, because doing so splits the implementation of the platooning logic between the Wireless and the Control components. Future upgrades to the platooning logic need to be done in more than one component, and besides, it pollutes the function of the Wireless component. The appropriate thing to do here is to propagate the request to the Control component, where all platooning logic decisions are made.

The reference architecture enables the Supervisor to monitor the health of other components, but in practice, it was often difficult to find algorithms to robustly determine a component's state of health. Although this is more of an algorithm issue, we state it here because it has a direct impact on the Supervisor's ability to perform its intended function and can thus undermine the Supervisor's effective role within the architecture.

Other concerns raised during development of the instantiation were more related to the instantiation technologies and do not directly reflect on the reference architecture. For example, the CAN communication protocol between the general purpose computer and the ECU had to be very carefully and painstakingly designed, simply because the size of the CAN message frames (8 bytes, in this instantiation) was a bottleneck. However, those concerns are not within the scope of this paper.

6. Discussion

6.1. Reflection on system characteristics

Section 2.1 presented some characteristics of cooperative driving systems. This subsection shows how the reference architecture relates to those characteristics.

1. The entire architecture is developed within the context of a 'node on the vehicle network'. Thus, it is minimally invasive in the sense that it does not dictate changes to the existing vehicle architecture.
2. The abstraction of a world model and model data processors together provide a way to deal with limited trust in the inputs of individual sensors. Data processors capable of fusing, filtering and estimating sensor inputs provide the world model with resilience to temporary bad inputs from single sensors. They assure the accuracy of world model and ego vehicle data between desired bounds.
3. The abstraction of world model data suppliers acts as a wrapper for sensor data going into the world model. This abstraction contains and localizes

any system modifications that need to be made as a result of changes to the sensor system.

4. The safety paradox can be resolved to some extent by introducing model data processors that conduct plausibility checks on the received data. Plausibility checks can include a wide variety of checks based on physical models, as well as arbitrary rules. For example: It is implausible that a vehicle that has consistently been a few meters ahead will suddenly report a position a few kilometers away. Also, diagnostic and error handling services ameliorate the situation further.
5. From an autonomous systems perspective, the concept of a system Ego or Self maps directly to the functionality of the supervisor element.
6. The cooperative driving system uses the vehicle gateway to provide setpoints for other controllers, for example, the engine and brake ECUs.
7. World model data processors responsible for data fusion and filtering can generate measures of confidence for the model data. These measures of confidence can be used for controlling the accuracy of broadcasted ego vehicle data.
8. The HMI can be easily implemented and modified via the world model data consumer abstraction. Changes to the HMI essentially have no impact on the rest of the system.
9. The architectural elements are split in a way that correlates well to the split in domain expertise. For example, the entire control domain expertise is isolated within the control element, and the control experts can operate within the confines of this element without much consideration for other system elements. Similarly, specialized signal processing and data fusion experts can operate within the confines of individual world model data processors.
10. The supervisor component encodes knowledge of the various architectural elements and how they are supposed to work together in order to achieve the system goals. Thus, it can be used for monitoring and diagnosis of the elements and performing system degradation as applicable.
11. Since the reference architecture places no restrictions on the either the choice or the number of execution environments, safety critical and non-critical functions can be implemented on different computer systems. This enables decomposing the system into different Automotive Safety Integrity Levels (ASIL), permitting easier certification. While this is true of execution environments, the services and architectural elements could still refer to multiple functionalities with different ASIL levels. So, finer decomposition would be necessary within these services/elements and this may or may not always be straightforward. The reference architecture thus partly addresses the issue, but further work is required.
12. The time criticality of tasks can similarly be managed by varying the execution environment. Hard real time operating systems could be used for time-critical tasks. In general, timing requirements are closely related to safety requirements.

13. System safety needs to be assured via a combination of algorithms and architecture. From the architectural side, the effort can be concentrated on preventing and detecting software structural failures, as well as providing hot backups of critical architectural elements. Additional 'observer' components can be introduced that monitor values of critical variables in the world model (which itself can be a implemented via a redundant, distributed database). However, we stress that although this architecture shows promise for tackling safety related concerns, a thorough safety analysis has not yet been conducted.
14. Datalogging is very much dependent on the implementation technologies, and as such is out of scope for the reference architecture. That said, it must be mentioned that most implementation technologies that have the depth and breadth needed to instantiate the reference architecture include datalogging facilities.
15. The separation of data, computation and control occurs naturally via elements like the world model, control and model data processors.
16. Parametrization and calibration are heavily implementation specific. They are beyond the scope of the reference architecture and need to be dealt with at the instantiation level.
17. By maintaining the reference architecture at a conceptual level, no implementation specific constraints are enforced on the instantiations. Therefore, there is complete freedom to choose the execution environments and internal communication mechanisms.
18. High and low level functions are segregated into distinct architectural elements (model data suppliers, the world model, control) enabling their execution on an environment most suitable to the nature of the function.
19. Static and dynamic functions in the system are naturally separated by the reference architecture. World model data suppliers and consumers are relatively static, while the world model and control elements are relatively dynamic.

It is therefore valid to claim that the reference architecture gives due regard to the system characteristics.

6.2. Scenario testing

A litmus test for an architecture is how well it performs in the fifth view of the 4+1 set of architectural views described by [43] . The fifth view is that of *scenarios*. The scenarios are instances of more general *use cases* and in a sense are abstractions of the most important requirements. The fifth view demonstrates how architectural elements work together seamlessly to satisfy the scenarios. The demonstration could be expressed using object scenario diagrams and object interaction diagrams [54]. In many cases though, it is sufficient to sit with the architectural team and discuss how the architecture elements should interact to satisfy a given scenario. If the scenario breaks the architecture by requiring ugly fixes a.k.a hacks, then there is a shortcoming in the architecture.

On the other hand, if the architecture can smoothly deliver the scenario, then the architecture passes that scenario's test.

An exhaustive document compiling scenarios and use cases in cooperative driving is not yet available. There exist, however, use cases that can be extracted from various cooperative driving projects [30, 28, 32, 22, 33]. This reference architecture and its instantiation within the Scoop project was tested for scenarios generated by the GCDC 2011. The architecture handled all the scenarios elegantly. In fact, in the days leading up to the GCDC 2011, the scenarios and requirements changed frequently. These changes were smoothly absorbed by the architecture as evidenced by the fact that the development team could easily find ways in which the architectural elements could work together to meet the scenario requirements.

Some example scenarios and informal descriptions of how they are satisfied by the architecture are given in the following subsection. The descriptions are not exhaustive and are provided merely to give an idea of scenario testing.

6.2.1. Example scenarios

This subsection explains what happens during each of the following scenarios.
Scenario: Controlling vehicle motion while in a platoon.

1. Information regarding other vehicles comes in over the wireless. The semantic data analyzer forwards it to the world model.
2. Additional information regarding the vehicle ahead comes from the vehicle radar sensor. This information too goes to the world model.
3. World model data processors fuse and filter the data to maintain a consistent world representation.
4. The control uses the world model data to control ego vehicle motion. It also passes back relevant data about the ego vehicle control to the world model. (Note that the data exchange is application specific, and not part of the reference architecture.)
5. Supervisor non-obstructively monitors various elements.

Scenario: Vehicle approaches platoon from behind and wishes to join it. (Note: The applicable interaction protocol states that a join request is made by modifying a status parameter in the ego vehicle's periodic information broadcast.)

1. Information about vehicles in the platoon ahead is received over the wireless. The semantic data analyzer forwards it to the world model.
2. The world model updates, to reflect the presence of the vehicles ahead.
3. The platooning logic in the control realizes that ego vehicle can join the platoon. It modifies ego vehicle's relevant status parameter in the world model.
4. The status parameter (and thus, a request to join the platoon) is broadcasted via the wireless as part of the standard periodic ego vehicle information broadcast.
5. If a response (ok/not ok to join) is received over the wireless, the semantic data analyzer sends it to the supervisor, who in turn notifies the control.

6. If no response is received, the control times out after a predefined time has elapsed and resets the ego vehicle status information in the world model

Scenario: GPS signal is lost.

1. The model data supplier that supplies GPS signal to the world model sends 'No signal'.
2. The model data processor responsible for estimating ego vehicle position starts lowering its confidence estimates in the continued absence of the GPS data.
3. When the confidence of the position estimates falls below a pre-specified level, the control stops taking charge of vehicle motion. Appropriate messages are sent to the supervisor and the world model is updated.
4. The update to the world model is picked up by the HMI and the driver is notified that the control is no longer in charge.

Scenario: There is an external demand for imposition of radio silence.

1. The demand is received over the wireless.
2. The semantic data analyzer forwards it to the supervisor.
3. The supervisor sets appropriate modes in the wireless broadcast and control.
4. The control updates the world model with the current control mode
5. The HMI picks up the update in the world model and notifies the driver if necessary.

6.2.2. Handling unexpected scenarios

It is a fact that scenarios will turn up, which can not be immediately handled by a particular instantiation. In such cases, the primary test of the reference architecture is the ease with which temporary solutions can be implemented. A secondary test is the ease with which the temporary solutions can later be migrated to the various architectural elements. While no standard metrics exist for measuring the ease of these practices, our reference architecture aids the former by suggesting and enabling a straightforward approach: All unanticipated functionality goes in the supervisor. This approach works both from the conceptual as well as the practical viewpoints. Conceptually, the supervisor is ultimately responsible for all the system functionality and must make up for the shortcomings of other architectural elements. Practically, the supervisor has data and control communications with all elements which makes it easy from a development perspective to add functionality there.

6.3. AUTOSAR and the reference architecture

Any discussion of automotive architecture today would be incomplete without the mention of AUTOSAR [55]. The AUTOSAR partnership is an alliance of OEM manufacturers and Tier 1 automotive suppliers working together to develop and establish a de facto open industry standard for automotive E/E

architecture which will serve as a basic information infrastructure for the management of functions within both future applications and standard software modules. The AUTOSAR standard comprises a set of specifications describing software architecture components and defining their interfaces as well as the definition of a standardized development methodology [56].

The AUTOSAR Run Time Environment (RTE), Basic Software and Virtual Function Bus (VFB) provide the basis for specifying execution environments and platform services needed by the reference architecture. Furthermore the VFB and AUTOSAR's standardized interface definitions can be used for specifying needed communication facilities between the reference architecture elements as well as with the platform and vehicle interfaces. The concept of the AUTOSAR software component can be directly used to realize the functionality of individual reference architecture elements. AUTOSAR's special sensor/actuator software components can be used to encapsulate the functionality of the model data suppliers and provide abstract actuator interfaces. Elements like the world model, if accepted widely enough, could form a part of AUTOSAR's infrastructure services.

The concepts of the reference architecture fit well within the AUTOSAR paradigm. Therefore, AUTOSAR tools and methods can potentially be used when mapping the reference architecture to a specific instantiation. The reference architecture can be rapidly instantiated using industry-accepted toolchains and development processes. This in turn reduces the time to market, which is a competitive factor for automotive manufacturers.

6.4. Comparison with autonomous system architecture

A cooperative driving system can be considered to be an autonomous system, subject to operational constraints. Therefore, it is logical to compare our reference architecture with architectures specifically targeted for autonomous systems.

One of the successful architectures for autonomy is Realtime Control System (RCS) [57, 58, 59, 60]. RCS and derivative architectures are based on a general theory of intelligence described in [61] . A comparison with these architectures shows that some of their principles are shared by our reference architecture. Specifically, RCS version 4 considers Sensory Processing (SP), World Model (WM), Value Judgment (VJ) and Behavior Generation (BJ) as the building blocks of intelligence. In our reference architecture, there is a one to one correspondence between the world models. The SP is equivalent to the model data supplier elements, while VJ and BG are abstracted by the control element. RCS-4 decomposes the architecture into hierarchies based on temporal and spatial resolution of goals and tasks. Our reference architecture does not specify specific hierarchical decomposition stratagems, but supports hierarchies within and between architectural elements via their interface abstractions. Hierarchies in a cooperative driving system are affected by the context of the system i.e. by the fact that it is essentially just one box among many others on the vehicle network, without direct control over the contents of the other boxes.

Blackboard architectures [62] are also common in autonomous systems. In a blackboard architecture, a set of problem solving modules share a common global database. Our architecture essentially uses the world model as a blackboard representation of the perceived world. The model data processors behave like the problem solvers. Depending on the particular instantiation, the model data processors (problem solvers) can make concerted efforts to solve specific problems, or they may be redundant solvers, checking each others solutions. The world model acts as the control shell, coordinating and managing access to the blackboard by the various model data processors. By restricting the blackboard to a single architectural element with well defined interfaces, the ease of implementing the blackboard is improved.

These brief comparisons show that our reference architecture does not make radical departures from established principles of autonomous system architectures. In the following section we elaborate on novel aspects of our reference architecture and point to directions for future work.

6.5. Role during instantiation

The reference architecture was instantiated as a cooperative adaptive cruise controller for the Grand Cooperative Driving Challenge, 2011, as described in section 5. In this section, we reflect on the role played by the reference architecture during the instantiation.

When attempting to create a working implementation of any system, starting with a completely blank slate is a time consuming and error prone process. It is helpful to have a carefully considered set of guidelines and solution ideas to kickstart the thinking. The reference architecture endeavors to provide a generic template for implementing a cooperative driving solution. It becomes a starting point for the design, which can be refined and specifically customized for the system being implemented.

The reference architecture next helps to communicate the key ideas and concepts of the architecture to the development team. It provides a common terminology within the project which helps to eliminate confusion caused by subjective interpretation of certain words. For example, we have experienced that the word 'Supervisor' has different connotations and meanings to different people. By specifying the role of the Supervisor in this architecture and providing examples of what the Supervisor does, the reference architecture facilitates a common understanding.

The reference architecture plays an important role in eliciting technical requirements for the instantiation. This is because it describes the design at a rather abstract level and then includes guidelines and 'points to ponder' for an instantiation. A lot of the technical requirements for the instantiation emerge when thinking about these guidelines. It also helps to identify key issues and performance constraints that the particular instantiation needs to address.

The reference architecture triggers proactive planning of the system testing and verification processes. Advance knowledge of the main system elements enables the development of test harnesses and protocols simultaneously with the development of the components themselves.

Finally, the reference architecture provides a 'big picture' throughout the instantiation phase. This helps to clarify the functionality which should be provided by each component. Without the reference architecture, it is all too easy to succumb to programming convenience/laziness and quickly add in minor functions where they should not be located. Doing this inevitably leads to polluting the clean design and can lead to complex interdependencies among the components.

6.6. Highlights and future work

This work took the concept of a reference architecture and applied it to a cooperative driving system. The resulting reference architecture was instantiated, tested and validated on a commercial truck, in a competitive real world usage scenario. This work thus provides an engineering basis for development of cooperative driving systems.

From the view point of industrial exploitation, it is important that the reference architecture be realizable using Commercial Off The Shelf (COTS) components, established toolchains and development processes. The reference architecture corresponds well with the industry standard AUTOSAR systems, and thus there are no conflicts in fulfilling this requirement. The correspondence also implies a shorter time to market, which is a competitive advantage. The architecture is minimally invasive as it requires no significant modification to existing vehicle functionality. Since it is restricted to a "node on a bus", it is easy to provide the cooperative driving feature only on specific vehicle variants, or as an upgrade to an existing vehicle. That said, assumptions have been made regarding the availability of certain services from the rest of the vehicle. Such services typically involve the ability to provide actuation messages to other subsystems like the engine, transmission and the brakes. It is possible that these existing services in the vehicle may need to be slightly modified or recalibrated in order to better serve the needs of the cooperative driving functions. For example, when instantiating the architecture on a commercial truck, the minimum speed for permitting activation of cruise control had to be recalibrated from 10km/h down to 0 km/h. In all such cases, the modifications and their impact needs to be thoroughly understood.

The ability to separate timing and safety critical functions from non-critical ones facilitates the support for functional safety according to the upcoming ISO 26262 standard. The modular concept provides good separation of concerns and domain specific knowledge is confined within the boundaries of individual architectural elements. The modularity also helps to contain errors and prevents faults from propagating beyond the element they occurred in. It must be mentioned though, that a comprehensive analysis of the architecture with regards to safety issues has not yet been conducted. This, together with ASIL decomposition of the architecture is a significant future work.

Services needed for cooperative driving are listed, together with the architectural elements for enabling them. This is done without constraining the implementation technologies. Keeping high levels of abstraction provides the necessary flexibility needed by specific instantiations. All instantiations though,

retain the proven principles of the reference architecture. The usage of plugins within architectural elements enables a two pronged advantage: New algorithms for specific architectural elements can be easily tested. At the same time, the architecture becomes a natural basis for evaluating and comparing different algorithms.

Critical parts of the architecture are completely isolated from each other. The world model has no dependencies on the control, which in turn is unrelated to elements like the HMI or wireless. There is predefined distribution of work and roles among the elements, yet the supervisor element can override the structure wherever and whenever the need arises. These overrides can be realized via the properties common to all elements, like diagnostics, (a)synchronous command execution, rerouting of dataflow ports etc.

It might be considered that the reference architecture is not sufficiently refined for a practical implementation, but doing so is missing the entire point of a reference architecture. Details related to an implementation are missing precisely because they are implementation specific. That said, there are certain undeveloped aspects. Foremost among these is that the onus of assuring non-functional system properties like safety, is laid squarely on the architect instantiating the reference architecture. In theory, a reference architecture should provide some guidelines and mechanisms for assuring such properties. System safety is a big area of research and some of the ideas developed in that area could be transferred to reference architectures in the form of system structures and behavior rules.

The reference architecture could also be extended with mechanisms for supporting timing analysis and other analytical formal methods. The world model and control elements could be refined by presenting various alternative representations and hierarchies. For example, some cooperative driving scenarios may require functionalities like look-ahead, shorter as well as longer term planning, etc. The control element could be hierarchically refined towards such ends. This would be pushing into the area of domain specific algorithms and structures, but sometimes there is a blurred line between generic architecture and domain specific knowledge.

Finally, no matter how good a reference architecture is, it can always be ruined by a bad implementation. Therefore, a relevant question to ask is, "How should one go about instantiating a reference architecture? What considerations and reasoning should be applied?" The reference architecture provides some guidelines, but it would be nice to include a description of the instantiation process, together with templates, checklists and mechanisms to avoid common traps and pitfalls.

References

[1] P. Kruchten, The Rational Unified Process, Rational Software White Paper, Addison-Wesley, 2003.

[2] U. Eklund, O. Askerdal, J. Granholm, A. Alminger, J. Axelsson, Experience of introducing reference architectures in the development of automotive electronic systems, in: Proceedings of the second international workshop on Software engineering for automotive systems - SEAS '05, ACM Press, New York, New York, USA, 2005, pp. 1–6.

[3] Grand Cooperative Driving Challenge. http://www.gcdc.net, 2011.

[4] P. Zwaneveld, B. van Arem, Traffic effects of Automated Vehicle Guidance Systems: A Literature Survey, 1997.

[5] B. van Arem, C. J. G. van Driel, R. Visser, The Impact of Cooperative Adaptive Cruise Control on Traffic-Flow Characteristics, IEEE Transactions on Intelligent Transportation Systems 7 (2006) 429–436.

[6] M. Sichitiu, M. Kihl, Inter-vehicle communication systems: a survey, IEEE Communications Surveys & Tutorials 10 (2008) 88–105.

[7] S. Tsugawa, Inter-vehicle communications and their applications to intelligent vehicles: an overview, in: Intelligent Vehicle Symposium, 2002. IEEE, volume 2, IEEE, 2002, pp. 564–569.

[8] T. Willke, P. Tientrakool, N. Maxemchuk, A survey of inter-vehicle communication protocols and their applications, IEEE Communications Surveys & Tutorials 11 (2009) 3–20.

[9] R. Nagel, S. Eichler, J. Eberspacher, Intelligent wireless communication for future autonomous and cognitive automobiles, in: Intelligent Vehicles Symposium, 2007 IEEE, IEEE, 2007, pp. 716–721.

[10] A. Khodayari, A. Ghaffari, S. Ameli, J. Flahatgar, A historical review on lateral and longitudinal control of autonomous vehicle motions, in: 2010 International Conference on Mechanical and Electrical Technology, Icmet, IEEE, 2010, pp. 421–429.

[11] A. Girard, J. de Sousa, J. Hedrick, An overview of emerging results in networked multi-vehicle systems, in: Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228), volume 2, IEEE, 2001, pp. 1485–1490.

[12] S. Kato, S. Tsugawa, K. Tokuda, Vehicle control algorithms for cooperative driving with automated vehicles and intervehicle communications, , IEEE Transactions on 3 (2002) 155–161.

[13] O. Gehring, H. Fritz, Practical results of a longitudinal control concept for truck platooning with vehicle to vehicle communication, in: Proceedings of Conference on Intelligent Transportation Systems, IEEE, 1998, pp. 117–122.

[14] G. Naus, R. Vugts, J. Ploeg, R. van de Molengraft, M. Steinbuch, Cooperative adaptive cruise control, design and experiments, in: American Control Conference (ACC), 2010, volume 1, IEEE, 2010, pp. 6145–6150.

[15] R. Rajamani, Semi-autonomous adaptive cruise control systems, in: Proceedings of the 1999 American Control Conference (Cat. No. 99CH36251), volume 2, IEEE, 1999, pp. 1491–1495.

[16] J. Hedrick, M. Tomizuka, P. Varaiya, Control issues in automated highway systems, IEEE Control Systems Magazine 14 (1994) 21–32.

[17] P. Ioannou, C. Chien, Autonomous intelligent cruise control, IEEE Transactions on Vehicular Technology 42 (1993) 657–672.

[18] P. Varaiya, Smart cars on smart roads: problems of control, IEEE Transactions on Automatic Control 38 (1993) 195–207.

[19] R. Horowitz, Control design of an automated highway system, Proceedings of the IEEE 88 (2000) 913–925.

[20] G. Karagiannis, O. Altintas, E. Ekici, G. Heijenk, B. Jarupan, K. Lin, T. Weil, Vehicular Networking: A Survey and Tutorial on Requirements, Architectures, Challenges, Standards and Solutions, IEEE Communications Surveys & Tutorials 13 (2011) 584–616.

[21] C. Calefato, D. Cardillo, F. Tango, Vehicle towards vehicle: Current status and beyond on research about adaptive and cooperative vehicles and their smart behaviors, in: 2009 2nd Conference on Human System Interactions, IEEE, 2009, pp. 588–595.

[22] The SAFESPOT Project. http://www.safespot-eu.org/, 2010.

[23] R. Bishop, A survey of intelligent vehicle applications worldwide, in: Proceedings of the IEEE Intelligent Vehicles Symposium 2000 (Cat. No.00TH8511), id, IEEE, 2000, pp. 25–30.

[24] C. Bergenhem, Q. Huang, A. Benmimoun, Challenges Of Platooning On Public Motorways, in: 17th World Congress on Intelligent Transport Systems, pp. 1–12.

[25] W. Niehsen, IVS 05: New Developments and Research Trends for Intelligent Vehicles, IEEE Intelligent Systems 20 (2005) 10–14.

[26] S. Tsugawa, S. Kato, T. Matsui, H. Naganawa, H. Fujii, An architecture for cooperative driving of automated vehicles, in: Intelligent Transportation Systems, 2000. Proceedings. 2000 IEEE, Mi, IEEE, 2000, pp. 422–427.

[27] S. Tsugawa, S. Kato, K. Tokuda, A cooperative driving system with automated vehicles and inter-vehicle communications in Demo 2000, Intelligent (2001) 918–923.

[28] The CVIS Project. http://www.cvisproject.org/, 2009.

[29] G. Vivo, P. Dalmasso, F. Vernacchia, The European Integrated Project "SAFESPOT"-How ADAS applications co-operate for the driving safety, in: 2007 IEEE Intelligent Transportation Systems Conference, IEEE, 2007, pp. 624–629.

[30] The COOPERS Project. http://www.coopers-ip.eu, 2010.

[31] G. Toulminet, J. Boussuge, C. Laurgeau, Comparative synthesis of the 3 main European projects dealing with Cooperative Systems (CVIS, SAFESPOT and COOPERS) and description of COOPERS Demonstration Site 4, in: 2008 11th International IEEE Conference on Intelligent Transportation Systems, IEEE, 2008, pp. 809–814.

[32] The HAVEit Project. http://www.haveit-eu.org, 2010.

[33] The SARTRE project. http://www.sartre-project.eu (2011).

[34] M. Broy, Challenges in automotive software engineering, in: Proceedings of the 28th international conference on Software engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 33–42.

[35] V. Schulte-Coerne, A. Thums, J. Quante, Challenges in Reengineering Automotive Software, Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on (2009) 315–316.

[36] A. Pretschner, M. Broy, I. H. Kruger, T. Stauner, Software Engineering for Automotive Systems: A Roadmap, Future of Software Engineering, 2007. FOSE '07 (2007) 55–71.

[37] M. Broy, I. H. Kruger, A. Pretschner, C. Salzmann, Engineering Automotive Software, Proceedings of the IEEE 95 (2007) 356–373.

[38] The MathWorks Inc, Simulink software, 2009.

[39] K. H. Johansson, M. Törngren, L. Nielsen, Vehicle Applications of Controller Area Network, Sensors Peterborough NH VI (2005) 741–765.

[40] Z. Papp, C. Brown, C. Bartels, World modeling for cooperative intelligent vehicles, in: 2008 IEEE Intelligent Vehicles Symposium, IEEE, 2008, pp. 1050–1055.

[41] D. Jiang, IEEE 802.11 p: Towards an international standard for wireless access in vehicular environments, Vehicular Technology Conference, 2008 (2008) 2036–2040.

[42] D. Soni, R. Nord, Software architecture in industrial applications, Software Engineering, 1995. (1995) 196–196.

[43] P. Kruchten, Architectural Blueprints - The "4+1" View Model of Software Architecture, Ieee Software 12 (1995) 42–50.

[44] dSpace http://www.dspace.com, 2012.

[45] xPC Target http://www.mathworks.se/products/xpctarget/, 2012.

[46] ZeroMQ: The Intelligent Transport Layer http://www.zeromq.org/, 2012.

[47] The OMG DDS Portal http://portals.omg.org/dds/, 2012.

[48] J. Må rtensson, A. Alam, S. Behere, The Development of a Cooperative Heavy-Duty Vehicle for the GCDC 2011: Team Scoop, IEEE Transactions on Intelligent Transportation Systems 13 (2012) 1033–1049.

[49] S. Behere, Scoop Technical Report: Year 2011, Technical Report, KTH Royal Institute of Technology, Stockholm, 2011.

[50] Xenomai. http://www.xenomai.org, 2010.

[51] Open Robot Control Software. http://www.orocos.org, 2011.

[52] E. Raymond, gpsd. http://www.catb.org/gpsd/, 2011.

[53] Ntpd. http://en.wikipedia.org/wiki/Ntpd, 2011.

[54] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, K. Houston, Object-oriented analysis and design with applications, third edition, volume 33, Addison-Wesley Professional, 2007.

[55] S. Fürst, B. M. W. Group, J. Mössinger, S. Bunzel, T. Weber, F. Kirschkebiller, F. M. Company, K. Lange, V. Ag, AUTOSAR - A Worldwide Standard is on the Road., VDI Congress (2009) 1–16.

[56] H. Fennel, S. Bunzel, H. Heinecke, J. Bielefeld, S. Fürst, B. M. W. Group, K.-p. Schnelle, W. Grote, N. Maldener, T. Weber, F. Wohlgemuth, J. Ruh, L. Lundh, T. Sandén, F. M. Company, P. Heitkämper, R. Rimkus, G. Motors, J. Leflour, A. Gilberg, P. S. A. P. Citroën, U. Virnich, S. Voget, S. Vdo, Achievements and exploitation of the AUTOSAR development partnership, Convergence 2006 (2006).

[57] J. S. Albus, R. Lumia, J. Fiala, A. J. Wavering, H. G. McCain, NASREM - The NASA/NBS Standard Reference Model for Telerobot Control System Architecture, in: proceedings of the 20th International Symposium on Industrial Robots, NIST 1235, NIST.

[58] J. Albus, A reference model architecture for intelligent systems design, An introduction to intelligent and autonomous control (1993) 27–56.

[59] J. Albus, F. Proctor, A reference model architecture for intelligent hybrid control systems, in: Proceedings of the 1996 Triennial World Congress, International Federation of Automatic Control (IFAC).

[60] A. J. Barbera, J. S. Albus, L. S. Haynes, RCS : The NBS Real -Time Control System (1984).

[61] J. Albus, Outline for a theory of intelligence, IEEE Transactions on Systems, Man, and Cybernetics 21 (1991) 473–509.

[62] B. Hayes-Roth, A blackboard architecture for control, Artificial Intelligence 26 (1985) 251–321.