A Standard Driven Software Architecture for Fully Autonomous Vehicles

Alexandru Constantin Serban[1,2] Erik Poll [1] Joost Visser[1,2]

[1] *Radboud University*
Nijmegen, The Netherlands
E-mail: {a.serban, erikpoll}@cs.ru.nl

[2] *Software Improvement Group*
Amsterdam, The Netherlands
E-mail: {a.serban, j.visser}@sig.eu

Abstract

Several key technologies have enabled the dream of autonomous driving. The development of self driving cars is often regarded as adding a layer of intelligence on top of classic vehicle platforms. However, the amount of software needed to reach autonomy will exceed the software deployed for operation of normal vehicles. As complexity increases, the demand for proper structure also increases. In this paper we introduce a functional software architecture for fully autonomous vehicles aimed to ease the development process. Existing literature presents past experiments with autonomous driving or implementations specific to limited domains (e.g. winning a competition). The architectural solutions are often an after-math of building or evolving an autonomous vehicle and not the result of a clear software development lifecycle. A major issue of this approach is that requirements can not be traced with respect to functional components and several components group most functionality. Therefore, it is often difficult to adopt the proposals. In this paper we take a prescriptive approach starting with requirements from a widely adopted automotive standard. We follow a clear software engineering process, specific to the automotive industry. During the design process, we make extensive use of robotic architectures - which seem to be often ignored by automotive software engineers.

Keywords: Intelligent vehicles, Autonomous vehicles, Robotics, Software architecture

1. Introduction

Autonomous driving is no longer a lab experiment. As manufacturers compete to raise the level of vehicle automation, cars become highly complex systems. Driving task automation is often regarded as adding a layer of cognitive intelligence on top of basic vehicle platforms [1]. While traditional mechanical components become a commodity [2] and planning algorithms become responsible for critical decisions, software emerges as the lead innovation driver. Recent trends forecast an increase in traffic safety and efficiency by minimising human involvement and error . The transfer of total control from humans to machines is classified by the *Society of Automotive Engineers* (SAE) as a stepwise process on a scale from 0 to 5, where 0 involves no automation and 5 means full-time performance by an automated driving system of all driving aspects, under all roadway and environmental conditions [3]. Since the amount of software grows, there is a need to use advanced software engineering methods and tools to

handle its complexity, size and criticality. Software systems are difficult to understand because of their non-linear nature - a one bit error can bring an entire system down, or a much larger error may do nothing. Moreover, many errors come from design flaws or requirements (miss-) specifications [4].

Basic vehicles already run large amounts of software with tight constraints concerning real-time processing, failure rate, maintainability and safety. In order to avoid design flaws or an unbalanced representation of requirements in the final product, the software's evolution towards autonomy must be well managed. Since adding cognitive intelligence to vehicles leads to new software components deployed on existing platforms, a clear mapping between functional goals and software components is needed.

Software architecture was introduced as a means to manage complexity in software systems and help assess functional and non-functional attributes, before the build phase. A good architecture is known to help ensure that a system satisfies key requirements in areas such as functional suitability, performance, reliability or interoperability [5].

The goal of this paper is to design a functional software architecture for fully autonomous vehicles. Existing literature takes a descriptive approach and presents past experiments with autonomous driving or implementations specific to limited domains (e.g. winning a competition). The architectural solutions are therefore an after-math of building or evolving an autonomous vehicle and not the result of a clear software development life-cycle. A major issue of this approach is that requirements can not be traced with respect to functional components and several components group most functionality. Therefore, without inside knowledge, it is often not straight forward to adopt the proposals.

In this paper we take a prescriptive approach driven by standard requirements. We use requirements from the SAE J3016 standard, which defines multiple levels of driving automation and includes functional definitions for each level. The goal of SAE J3016 is to provide a complete taxonomy for driving automation features and the underlying principles used to evolve from none to full driving automation. At the moment of writing this paper, it is the only standard recommended practice for building autonomous vehicles. We provide an extensive discussion on the design decisions in the form of trade-off analysis, which naturally leads to a body of easily accessible distilled knowledge. The current proposal is an extension of our prior work [6].

The term *functional architecture* is used analogous to the term *functional concept* described in the ISO 26262 automotive standard [7, 1]: a specification of intended *functions* and necessary *interactions* in order to achieve desired behaviours. Moreover, it is equivalent to *functional views* in software architecture descriptions; which provide the architects with the possibility to cluster functions and distribute them to the right teams to develop and to reason about them [4]. Functional architecture design corresponds to the second step in the V-model [7, 8], a software development life cycle imposed by the mandatory compliance to ISO 26262 automotive standard.

We follow the methodology described by Wieringa [9] as the design cycle; a subset of the engineering cycle which precedes the solution implementation and implementation evaluation. The design cycle includes designing and validating a solution for given requirements.

The rest of the paper is organised as follows. In Section 2 we introduce background information. In Section 3 we infer the requirements from the SAE J3016 standard. In Section 4 we present the reasoning process that lead to a solution domain. The functional components are introduced in Section 5, followed by component interaction patterns in Section 6 and a general trade-off analysis in Section 7. A discussion follows in Section 8. In Section 9 we compare the proposal with related work and conclude with future research in Section 10.

2. Background

The development of automotive systems is distributed between vehicle manufacturers, called *Original Equipment Manufacturers* (OEM), and various component suppliers - leading to a distributed software development life cycle where the OEM play the role of technology integrators. This development process allows OEM to delegate re-

sponsibility for development, standardisation and certification to their component suppliers. The same distributed paradigm preserves, at the moment, for component distribution and deployment inside a vehicle.

Embedded systems called *Electronic Control Units* (ECU) are deployed on vehicles in order to enforce digital control of functional aspects such as steering or brakes. Many features require interactions and communications across several ECUs. For example, cruise control needs to control both breaking and steering systems based on the presence of other traffic participants. In order to increase component reuse across systems, manufacturers and vendors developed a number of standardised communication buses (e.g. CAN, FlexRay) and software technology platforms (AUTOSAR). This paper is concerned with the design of functional software components (deployed on ECUs) and the way they interact in order to develop autonomous vehicles.

Although the purpose of SAE J3016 is not to provide strict requirements for autonomous vehicles, it is the only standard guide towards autonomy available at the moment. Moreover, it is widely adopted. With the goal of understanding the vehicle automation process, we first introduce the most important terms as defined by SAE J3016 [3]:

- *Dynamic Driving Task* (DDT) - real-time operational and tactical functions required to operate a vehicle, excluding strategic functions such as trip scheduling or route planning. DDT is analogous to driving a car on a given route.
- Driving automation system - hardware and software systems collectively capable of performing part or all of the DDT on a sustained basis. Driving automation systems are usually composed of design-specific functionality called *features* (e.g. automated parking),
- *Operational Design Domains* (ODD) - the specific conditions under which a given driving automation system or feature is designed to function.
- DDT feature - a design-specific functionality at a specific level of driving automation with a particular ODD.

Besides hardware constraints, full vehicle automation involves the automation of the DDT in all ODD, by developing a driving automation system. Recursively, driving automation systems are composed of design-specific features. In this sense, complete vehicle automation is seen as developing, deploying and orchestrating enough DDT features in order to satisfy all conditions (ODDs) in which a human driver can operate a vehicle.

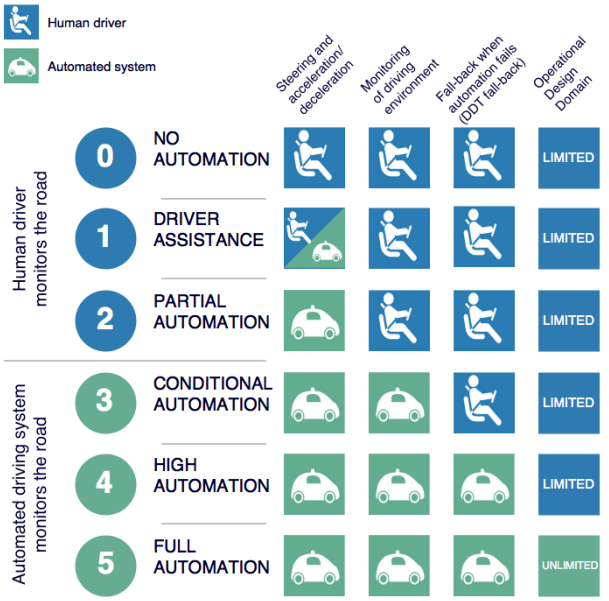

Fig. 1. SAE J3016 levels of driving automation.

The SAE classification of driving automation for on-road vehicles, showcased in Figure 1, is meant to clarify the role of a human driver, if any, during vehicle operation. The first discriminant condition is the environmental monitoring agent. In the case of no automation up to partial automation (levels 0-2), the environment is monitored by a human driver, while for higher degrees of automation (levels 3-5), the vehicle becomes responsible for environmental monitoring.

Another discriminant criteria is the responsibility for DDT fall-back mechanisms. Intelligent driving automation systems (levels 4-5) embed the responsibility for automation fall-back constrained or not by operational domains, while for low levels of automation (levels 0-3) a human driver is fully responsible.

According to SAE:

- If the driving automation system performs the longitudinal and/or lateral vehicle control, while the driver is expected to complete the DDT, the division of roles corresponds to levels 1 and 2.
- If the driving automation system performs the entire DDT, but a DDT *fall-back ready user* is expected to take over when a system failure occurs, then the division of roles corresponds to level 3.
- If a driving automation system can perform the entire DDT and fall-back within a prescribed ODD or in all driver-manageable driving situation (unlimited ODD), then the division of roles corresponds to levels 4 and 5.

3. Requirements Inference

The process of functional architecture design starts by developing a list of functional components and their dependencies [4]. Towards this end, SAE J3016 defines three classes of components:

- Operational - basic vehicle control,
- Tactical - planning and execution for event or object avoidance and expedited route following, and
- Strategic - destination and general route planning.

Each class of components has an incremental role in a hierarchical control structure which starts from low level control, through the operational class and finishes with a high level overview through the strategic class of components. In between, the tactical components handle trajectory planning and response to traffic events. This hierarchical view upon increasing the level of vehicle automation is an important decision driver in architecture design.

Later, the SAE definition for DDT specifies, for each class, the functionality that must be automated in order to reach full autonomy (level 5):

- Lateral vehicle motion control via steering (operational).
- Longitudinal vehicle control via acceleration and deceleration (operational).
- Monitoring of the driving environment via object and event detection, recognition, classification and response preparation (operational and tactical).
- Object and event response execution (operational and tactical).
- Manoeuvre planning (tactical).
- Enhanced conspicuity via lighting, signalling and gesturing, etc. (tactical).

Moreover, an autonomous vehicle must ensure DDT fall-back and must implement strategic functions, not specified in the DDT definition. The latter consists of destination planning between two points provided by a human user.

An overview of the hierarchical class of components defined is illustrated in Figure 2. The level of complexity increases from left to right - from operational to strategic functions.

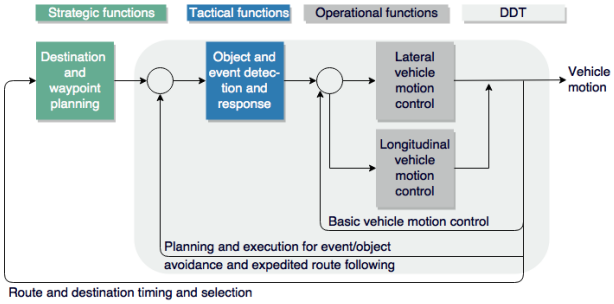

Fig. 2. Functional component classification according to SAE J3016 [3].

Although not exhaustive, the list of components and their intended behaviour, as specified by SAE J3016, represents a good set of initial requirements. It is the choice of each OEM how any of the intended functionality will be implemented. For example, different algorithms or software systems could be used to implement object detection, recognition and classification. This choice can impact the final set of functional components because one OEM can choose to use multiple sensors and powerful sensor fusion algorithms, while other OEM can build a single component that can handle all tasks. In this paper, we strive to divide each function into atomic components. If any of these pieces will be combined in higher level components, it won't have any impact on this proposal. An in-depth analysis of such

trade-offs, often driven by the distributed software development life-cycle, is presented in Section 7.

Lastly, we mention that the automation of any task is a control loop which receives input from sensors, performs some reasoning and acts upon the environment (possibly through actuators) [10]. The automation of complex tasks requires a deeper (semantic) understanding of sensor data in order to generate higher order decisions or plans. However, the loop behaviour is preserved. This analogy holds for the SAE classification of functional components illustrated in Figure 2. Each class of components can be implemented as a control loop which receives input from sensors and acts accordingly. From left to right - from operational to strategic functions - the level of semantic knowledge needed in order to make a decision increases. Moreover, as we move further from left to right, the components do not control actuators anymore, but other components. For example, tactical components will send commands to operational components. Similarly, strategic functions can not directly act upon actuators, but communicate with tactical functions, which will later command operational functions.

Thus we can regard each class of components as a big loop - illustrated in Figure 2 - and each component in each class as a smaller loop because each component will require certain semantic information and not all the information available at class level. We will exploit this behaviour in the following section.

4. Rationale

Software architecture design for autonomous vehicles is analogous to the design of a real-time, intelligent, control system - aka a robot. Although we can find considerably literature concerning software architecture in the field of robotics and artificial intelligence [11, 12, 13, 14, 15, 16, 17], these proposals seem to be overlooked by automotive software engineers. Therefore, many reference architectures for autonomous vehicles miss developments and trade-offs explored in the field of robotics - as we will shall see in Section 9. We aim to bridge this gap in this section, by discussing the most important developments in the field of robotics and select the best choices for the automotive domain.

For a long time, the dominant view in the AI community was that a control system for autonomous robots should be composed of three functional elements: a sensing system, a planning system and an execution system [18]. This view led to the ubiquitous sense-plan-act (SPA) paradigm. However, this architecture has a number of shortcomings: at first, because planning is time consuming, the world may change during this phase. Secondly, an unexpected outcome from the execution of a plan step can cause next plan steps to be executed in an inappropriate context.

One question naturally stood up from these shortcomings: how important is the internal state modelling? (which generated several definitions, meant to achieve approximate goals). Maes [11] first distinguishes between *behaviour* and *knowledge* based systems - where knowledge-based systems maintain an internal state of the environment, while behaviour-based systems do not [11].

Similarly, the literature distinguishes between *deliberative* and *reactive* systems, where deliberative systems *reason* upon an internal representation of the environment and *reactive* system fulfil goals through *reflexive* reactions to environment changes [13, 14, 15, 16].

We find both definitions to answer the same questions - how will a system take decisions? Through reasoning on complex semantic information extracted from its sensors or by simple reactions to simple inputs? Deciding on this is an initial trade-off between speed of computation and the amount of environmental understanding a system can have.

When considering the development of autonomous vehicles through these lenses, we can see that vehicles require both reactive and deliberative components. Maintaining a pre-defined trajectory and distance from the objects around a vehicle is an example of a reactive system, which should operate with high frequency and be as fast as possible in order to overcome any environmental change. In this case, maintaining a complex representation of the surrounding environment is futile.

However, a decision making mechanism respon-

sible, for example, to overtake the vehicle in front is an example of a deliberative system. In this case maintaining a complex world model can help the system to take a better decision.

For example, one can not only judge the distances to the surrounding objects, but also the *relevance* of the decision in achieving the goal. Is it worth to overtake the car in front if the vehicle must turn right in a relatively short distance after the overtake? In order to answer this question a complex world model that must combine semantic information about the overall goal, future states and nearby objects is needed. Processing this amount of information will naturally take a longer time. However, since the result can only impact the passengers comfort (assuming that driving behind a slow car for a long time is un-comfortable) the system can assume this processing time.

Gat and Bonnasso [13] first debate the role of internal state and establish a balance between reactive and deliberative components inside a system. In their proposal, the functional components are classified based on their memory and knowledge about the internal state in: *no knowledge*, knowledge of the *past*, or knowledge of the *future* - thus resulting in three layers of functional components. However, their model does not specify how, or if, the knowledge can be shared between the layers. Moreover, it is not clear how and if one any components incorporate knowledge about past, future and other static data.

A better proposal, that bridges the gap between reactive and deliberative components, is the NIST *Real Time Control Systems* (RCS) reference architecture introduced by Albus [12]. This architecture does not separate components based on memory, but builds a hierarchy based on semantical knowledge.

Thus, components lower in the hierarchy have limited semantic understanding and can generate inputs for higher components, deepening their semantic knowledge and understanding. Moreover, RCS has no temporal limitations for a component's knowledge. One can hold static or dynamic information about past, present or future. Although all components maintain a wold model, this can be as simple as reference values, to which the input must be compared.

We find this proposal a good fit for automotive requirements and for the functional classification presented in Section 3 because it allows a balanced representation of reactive and deliberative components and it allows hierarchical semantic processing - one of the requirements given by the classification of functional components. Therefore, we decide to introduce more details about it and illustrate it in Figure 3.

At the heart of the control loop for an RCS node is a representation of the external world - the world model - which provides a site for data fusion, acts as a buffer between perception and behaviour, and supports both sensory processing and behaviour generation.

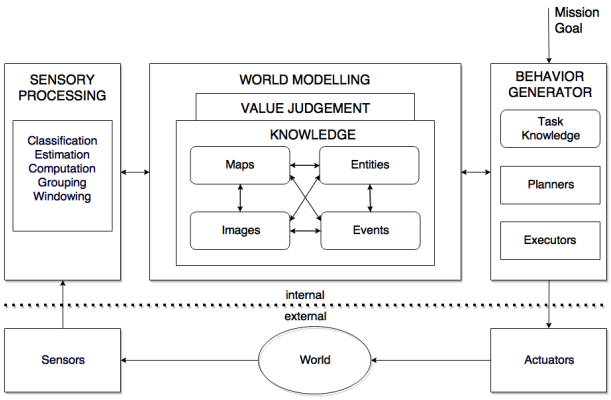

Fig. 3. Real time intelligent control systems reference architecture [19].

Sensory processing performs the functions of windowing, grouping, computation, estimation, and classification on input from sensors. World modelling can also maintain static knowledge in the form of images, maps, events or relationships between them. Value judgment provides criteria for decision making, while behaviour generation is responsible for planning and execution of behaviours [12].

Albus proposed the design for a node in a hierarchical control structure, where lower level nodes can generate inputs for higher level nodes, thus increasing the level of abstraction and cognition. Therefore, nodes lower in a hierarchy can have a very simplistic world model and behaviour generation functions and can easily implement reactive components. In order to keep in line with the example above, we consider

the case of maintaining a distance from an object in front. A node implementing this functionality will only have to keep the distance from the vehicle in front, the reference distance and the vehicle's speed in the world model block. The behaviour generation block will decide to issue a braking command whenever the distance will be to close or whenever it predicts (using the speed of the vehicle) that the distance will decrease.

Higher nodes, representing deliberative components, can easily be described by the same architecture. Their world model block will process more semantic information and generate more complex behaviours (such as the decision to overtake the vehicle in front in order to increase the overall ride comfort and optimise the goal).

In this hierarchy, higher nodes consume semantic information extracted by lower nodes. However, this is not always the case with autonomous vehicles where several nodes, at different hierarchical levels, can consume the same information. For example, the distance from the vehicle in front can be used by both the reactive and the deliberative components introduced earlier. Moreover, several components at the same hierarchical layer can interact, as a sequential process.

From an architectural point of view, a sequential processing stream which follows different, individual, processing steps is represented through a *pipes and filters* pattern [20]. The pattern divides a process in several sequential steps, connected by the data flow - the output data of a step is the input to the subsequent step. Each processing step is implemented by a *filter* component [20].

In its pure form, the pipes and filters pattern can only represent sequential processes. For hierarchical structures, a variant of the pattern called *tee and join pipeline systems* is used [20]. In this paradigm, the input from sensors is passed either to a low level pipeline corresponding to a low level control loop, to a higher level pipeline or both.

An example is shown in Figure 4: the input from sensors is fed to different processing pipes. At a low level of abstraction, pipeline 1 only executes simple operations and sends the result to actuators.

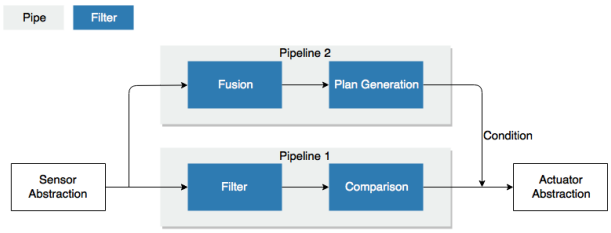

Fig. 4. Tee and join pipelines architectural pattern.

At higher levels of abstraction, pipeline 2 processes more sensor data and generates manoeuvre plans which are translated to actuator language. A priority condition will decide which input to send to the actuators. Alternatives to this patterns will be further discussed in Sections 6 and 7. At the moment, we concentrate on the functional decomposition, which will suggest the nodes in the future architecture.

5. Functional Decomposition

We start by introducing the functional components and, in Section 6, discuss interaction patterns. Figure 5 depicts the functional components that satisfy SAE J3016 requirements for fully autonomous vehicles. The data flows from left to right; from the sensors abstraction to actuator interfaces, simulating a closed control loop. The figure represents three types of entities: functional components (blue boxes), classes of components (light grey boxes), and sub-classes of components (dark grey boxes).

The proposal maps onto the SAE classification of functional components, introduced in Section 2, in the following way: *vehicle control* and *actuators interface* class of components correspond to SAE operational functions, the *planning* class of components corresponds to SAE tactical functions, and the *behaviour generation* class maps to both strategic and planning class of functions.

Two orthogonal classes, corresponding to *data management* and *system and safety management*, are depicted because they represent cross-cutting concerns: data management components implement long term data storage and retrieval, while system and safety management components act in parallel of normal control loops and represent DDT fall-back mechanisms or other safety concerns.

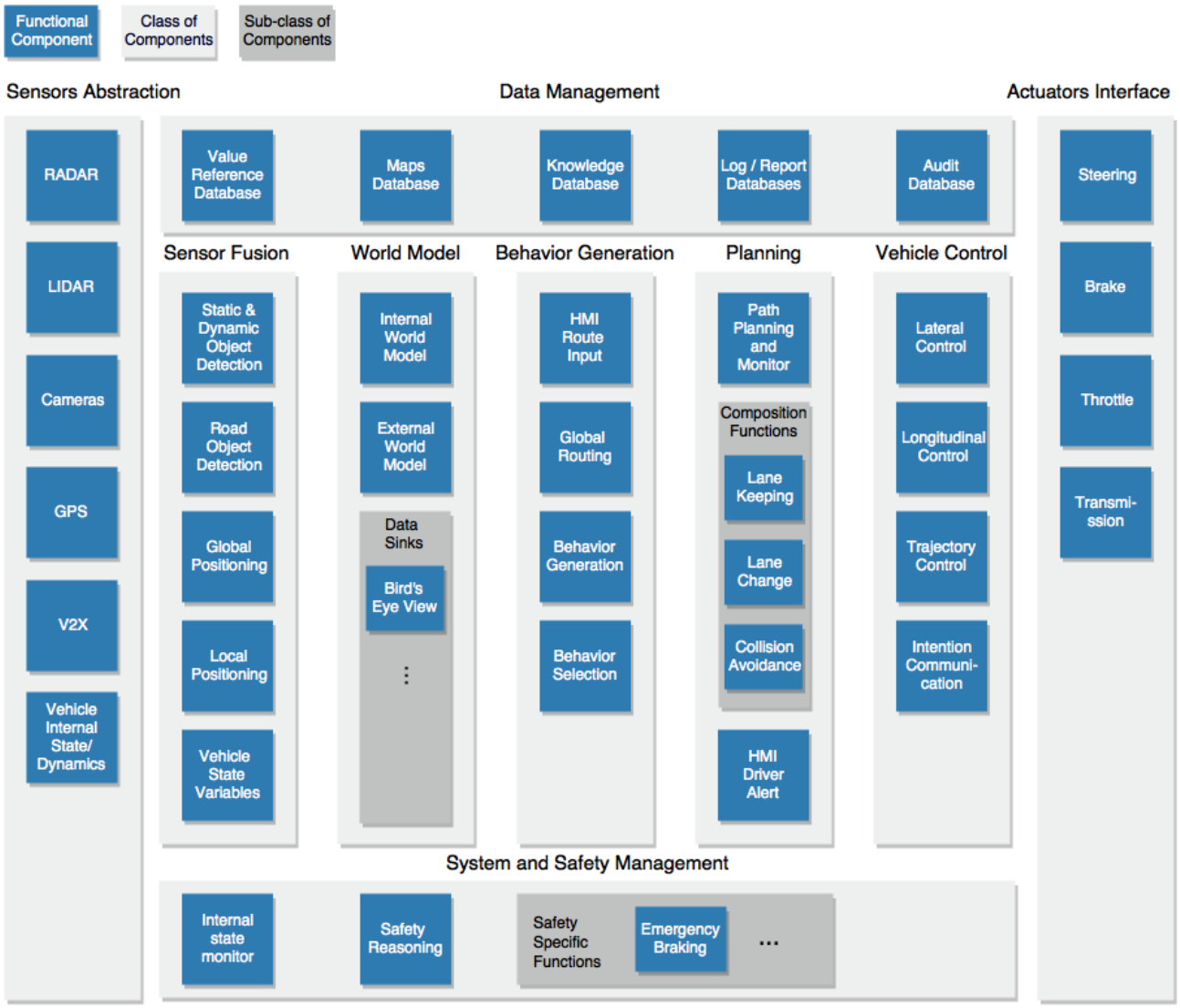

Fig. 5. Proposed functional architecture, part I: functional components.

In the following subsections each class of filters is discussed together with its components. The last sub-section discusses the relation with middle-ware solutions and AUTOSAR.

5.1. Sensor Abstractions

Sensor abstractions provide software interfaces to hardware sensors, possible adapters and conversion functionality needed to interpret the data. We distinguish two classes of sensors and, respectively, of abstractions: (1) sensors that monitor the internal vehicle state or dynamic attributes (e.g. inertial measurements, speed, etc.) and (2) sensors that monitor the external environment as required by the SAE requirements.

Environmental monitoring can be based on RADAR, LIDAR and camera technologies. In the case of cooperative driving, communication with other traffic participants is realised through *vehicle-to-everything* (V2X). Global positioning (GPS) is required to localise the vehicle in a map environment or to generate global routes and is therefore

represented as a separated functional component.

All abstractions concerning the internal vehicle state are grouped into one functional component, because the choice is specific to each OEM.

5.2. Sensor Fusion

Multi-sensor environments generate large volumes of data with different resolutions. These are often corrupted by a variety of noise and clutter conditions which continually change because of temporal changes in the environment. Sensor fusion combines data from different, heterogeneous, sources to increase accuracy of measurements.

The functional components are chosen with respect to SAE requirements for object and event detection, recognition, and classification. We distinguish between *static and dynamic objects* (e.g. a barrier, pedestrians) and *road objects* (e.g. traffic lights) because they have different semantic meaning. Moreover, *local positioning* is needed to position the vehicle relative to the identified objects and *global positioning* is needed for strategic functionality.

Through sensor fusion, a processing pipeline gathering information from various sensors such as RADAR and camera can be imagined in order to classify an object, determine its speed, and add other properties to its description. The distinction between the external environment and the internal state of a vehicle is preserved in Figure 5: the first four components process data related to the external environment, while the internal state is represented by the last functional component.

5.3. World Model

The world model represents the complete picture of the external environment as perceived by the vehicle, together with its internal state. Data coming from sensor fusion is used together with stored data (e.g. maps) in order to create a complete representation of the world.

As in RCS architecture, the world model acts as a buffer between sensor processing and behaviour generation. Components in this class maintain knowledge about images, maps, entities and events, but also relationships between them. World modelling stores and uses historical information (from past processing loops) and provides interfaces to query and filter its content for other components. These interfaces, called *data sinks*, filter content or group data for different consumers in order to reveal different insights. One example heavily used in the automotive industry is the bird's eye view. However, the deployed data sinks remain OEM-specific.

5.4. Behaviour Generation

Behaviour generation is the highest cognitive class of functions in the architecture. Here, the system generates predictions about the environment and the vehicle's behaviour. According to the vehicle's goals, it develops multiple behaviour options (through *behaviour generation*) and selects the best one (*behaviour selection*). Often, the vehicle's behaviour is analogously to a *Finite State Machine* (FSM). The behaviour generation module develops a number of possible state sequences from the current state and the behaviour reasoning module selects the best alternative. Complex algorithms from *Reinforcement Learning* (RL) use driving policies stored in the knowledge database to reason and generate a sequence of future states. Nevertheless, the functional loop is consistent: at first a number of alternative behaviours are generated, then one is selected using an objective function (or policy).

A vehicle's goal is to reach a given destination without any incident. When the destination changes (through a *Human Machine Interface* (HMI) input), the *global routing* component will change the goal and trigger new behaviour generation. These components correspond to the SAE strategic functions.

5.5. Planning

The planning class determines each manoeuvre an autonomous vehicle must execute in order to satisfy a chosen behaviour. The *path planning and monitoring* component generates an obstacle free trajectory and composes the trajectory implementation plan from *composition functions* deployed on the vehicle. It acts like a supervisor which decomposes tasks, chooses alternative methods for achieving them, and

monitors the execution. The need to re-use components across vehicles or outsource the development leads the path to compositional functions. Examples of such functions are: lane keeping systems or automated parking systems (all, commercial of-the-shelf deployed products). Compositional functions represent an instantiation of the RCS architecture; they receive data input from sensor fusion or world modelling through data sinks, judge its value and act accordingly, sending the outputs to vehicle control. Path planning and monitoring acts as an orchestrator which decide which functions are needed to complete the trajectory and coordinate them until the goal is fulfilled or a new objective arrives. For vehicles up and including level 4, which cannot satisfy full autonomous driving in all driving conditions, the control of the vehicle must be handed to a trained driver in case a goal can not be fulfilled. Therefore, this class includes a *driving alert* HMI component.

5.6. Vehicle Control

Vehicle control is essential for guiding a car along the planned trajectory. The general control task is divided into lateral and longitudinal control, reflecting the SAE requirements for operational functions. This allows the control system to independently deal with vehicle characteristics (such as maximum allowable tire forces, maximum steering angles, etc.) and with safety-comfort trade-off analysis. The *trajectory control* block takes a trajectory (generated at the path planning and monitoring level) as input and controls the lateral and longitudinal modules. The trajectory represents a future desired state given by one of the path planning compositional functions. For example, if a lane-change is needed, the trajectory will represent the desired position in terms of coordinates and orientation, without any information about how the acceleration, steering or braking will be performed. The *longitudinal control* algorithm receives the target longitudinal state (such as brake until 40 km/h) and decides if the action will be performed by accelerating, braking, reducing throttle, or using the transmission module (i.e. engine braking). The *lateral control* algorithm computes the target steering angle given the dynamic properties of a vehicle and the target trajectory. If the trajectory includes a change that requires signalling, the communication mechanisms will be triggered through the *intention communication* module.

5.7. Actuator Interfaces

The actuator interface modules transform data coming from vehicle control layer to actuator commands. The blocks in Figure 5 represent the basic interfaces for longitudinal and lateral control.

5.8. Data Management

Data will at the centre of autonomous vehicles [21]. In spite of the fact that most data requires real-time processing, persistence is also needed. These concerns are represented using the data management class of components. Global localisation features require internal *maps storage*; intelligent decision and pattern recognition algorithms require trained models (*knowledge database*); internal state reporting requires advanced logging mechanisms (*logging database*). The *logging-report* databases are also used to store data needed to later tune and improve intelligent algorithms. Moreover, an *audit database* keeps authoritative logs (similar to black boxes in planes) that can be used to solve liability issues. In order to allow dynamic deployable configurations and any change in reference variables (e.g. a calibration or a decisional variable) a *value reference* database is included.

5.9. System and Safety Management

The system and safety management block handles functional safety mechanisms (fault detection and management) and traffic safety concerns. It is an instantiation of the separated safety pattern [22] where the division criteria split the control system from the safety operations. Figure 5 only depicts components that spot malfunctions and trigger safety behaviour (*internal state monitor*, equivalent to a watch dog), but not redundancy mechanisms. The latter implement partial or full replication of functional components. Moreover, *safety specific functions* deployed by the OEM to increase traffic safety are distinctly

represented. At this moment they are an independent choice of each OEM.

As the level of automation increases, it is necessary to take complex safety decisions. Starting with level 3, the vehicle becomes fully responsible for traffic safety. Therefore, algorithms capable of full safety reasoning and casualty minimisation are expected to be deployed. While it is not yet clear how *safety reasoning* will be standardised and implemented in future vehicles, such components will soon be mandatory [23]. An overview of future safety challenges autonomous vehicles face is illustrated in [24]. With respect to the separated safety pattern, in Figure 5 safety reasoning components are separated from behaviour generation.

5.10. AUTOSAR Context

AUTOSAR is a consortium between OEM and component suppliers which supports standardisation of the software infrastructure needed to integrate and run automotive software. This paper does not advocate for or against AUTOSAR. The adoption and use of this standard is the choice of each OEM. However, due to its popularity, we consider mandatory to position the work in the standard's context. Given AUTOSAR, the functional components in Figure 5 represent AUTOSAR *software components*. The interfaces between components can be specified through AUTOSAR's standardised interface definitions.

6. Interaction between Components

As mentioned in Section 4, the components in Figure 5 act as a hierarchical control structure, where the level of abstraction and cognition increases with the hierarchical level, mapping on the SAE classification of functional components. Components lower in the hierarchy handle less complex tasks such as operational functions, while higher components handle planning and strategic objectives (e.g. global routing or trajectory planning).

We propose the use of pipe-and-filter pattern for component interactions in flat control structures (same hierarchical level) and the use of tee-and-join pipelines to represent the hierarchy. In a hierarchical design, lower level components offer services to adjacent upper level components. However, the data inputs are often the same. A high level representation of the system, through the tee-and-join pipelines pattern is illustrated in Figure 6. The grey boxes represent processing pipelines and the blue ones represent components classes.

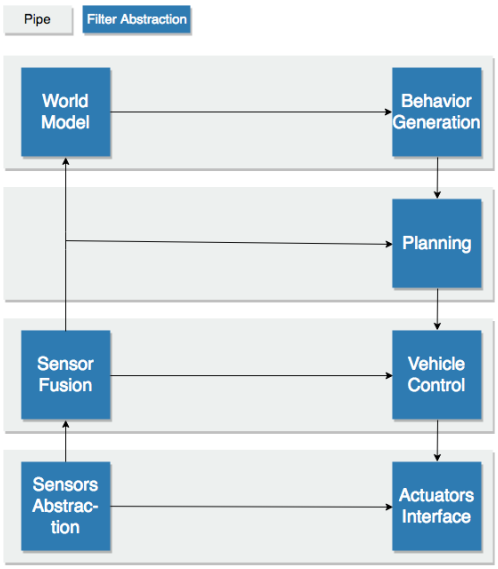

Fig. 6. Proposed functional architecture, part II: hierarchical control structure using tee-and-join pipelines pattern.

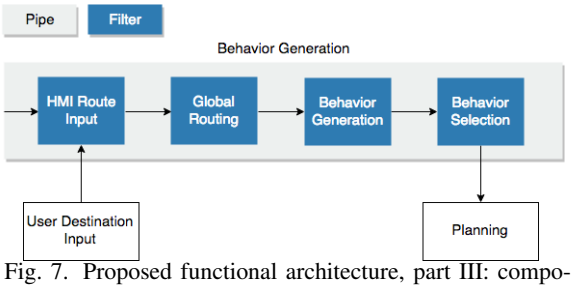

Fig. 7. Proposed functional architecture, part III: component interaction at class level. The behaviour generation process.

For each component class, a process is analogous to a pipeline. As example, once a user communicates a final destination, the behaviour generation process starts. This example is illustrated in Figure 7, where upon receiving a destination at the *HMI input filter*, the *global routing* filter forwards a route to the *behaviour generation* filter. This filter breaks down the route in actions that have to be taken by the vehicle in order to reach the destination. The actions are analogous to states in a FSM. Often, there

are several paths between two states. Further on, the *behaviour selection* component will select the best path between two states and forward it to the *planning* process.

Moreover, messages received at component level have to be prioritised. For example, an actuator interface can receive commands from the longitudinal control component or a safety specific function (e.g. emergency braking). In order to decide which command to execute first, the messages must contain a priority flag. Since this functionality is dependent on the component's interface and specific to OEM, this discussion is limited.

Various alternatives to the chosen interaction patterns will be discussed in the next section.

7. Trade-off Analysis

Software architecture design is often not deterministic. Given a set of requirements, different architects might come to different results. The process of weighting alternatives and selecting the most appropriate decisions is called *trade-off* analysis. Several decisions in this paper could go in other ways. Therefore, in this section we present several alternatives to our decisions and debate about their advantages and disadvantages.

We will start from the underlying assumption of this paper - that we can derive a set of valid requirements from the SAE J3016 automotive standard. As mentioned in Section 2, the purpose of this standard is not to provide a complete definition of autonomy, but rather a minimal illustration of the functions needed in order to achieve it. It is meant to guide the process, and not exhaustively define it. However, we find this information sufficient for our goals, because any functionality on top of the minimal requirements remains the choice of each OEM.

The second decision to be questioned is the use of NIST RCS architecture as a baseline for our work. An in-depth comparison with other works was presented in Section 4, however, we hereby present the trade-offs that come with choosing this reference architecture. Although it has the power to clearly discriminate between reactive and deductive components and is general enough to represent both types of components, this reference architecture can be sometimes too complex for simple nodes.

In order to properly represent simple functions, from the operational class, one does not need a complex world model or behaviour generator modules. However, the architecture does not impose any constraints on the complexity of the world model. Thus, with limited world modelling or behaviour generation module the architecture can easily represent very simple control loops such as value comparators. Moreover, when compared to other proposals, this reference model allows each individual nodes to have a level of reasoning - while in other only components higher in a hierarchy are responsible for planning. This comes as a benefit for the automotive software development life-cycle and for the compositional functions we represented in the architecture.

We expect OEM to maintain their status of technology integrators and outsource more complex functionality to their suppliers - such as complete lane keeping or collision avoidance systems. These systems can easily be implemented as RCS nodes and integrated in the overall architecture.

We find the RCS model broad enough to allow the development of complex functionality, but sometimes too complex for simple functions - a trade-off that can be easily overcome by simplifying each block of the architecture in the latter case. A more expressive alternative would be to propose RCS nodes of different complexity - with lower level nodes that can replace world modelling only with value judgement. This decision must be weighted independently, as it might add clutter and other trade-offs when deciding which type of node should one choose for each component.

A third trade-off clearly regards the functional decomposition. Although we aim to atomically decompose every function and represent each subcomponent individually, there is no way to prove this is the correct way. New market trends foresee the adoption of system-on-a-chip circuits that integrate more and more components. Technologies such as MobilEye [25] aim to group several functions on a dedicated chip. In such scenarios, sensor abstractions and fusion layers could be merged. However, the functions implemented by such circuits will still

resemble our proposal. For example, even though the sensor abstraction layer compresses to one component, all the functionality at the sensor fusion layer still have to be implemented because static, dynamic or road objects detection is absolutely mandatory for autonomous driving. Therefore, our atomic decomposition is able to represent these evolutions.

A fourth trade-off to be considered concerns the choice for component interaction patterns. Several alternatives to the tee and join pipelines pattern have been considered. One clear alternative is to use a layered architecture - where the functional components are grouped in a hierarchy and function calls can be made from higher layers to lower ones [4]. However, this choice will constrain the possible interactions between components, thus limiting the design. At first, because each layer will encapsulate some components, it will be impossible to reuse or re-orchestrate them in other processes. Secondly, because the function calls at lower layers are required to come from upper layers, thus limiting even more the design.

Another alternative is to use a component based architecture, where all the components rest at the same hierarchical layer and any component can communicate with all others. This pattern seems a better fit for the automotive domain, where different components are deployed on various ECU and they communicate through a bus. However, it is unable to represent hierarchies.

We argue that the tee and join pattern is equivalent to the component based architecture, but has the ability to represent hierarchical processes. This is because from a component, one can easily begin a new pipeline with any other components. If only the pipes and filter pattern was considered, than the design would be much limited. However, by giving the ability to re-orchestrate filters in different pipelines, the tee and join pipelines pattern can easily represent any kind of processes, either flat or hierarchical.

In this scenario one can define flat processes, similar to component based orchestration, but also hierarchical processes (similar to the hierarchical functional classes introduced in Section 3). The only thing to consider is a processing priority - which defines which pipelines should execute first.

8. Discussion

Software architects evaluate both the functional suitability of an architecture and non functional properties such as performance or maintainability [26]. In this paper we are only interested in functional suitability and completeness with respect to SAE J3016 requirements. However, we find of interest to discuss two other important aspects: the position of the proposed architecture with respect to (1) the automotive software development life cycle and (2) the ISO 26262 standard that regulates functional safety. Later, in Section 9, we provide a comparison with existing literature.

8.1. Incremental development and component reuse

The SAE classification presented in Section 2 shows an incremental transition from partially automated to fully autonomous vehicles. The functional division of software components should respect this incremental transition. Moreover, the OEM software development life-cycle and preference for outsourcing must be taken into account.

As mentioned in Section 2, DDT automation is analogous to deploying and orchestrating enough driving automation features in order to satisfy all driving conditions (ODD) in which a human can drive. This assumption employs two development paths:

1. the deployment of new software components specific to new DDT features or
2. updating a driving feature with enhanced functionality.

In Figure 5, new DDT features represent new compositional functions specific to path planning. The use of composition functions enables incremental and distributed development at the cost of increased complexity for *path planning and monitor*. These components can be commercial-of-the-shelf products that can easily be outsourced to tier one suppliers.

Behaviour generation improvements are solved through knowledge database updates. The V2X

component interfaces with the external world, therefore, updates can be pushed through this component. In most cases, the updates will target the knowledge or value reference databases.

8.2. Functional safety

The automotive industry has high *functional* safety constraints imposed by the mandatory adherence to ISO 26262 [7]. The objective of functional safety is to avoid any injuries or malfunctions of components in response to inputs, hardware or environmental changes. Error detection and duplication of safety critical components are mechanisms suggested by ISO 26262. In this proposal, we represent the functional component specific to error detection, however, omit to represent any redundancy or duplicated components.

We also aim to fulfil a gap in the ISO 26262 standard, with regards to autonomous vehicles: safety reasoning [24]. To this moment it is not clear how autonomous vehicles will behave in case an accident can not be avoided and which risk to minimise. However, it is expected for future safety standards to include specification for safety behaviour.

9. Related work

We focus on literature proposing functional and reference architectures starting with level 3, since level 2 vehicles only automate lateral and longitudinal control. A historical review of level 2 systems is presented in [27].

The only reference architecture for fully autonomous (level 5) vehicles was introduced by Behere et al. [1]. In this proposal, the authors make a clear distinction between cognitive and vehicle platform functionality, similar to the classification in tactical and operational SAE classes. The decision and control block [1] is responsible for trajectory reasoning, selection and implementation, equivalent to the behaviour generation and planning class of components from Figure 5. Yet it is not clear how this block handles all functionality, leading to a rough representation of functional classes. It is also interesting to observe that HMI components are ignored.

Other literature from this field focused primarily on systems developed for autonomous driving competitions or other constrained experiments. Introducing a project which attended one of the first competitions organised by DARPA, Montmerlo et al. [28] show a layer-based architectural model based on sensor interface, perception, navigation, user and vehicle interfaces. In this model localisation features are embedded in perception together with laser object detection. No object recognition or classification was needed in this competition. The model represents operational and tactical functions through navigation components, but excludes strategic functions.

Jo et al. [29, 30] present their experience from an autonomous vehicle competition held in Korea. The proposal comes one step closer to a general architecture, given broader competition goals. The model contains sensor abstractions, fusion, behaviour and path planning, vehicle control and actuator interfaces. In this regard, it represents similar concerns to Figure 5, without world modelling and HMI route inputs. Instead, the behaviour planning component integrates data coming from sensors in order to generate an execution plan. Since the goal of the competition was limited, both localisation and behaviour reasoning components are restricted (a finite state machine with only 8 possible states that can stop for a barrier, detect split road, etc.). The artefact successfully represents operational and tactical functions. Moreover, Jo et al. divide, for the first time, the concerns from behaviour and from path planning, thus obtaining several levels of cognition and control. The study also reveals important details for in-vehicle ECU deployment and a mapping to AUTOSAR.

An important contribution from industry research is the work of Ziegler et al. [31] at Mercedes Benz. Although its purpose is not to introduce a general functional architecture, the system overview reveals similar functional requirements. It features object recognition, localisation, motion planning and vehicle control, analogous to behaviour generation, planning and vehicle control in Figure 5. Once again, the concerns for behaviour generation are separated from path and trajectory planning, and

grouped under motion planning. Another important contribution is the representation of data storage functionality for digital maps and reactive components such as emergency braking.

Overall, we observe two approaches in the literature: (1) a high level overview of system components or (2) proofs-of-concept from experiments with autonomous features or competition with limited operational domain. This paper takes one step further and considers a fine-grained functional decomposition with respect to the automotive software development life cycle. Moreover, data concerns are central to the proposal, both for long term storage and fast update of reasoning and cognitive models. We advocate advanced logging mechanisms with specific architectures and data models that can help in fast identification of malfunctions and could be later used to improve learning algorithms.

10. Conclusions and future research

We have presented a functional software architecture for fully autonomous vehicles. Since the automotive industry is highly standardised, we follow the functional requirements from an automotive standard which defines multiple levels of driving automation and includes functional definitions for each level.

During the architecture design, we aim to respect the incremental development process of autonomous vehicles and the distributed software development process specific to the automotive industry. The final artefact represents an automotive specific instantiation of the NIST RCS reference architecture for real-time, intelligent, control systems. We use the pipe-and-filter architectural pattern for component interaction and the tee-and-join pipeline pattern to represent a hierarchical control structure. Several trade-offs and alternative decisions are discussed within the paper.

Future work might include refinement through expert opinion. Later steps consider component interface design, a choice for hardware architecture, functional component distribution across ECUs and component distribution inside local networks in order to satisfy security requirements.

References

[1] S. Behere and M. Törngren, "A functional reference architecture for autonomous driving," *Information and Software Technology*, vol. 73, pp. 136–150, 2016.

[2] M. Broy, "Challenges in automotive software engineering," in *International Conference on Software Engineering (ICSE'06)*, pp. 33–42, ACM, 2006.

[3] Society of Automotive Engineers (SAE), "J3016," *SAE international taxonomy and definitions for terms related to on-road motor vehicle automated driving systems," levels of driving automation*, 2014.

[4] M. Staron, *Automotive Software Architectures*, vol. 1. Springer International Publishing, 2017.

[5] D. Garlan, "Software architecture: a roadmap," in *Conference on The Future of Software Engineering (ICSE'00)*, pp. 91–101, ACM, 2000.

[6] A. Serban, E. Poll, and J. Visser, "A standard driven software architecture for fully autonomous vehicles," *Proceedings of WASA Workshop*, 2018.

[7] International Organization for Standardization (ISO), "ISO standard 26262:2011 Road vehicles - Functional safety," 2011.

[8] N. B. Ruparelia, "Software development lifecycle models," *ACM SIGSOFT Software Enginering Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[9] R. Wieringa, "Design science methodology: principles and practice," in *International Conference on Software Engineering (ICSE'10)*, pp. 493–494, ACM, 2010.

[10] R. Horowitz and P. Varaiya, "Control design of an automated highway system," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 913–925, 2000.

[11] P. Maes, "Behavior-based artificial intelligence," in *Proceedings of the Fifteenth Annual Meeting of the Cognitive Science Society*, pp. 74–83, 1993.

[12] J. S. Albus, "The NIST real-time control system (RCS): an approach to intelligent systems research," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 157–174, 1997.

[13] E. Gat and R. P. Bonnasso, "On three-layer architectures," *Artificial intelligence and mobile robots*, vol. 195, p. 210, 1998.

[14] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, and C. Plaunt, "Idea: Planning at the core of autonomous reactive agents," in *NASA Workshop on Planning and Scheduling for Space*, 2002.

[15] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti, "The Saphira architecture: A design for autonomy," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 215–235, 1997.

[16] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty architecture for robotic autonomy," in *IEEE Aerospace Conference*, vol. 1, pp. 121–132, IEEE, 2001.

[17] "An architectural blueprint for autonomic computing," tech. rep., IBM, 2006. White paper. Fourth edition.

[18] N. J. Nilsson, *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[19] J. S. Albus and W. Rippey, "Rcs: A reference model architecture for intelligent control," in *From Perception to Action Conference, 1994., Proceedings*, pp. 218–229, IEEE, 1994.

[20] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-oriented Software Architecture*, vol. 5. John Wiley & Sons, 2007.

[21] M. Johanson, S. Belenki, J. Jalminger, M. Fant, and M. Gjertz, "Big automotive data: Leveraging large volumes of data for knowledge-driven product development," in *Big Data (Big Data), 2014 IEEE International Conference on*, pp. 736–741, IEEE, 2014.

[22] J. Rauhamäki, T. Vepsäläinen, and S. Kuikka, "Functional safety system patterns," in *Proceedings of VikingPLoP*, Tampere University of Technology, 2012.

[23] J.-F. Bonnefon, A. Shariff, and I. Rahwan, "The social dilemma of autonomous vehicles," *Science*, vol. 352, no. 6293, 2016.

[24] A. Serban, E. Poll, and J. Visser, "Tactical safety reasoning. a case for autonomous vehicles.," *Proceedings of Ca2V Workshop*, 2018.

[25] N. Mobileye, "Introduces eyeq vision system-on-a-chip high performance," *Low Cost Breakthrough for Driver Assistance Systems, Detroit, Michigan*, 2004.

[26] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 638–653, 2002.

[27] A. Khodayari, A. Ghaffari, S. Ameli, and J. Flahatgar, "A historical review on lateral and longitudinal control of autonomous vehicle motions," *ICMET*, pp. 421–429, 2010.

[28] M. Montemerlo, J. Becker, S. Bhat, *et al.*, "Junior: The Stanford entry in the urban challenge," *Journal of Field Robotics*, vol. 25, no. 9, pp. 569–597, 2008.

[29] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of autonomous car - part I," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 12, pp. 7131–7140, 2014.

[30] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of autonomous car - part II," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 8, pp. 5119–5132, 2015.

[31] J. Ziegler, P. Bender, M. Schreiber, H. Lategahn, T. Strauss, C. Stiller, T. Dang, U. Franke, N. Appenrodt, C. G. Keller, *et al.*, "Making Bertha drive - an autonomous journey on a historic route," *IEEE ITS Magazine*, vol. 6, no. 2, pp. 8–20, 2014.