Chapter 1
Systems Engineering and Architecting for Intelligent Autonomous Systems

Sagar Behere and Martin Törngren

Abstract This chapter provides an overview of architecture and systems engineering for autonomous driving system, through a set of complementary perspectives. For practitioners, a short term perspective uses the state of the art to define a three layered functional architecture for autonomous driving, consisting of a vehicle platform, a cognitive driving intelligence, and off-board supervisory and monitoring services. The architecture is placed within a broader context of model based systems engineering (MBSE), for which we define four classes of models: Concept of Operations, Logical Architecture, Application Software Components, and Platform Components. These classes aid an immediate or subsequent MBSE methodology for concrete projects. Also for concrete projects, we propose an implementation setup and technologies that combine simulation and implementation for rapid testing of autonomous driving functionality in physical and virtual environments. Future evolution of autonomous driving systems is explored with a long term perspective looking at stronger concepts of autonomy like machine consciousness and self-awareness. Contrasting these concepts with current engineering practices shows that scaling to more complex systems may require incorporating elements of so-called *constructivist* architectures. The impact of autonomy on systems engineering is expected to be mainly around testing and verification, while implementations shall continue experiencing an influx of technologies from non-automotive domains.

Sagar Behere
KTH The Royal Institute of Technology, Stockholm, Sweden e-mail: behere@kth.se

Martin Törngren
KTH The Royal Institute of Technology, Stockholm, Sweden e-mail: martint@kth.se

1.1 Introduction

This chapter provides practical insights into specific systems engineering and architecture considerations for building autonomous driving systems. It is aimed at the ambitious practitioner with a solid engineering background. We envision such a practitioner to be interested not just in concrete system implementations, but also in borrowing ideas from the general theory of intelligent systems to advance the state of autonomous driving.

The practical development of autonomous driving systems involves domain specific algorithms, architecture, systems engineering, and technical implementation, as shown in Figure 1.1. Of these, this chapter focuses on the latter three. Architecture and its development may possibly be considered as a part of systems engineering, but for the purpose of this chapter, we treat it as a distinct area. This is because of the extensive coverage of architecture in the chapter. The arrows in Figure 1.1 represent an "impact" relationship. Thus, the arrow from architecture to systems engineering implies that architecture has an impact on systems engineering. The impact can be of various types, but the key point is that the practical development of autonomous driving systems must *holistically* consider all the areas and their impacting interrelations. Such a holistic view is not always within the scope of researchers "deep diving" into the specifics of a particular area. Nevertheless, for practicing engineers and concrete projects, it can not (should not) be ignored. This chapter presents the three areas within a holistic context, with the aim of providing the practicing engineer with a grasp of key concepts in each area, and how they all come together for autonomous driving. Within each area, references to more detailed topics are also provided. This simultaneous consideration of three, typically disparate topics, is one of the innovative aspects of this book chapter. Very often in research, the technical aspects of the implementation are considered less important. In our experience with autonomous driving, the technical implementation aspects provide opportunities and strong constraints which *must* be considered during the processes of architecting and systems engineering. This prevents gaps imposed by "reality" when transitioning between different concerns and a subsequent weakness in the application of theoretical results. Therefore, this chapter devotes a complete section to the core tools and technologies supporting architecting, systems engineering, and implementation. This is a second innovative aspect of this chapter. The third innovation is an examination of key results in the areas of machine consciousness and the theory of mind, which are usually omitted from "hardcore" engineering discussions, because the gap between these areas and pragmatic, safety critical engineering is very large. But by covering these areas, we show how it can influence and provide guidelines for future developments in autonomous driving. The chapter deliberately emphasizes breadth of coverage rather than depth in one selected topic, because we have noticed a conspicuous lack of literature in the field that collects together the important considerations and topics relevant to the

practicing engineer. It must be noted that some of the presented topics are not exclusive to the development of autonomous systems, but gain significant importance in that context.

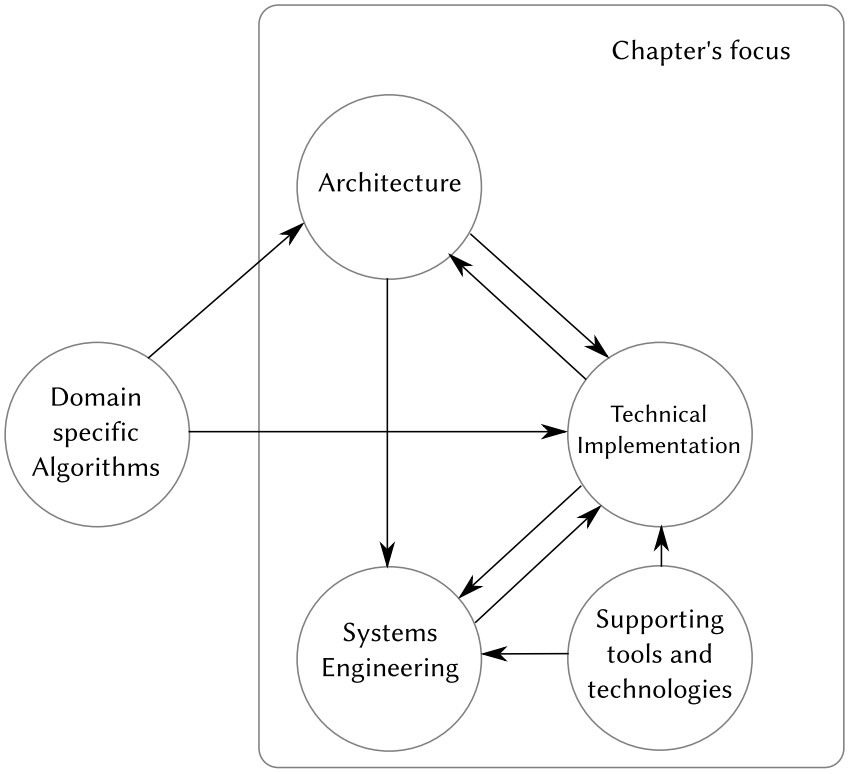

Fig. 1.1 The areas in focus of this chapter

The chapter thus contributes with a series of takeaways in the areas of architecture, systems engineering, technical implementation, and longer term evolution of autonomous driving. For architecture, the reader will find explicit descriptions of key functional components needed for autonomous driving and a three layered reference architecture showing the distribution of these components and their interconnections. For model based systems engineering (MBSE), the chapter outlines four classes of models to aid MBSE methodologies for concrete projects. The model classes may be viewed as a lightweight complement to methodologies advocated by the 'V-model' and functional safety standards like ISO26262. For technical implementation, a development setup and key technologies are described. For longer term evolution, the chapter provides a brief overview of stronger autonomy concepts like machine consciousness and self-awareness, and relates them to current engineering practices. This is used to point out directions for evolution of current

autonomous driving architectures. The impact of autonomy on architecting, systems engineering, and technical implementations in the automotive domain is also briefly discussed.

The scope is delimited to a broad treatment of functional architectures and systems engineering concerns relevant for autonomous driving. The emphasis is on early stages of development and prototyping. Concerns exclusively related to the engineering of safety critical systems, as well as human machine interaction, metrics for architecture and systems engineering, and topics related to engineering ethics are not covered.

The chapter is structured as follows: following this Introduction, Section 1.2 provides a description of the research method, followed by a summary of important terms in the text in Section 1.3 provides a quick description of important terms in the text. This is followed by Section 1.4 which provides a longer term perspective of the research area, exploring more abstract concepts for enabling machine autonomy. The paper then becomes progressively more concrete via Section 1.5 on Architecture, Section 1.6 on Systems Engineering, and Section 1.7 on Technical Implementation. These chapters cover the immediate and short term perspective in the field. The progression from abstract to concrete is deliberately selected to guide the reader's thinking from the more esoteric and principled notions of machine autonomy to a practical immersion in the engineering state-of-practice. Finally, Section 1.8 presents a synthetic discussion that reflects on each of the covered areas, and how they are affected by the characteristics of autonomy.

1.2 Research method

Research methods of Engineering Design have been used to generate this chapter's content. Engineering design is one of the research methods in systems engineering [44] wherein researchers address a problem which is important and novel through the activity of designing a solution [45]. The knowledge developed is primarily for practical application. An additional outcome is some theoretical development based on generalization of design experiences. A potential weakness of engineering design, as a qualitative research method, is that of *external validity*. This is addressed here by a multitude of different case studies and engagement with experts in different domains. Since 2010, we have designed solutions for, and engaged in a variety of self-driving vehicle projects. The projects have involved a variety of commercial and research vehicles, in academic and industrial contexts. A concise summary is provided in Table 1.1. In 2010, an architecture for autonomous longitudinal motion control was designed and implemented on an R730 commercial truck from Scania CV AB. The truck participated in the Grand Cooperative Driving Challenge (GCDC) 2011, wherein vehicles operated autonomously on a public highway in a platooning scenario, with constant wireless communica-

tion between participating vehicles and the environment. The communication contained operating parameters of each vehicle, like its speed, acceleration, location etc. as well as the states of infrastructural elements like traffic lights and prevailing speed limits. The architecture was refined and re-applied an year later, on a different truck (an R430 model), for the CoAct 2012 project. This project also involved a platooning scenarios similar to the GCDC 2011 event, but it included more demanding operational situations like splitting and merging lanes, and overtaking. The accumulated architecture underwent further evaluation and analysis in the course of three different projects with various industrial partners, including a potential application to passenger cars. One of these projects was DFEA2020 — a large Swedish consortium project aimed at development of green, safe, and connected vehicles. Another project is FUSE — also a Swedish project, with a tighter focus on functional safety and architectures for autonomous driving. The third project is ARCHER — which investigates safety, reference architectures, and testing and verification techniques applicable to commercial trucks. The FUSE and ARCHER projects are still in progress. Starting from 2014, the architecture was then applied to a novel research concept vehicle (RCV) at KTH, with a view to endow it with autonomous driving capabilities. The RCV has an all-electric, drive-by-wire powertrain with a propulsion motor embedded inside each wheel, and active steering and camber control of all four wheels. The architecture was then adapted to a second variant of the RCV (RCV-2.0), where it serves as the foundation for autonomous urban driving capabilities in situations where a human driver is not expected to be available (or capable) of taking over vehicle control.

Year(s)	Projects	Vehicle	Partners	Outcome
2010-11	GCDC 2011	Heavy duty commercial truck	Scania CV AB (OEM)	1. Autonomous longitudinal motion in platooning scenario [60] 2. A reference architecture for cooperative driving [25]
2011-12	CoAct 2012	Heavy duty commercial truck	Scania CV AB (OEM)	Second, different instantiation of above mentioned reference architecture for cooperative driving
2013-14	DFEA2020 + FUSE + ARCHER	Passenger cars	Volvo Car Corporation (OEM) + Scania CV AB (OEM)	Problem analysis, methods, and a reference architecture for autonomous driving [23, 24]
2014-	RCV	Novel research vehicle prototype	Departments within KTH (Academia)	Novel electric vehicle prototype with -by-wire control of steering and propulsion [83]
2015-	RCV-2.0	Novel research vehicle prototype	KTH + A private company	Novel vehicle prototype with full perception stack and urban autonomous driving capabilities (under development)

Table 1.1 Projects contributing to this chapter's content

1.3 Essential terminology and concepts

In this section, we briefly describe our specific usage of some important terms that occur repeatedly in the text.

Autonomy is used in the practical sense as *a machine's ability to effectively (with respect to its goals) operate in an uncertain environment, without constant human supervision or intervention.* Effective operation implies that the desired goals are met in a safe manner with a desired level of performance. **Safety** implies absence of unacceptable risk.

Architecture is defined by ISO 42010:2011 as *"..fundamental concepts or properties of a system in its environment, embodied in its elements, relationships, and principles of its design and evolution."* The architecture is thus the "blueprint" of the system and one practical way to think of it is to decompose it into conceptual and technical design aspects [33, 32] as shown in Figure 1.2. These aspects may alternatively be referred to as "views" of the architecture, a term recommended by ISO 42010 and pertaining to an architecture description from a specific "viewpoint". A detailed explanation of Figure 1.2 can be found in [23]. Briefly, the process of architecting a system can be initiated by describing the functions which the system shall offer to its user (service taxonomy), without stating how the functions are internally realized by the system. The service taxonomy is realized by the logical architecture, which shows the logical decomposition of the system into its constituent components, without specifying how those components are actually realized in hardware and/or software. The logical architecture components are subsequently mapped onto software elements, which are deployed on hardware computation units. The computation and communication systems may further be partitioned in time and space, depending on a variety of requirements related to performance, availability, safety, prototyping tools etc.

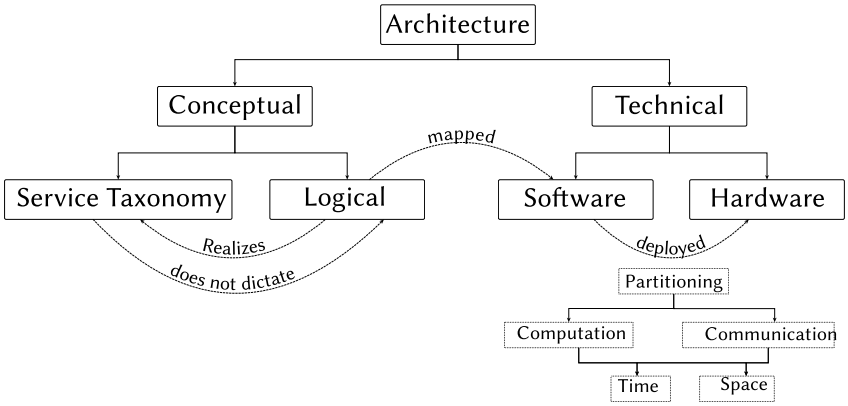

Fig. 1.2 An overview of system architecture

Systems engineering is defined by INCOSE as *an interdisciplinary approach and means to enable the realization of successful systems.* It integrates all the disciplines and domain experts into a team effort forming a structured development process that proceeds from concept to production to operation. Functionality is defined early in the development lifecycle, requirements are documented followed by design synthesis, testing, verification and validation all while considering the complete problem which the system solves. Systems engineering is especially relevant to the construction of large, complex and safety critical systems. The formalized application of modeling to support various systems engineering tasks is termed as **model-based systems engineering**.

1.4 The context of machine consciousness

When thinking about autonomous driving, it is very easy to get lost in practical minutiae of sensors, hardware, programming, modeling etc. However, staying entirely at this level of thinking can lead to "not seeing the forest due to all the trees". As the complexity of systems rises, it is worth taking a step back and asking, "What is it that we are really trying to do, and is this the right direction?"

In this section, we take a step back from the immediate engineering concerns, and look at autonomous driving within the larger and somewhat "philosophical" context of generally autonomous and intelligent machines. The engineering of autonomous driving systems rarely considers investigations into domains like machine consciousness [50], self-awareness, and theories of mind [38]. This is primarily due to two reasons: the technological/experiential background of the engineers involved, and lack of clear mappings from the mentioned domains to engineering concepts for certifiable, safety-critical embedded systems. Nevertheless, we believe that keeping abreast of key results in these domains is valuable for practitioners of autonomous driving because it provides strategic guidance for future architecture concepts and identifies the gaps that prevents utilization of results from these domains.

The ultimate goal for autonomous driving is not just human-like driving but to go beyond human-like driving, in order to overcome human limitations and appreciably increase road safety, traffic efficiency, and environmental benefits. To achieve this goal, and to interact and coexist with human environments, machines would need a level of consciousness that approaches human (admittedly under tightly constrained notions). This is because consciousness is instrumental to reasoning, decision making and problem solving capabilities in the face of uncertainty and disturbances. There is a variety of literature [61] which suggests that robots' problem solving capacities would be enhanced by the ability to introspect. This is also a recurring theme across disciplines like computer Systems-on-Chip [74], programming languanges [57], robotics

and even explorations for fault-tolerant on-board computing for robotic space missions [87].

What would consciousness mean in the context of autonomous driving? A dictionary definition [10] of **consciousness** is *"..the fact of awareness by the mind of itself and the world"* where **awareness** is further defined as *"Knowledge or perception of a situation or fact"*. Within the field of philosophy and cognitive sciences, however, consciousness is recognized as an umbrella term covering a wide variety of heterogenous mental phenomena, often grouped under the categories of Creature Consciousness, State consciousness, and Consciousness as an entity [82]. While comprehensive and precise definitions remain a topic of research and debate, from an engineering perspective we are interested in definitions only insofar as they help us to identify and characterize specific machine behavior. Indeed, as far back as in the 1950s, researchers like Alan Turing believed that questions about *actual* intelligence (and presumably consciousness) were too vague or mysterious to answer. Turing instead proposed a behaviorist alternative [81] wherein if a savvy human judge could not distinguish a computer's conversational abilities from those of a real person [1] at a rate better than chance, then we would have some measure of the computer's intelligence. This measure of perceived intelligence could be substituted for the computer's *real* intelligence or *actual* consciousness. A loose application to autonomous driving could be: If a savvy human judge can consistently accept a computer's driving abilities as equivalent to those of a competent human driver, then we would have some measure of the computer's driving capabilities. Indeed, on 4th May 2012, one of Google's self- driving Prius vehicles was granted a "driving license" by the Nevada state Department of Motor Vehicles, after the vehicle successfully passed driving tests similar to those administered to human drivers[7] .

At this point, is useful to decompose consciousness into awareness of the external and the internal, from the machine's perspective. Awareness of the external world involves elements of sensing, data fusion and perception, all of which demonstrate progressive and on-going improvements within the domain of autonomous driving (compare the sensors available for autonomous driving today, with those from just a decade ago). This awareness of the external world is usually explicitly represented in the internal world of the machine, by maintenance of data structures reflecting perception of the external world. From a philosophical view then, awareness of the external world is "absorbed" by the machine's internal states, to which engineering attention must be devoted, in order to achieve progressive results in machine consciousness. However, for meaningful exploitation of any internal awareness, the machine needs to be aware of the awareness i.e. it needs to be **self-aware**. This is supported by explicit conclusions in the domain of machine consciousness, for example McCarthy states [61] *"..some consciousness of their own mental processes will be required for robots to reach a level of intelligence needed*

[1] This is the popular version. Turing actually framed a somewhat different test, as discussed in [41]

to do many of the tasks humans will want to give them.. consciousness of self i.e. introspection is **essential** *for human level intelligence, not a mere epiphenomenon.".*

Which system characteristics and structures are instrumental for consciousness and self-awareness? One approach to answering this is to understand how the human brain functions, and mimic the biological structures found therein. The strongest approach to understanding human consciousness (and the only one relevant to autonomous driving) is that of *computationalism* [62] which is the theory that many relevant aspects of the human brain can be modeled as having a computational structure. This approach is the basis of the field of *Artificial Intelligence* (AI) which explores computational models of problem solving, although it reserves the possibility that digital computers and brains-as-computers may compute things in different ways. Research on computational models of consciousness has been driven by researchers like Hofstadter, Minsky, McCarthy, Dennet, Perlis, Sloman, and Cantwell Smith. A synthetic summary of their principal propositions is presented in [62], where the dominant proposition is that *"..consciousness is the property a computational system X has if X models itself as experiencing things."*. Thus, central to the theory of computational consciousness is that introspection is mediated by models.

As far back as in 1968, Minsky [64] introduced the concept of a 'self-model': *"To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A. A* is a good model of A, in B's view, to the extent that A*'s answers agree with those of A, on the whole, with respect to the questions important to B."*. This concept is semantically the same as the engineering definition of *model* defined by IEEE 610.12-1990, given in section 1.3 above. Most engineers know that a model of a system is an abstraction of the system which provides answers about desired properties and behavior of the system, with desired accuracy. So we see that a specific category of models (the self model) can be the theoretical basis for a meeting point between the more abstract reasoning of consciousness and the concerns of "everyday engineering".

The computing models based on results from cognitive theory and speculations on the structure of the human mind are explored in the field of *cognitive architectures*. There are a variety of cognitive architectures created to mimic specific aspects of human reasoning and decision making in machines. These include RCS, ACT-R, SOAR, CLARION, NARS, YMIR and others. A review of the prominent architectures (with further references) is presented in [73] which is based on actively maintained online resources[4]. The review mentions 54 architectures, out of which 26 are described, and observes that virtually all take their origin from biological inspiration, and different approaches are remarkably similar in their basic foundations. A comparative review of these architectures with specific relevance to machine autonomy is provided in [78], which draws some important conclusions of relevance to practicing engineers working on autonomous driving and other

safety critical systems. The comparison is made along four main themes of 'Realtime', 'Resource management', 'Learning' and 'Meta-learning'. The conclusions highlight a prevalent gap between cognitive architecture design and concrete operation in real-world settings. This gap is fueled by the observation that cognitive architectures generally tend to ignore realtime operation and resource management aspects. Ignoring such practical matters not only delays useful technological application, but likely leads to flawed theoretical foundations. In turn, this limits the usefulness of the architectures to toy problems, *"devoid of the complexity of the real world that human beings live and operate in."* [78]. A similar conclusion is drawn by other researchers exploring challenges in the domain of embedded systems [51]. Fortunately, these limitations are not shared uniformly by the cognitive architectures, promoting the possibility of designing architectures with a more complete set of cognitive functions and usable operational capabilities.

The current approach to autonomous driving capabilities is bottom-up: based on refinement of features starting from the automatic transmission and cruise control, to adaptive cruise control, lane departure indications, and autonomous emergency braking, to traffic jam assist and advanced driving assistance systems, eventually all converging on autonomous driving capabilities. This approach has yielded excellent results so far; results which may not have been reachable (in this timeframe) with a top-down approach that started inquiring into the nature and structure of (human) consciousness. The current approach has heavily adopted results from the robotics and AI domains. The adoption has exclusively involved careful, manual construction of systems, where learning and decision making takes place on the data/content or module levels. Such a hand-crafted approach is referred to as the *Constructionist Design Methodology*(CDM) in the AI domain [78, 79]. However, to create systems that approach human-level intelligence, significantly larger and more complex architectures are necessary. There is a very real danger that methodologies based on constructionist AI will prove inadequate ([78] states more strongly *"..are doomed"*) because of practical restrictions on complexity and size of software based systems designed and implemented by humans. These restrictions are captured in the term *cognitive complexity* [55], in the domain of embedded systems architectures. The cognitive complexity attribute of an architecture limits the ability of humans to hold the entire architecture in their head and reason about it. As cognitive complexity rises, it becomes increasingly more difficult (and costly) to verify and validate the systems, and assure properties like safety. Indeed, phenomena like *feature interaction* [63, 53] have already started occuring in automotive architectures (and generally, within cyber-physical systems) and their study and control is an emerging area of research.

Within the AI domain, a relatively new *constructivist* approach [77] has emerged, that advocates self-directed, introspective, learning and dynamically adapting conscious architectures instead of the 'carefully handcrafted' approach prevalent in the automotive domain. This is considered a paradigm

shift, whereunder the machine may proactively invoke stimulus-response cycles to continuously form and maintain self-models, and reason about their characteristics. The evolution of existing engineering approaches towards constructivist concepts requires developments in at least four relevant areas [75] (i) temporal grounding (ii) feedback loops (iii) pan-architectural pattern matching and (iv) small white-box components. Temporal grounding comprises of an awareness of time in the real world, as well as the time needed for the execution of software instructions, and how the two are corelated. Such an awareness is already important in the design of distributed real-time embedded systems, but as pointed out in [56], timeliness is a *semantic* property that is not well captured in popular programming paradigms. At a behavior level, temporal grounding involves predicting temporal result based on internal models, and updating the internal models if/when the predictions do not match the results. Traditional closed loop feedback is already prevalent in autonomous driving. These loops typically control short-time horizon actuation in response to the immediate sensing and perception of the environment. When moving towards constructivist AI, there need to be additional feedback loops that modify the control structures themselves, based on their perceived efficacy. This is closely related to the concept of self-modeling based on observations of stimulus-reaction experiments in a given context. Such modifications enable an expansion of existing skills and capabilities, which is an important characteristic of general purpose intelligence. Pan-architectural pattern matching is useful for the self induced comparison of temporal versions of the architecture, as the system grows/evolves over time in response to some specification. Pattern matching is also useful for identification of contexts and operational scenarios, which in turn is useful for controlling mechanisms of attention and recall. These mechanisms are important because they enable the machine to filter out the large number of stimuli present in complex operational situations, and focus only on those that have been learned as being relevant. Existing autonomous driving architectures are composed out of integration of large components like localization, trajectory planning, propulsion etc. These components are typically developed by different suppliers and are often "black-box" components with well-defined input/output interfaces. However their large size and black-box nature makes it difficult for a machine to reason about their internals based purely on an observation of the input/output signals. This is a critical issue for self-organizing and self-aware systems, because it is difficult to reach self-awareness in the presence of large, opaque internal components.

Constructivist AI certainly presents new challenges, some of which lie in the determinism and predictability of state evolution, internal beliefs, and actions of truly autonomous systems. These challenges currently run counter to requirements on provable safety, determinism, and assurance of other critical properties, and the constructivist approach is unlikely to be adopted for autonomous driving until it evolves sufficient tools, methods, reference

architecture patterns etc. to create practical and demonstrably safe and predictable systems.

1.5 An architecture for autonomous driving

In its current state, the tasks expected from an autonomous driving system are fairly well-constrained and specific. Therefore, it is possible to create domain specific reference architectures for autonomous driving. Such architectures would include definitions of the various architectural elements needed by an autonomous driving system, the hierarchy and data-flows between these elements and instantiation guidelines for specific use cases. In this section, we present a brief introduction to the required architectural components and some reasoning on their hierarchy and distribution. As with most domain specific reference architectures, we restrict the scope to the functional/logical views although some relevant technologies for instantiation are later described in section 1.7.

1.5.1 Main architectural components

The main functional components needed for autonomous driving are summarized in Figure 1.3. We have chosen to categorize these components into three categories

1. Perception of the external environment/context in which the vehicle operates
2. Decisions and control of the vehicle motion, with respect the external environment/context that is perceived
3. Vehicle platform manipulation which deals mostly with sensing and actuation of the Ego vehicle, with the intention of achieving desired motion

Each category has several components, whose functionality (from a strictly architectural perspective) will now be briefly described. A detailed description can be found in [24].

The **sensing** components are those that sense the states of the ego vehicle and the environment in which it operates. The **sensor fusion** component considers multiple sources of information to construct a hypothesis about the state of the environment. In addition to establishing confidence values for state variables, the sensor fusion component may also perform object association and tracking. The **localization** component is responsible for determining the location of the vehicle with respect to a global may, with needed accuracy. It may also aid the sensor fusion component to perform a

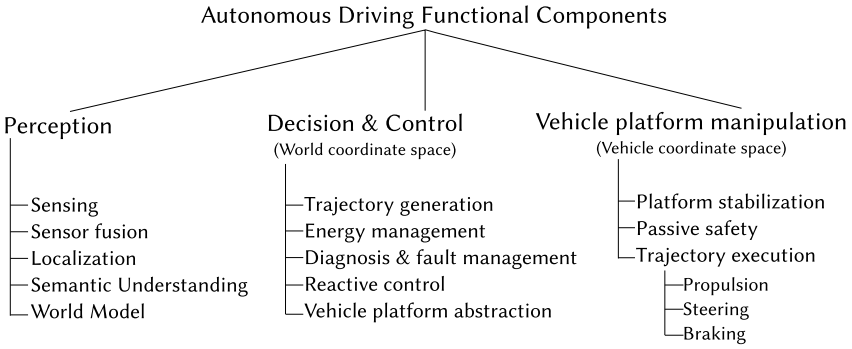

Fig. 1.3 Main components of an autonomous driving system

task known as *map matching*, wherein physical locations of detected objects are referenced to the map's coordinate system.

The **semantic understanding** component is the one in which the balance shifts from sensing to perception. More concretely, the semantic understanding component can include classifiers for detected objects, and it may annotate the objects with references to physical models that predict likely future behavior. Detection of ground planes, road geometries, representation of driveable areas may also happen in the semantic understanding component. In specific cases, the semantic understanding component may also use the ego vehicle data to continuously parameterize a model of the ego vehicle for purposes of motion control, error detection and potential degradation of functionality. This component comes closest to incorporating the 'self-model' needed for generating machine consciousness.

The **world model** component holds the state of the external (and possibly, internal) environment, as perceived by the ego vehicle. It can be characterized as either passive or active. A passive world model is more like a data store and may lack semantic understanding of the stored data. It can not, by itself, perform physics related computations on the data it contains, to actively predict the state of the world given specific inputs. The active world model, on the other hand, may incorporate kinematic and dynamic models of the objects it contains, and be able to evolve beliefs of the world states when given a sequence of inputs.

The **trajectory generation** component repeatedly generates obstacle free trajectories in the world coordinate system and picks an optimal trajectory from the set. The **energy management** component is usually split into closely-knit sub-components for battery management and regenerative braking. Since energy is a system wide concern, it is not uncommon for the energy managemnt component to have interfaces with other vehicular systems like HVAC, lights, chassis, and brakes. The **diagnosis and fault management** components identify the state of the overall system with respect to available capabilities, in order to influence redundancy management, system-

atic degradation of capabilities, etc. **Reactive control** components are used for immediate (or "reflex") responses to unanticipated stimuli from the environment. An example is automatic emergency braking (AEB). These components tyically execute in parallel with the nominal system, and if a threat is identified, their output overrides the nominal behavior requests.

The **vehicle platform abstraction** provides a minimal model of the vehicle, whose data is used to ensure that the trajectories being generated are compatible, optimal, and safe for the physics and capabilities of the actual vehicle.

The **platform stabilization** components are usually related to traction control, electronic stability programs, and anti-lock braking features. Their task is to keep the vehicle platform in a controllable state during operation. Unreasonable motion requests may be rejected or adapted to stay within the physical capabilities and safety envelope of the vehicle. The **trajectory execution** components are responsible for actually executing the trajectory generated by Decision and Control. This is achieved by a combination of longitudinal acceleration (propulsion), lateral acceleration (steering) and deceleration (braking). Most recent vehicles already incorporate such components and they may be considered "traditional" from the perspective of autonomous driving development.

1.5.2 A reference architecture

Having introduced the necessary functional components in the previous section, we now combine them into a suggested reference architecture, as shown in Figure 1.4, where the arrows show directed data-flows between the architectural elements.

The architecture is organized into three layers: the vehicle platform and cognitive driving intelligence which are on-board the vehicle, and an off-board or "cloud" based layer for potential tele-operation, remote monitoring and/or vehicle management. The off-board layer may be optional for many use cases, nevertheless it is almost always useful at least during the early phases of vehicle prototyping and testing. For heavy commercial trucks, some form of fleet management systems are usually provided to the fleet operators. Moreover, there are compelling drivers for including vehicle-to-infrastructure (V2I) communication to improve traffic efficiency and safety by sharing information acquired from multiple vehicles and other sources.

One of the key data flows in Figure 1.4 is between the cognitive driving intelligence and the vehicle platform layers. Functionally, this contains motion requests in the form of desired, instantaneous vehicle velocities, accelerations (longitudinal and lateral) and deceleration. These are typically in some absolute, global coordinate space, rather than being relative to the vehicle's motion. In practice, it may contain a short time series of these val-

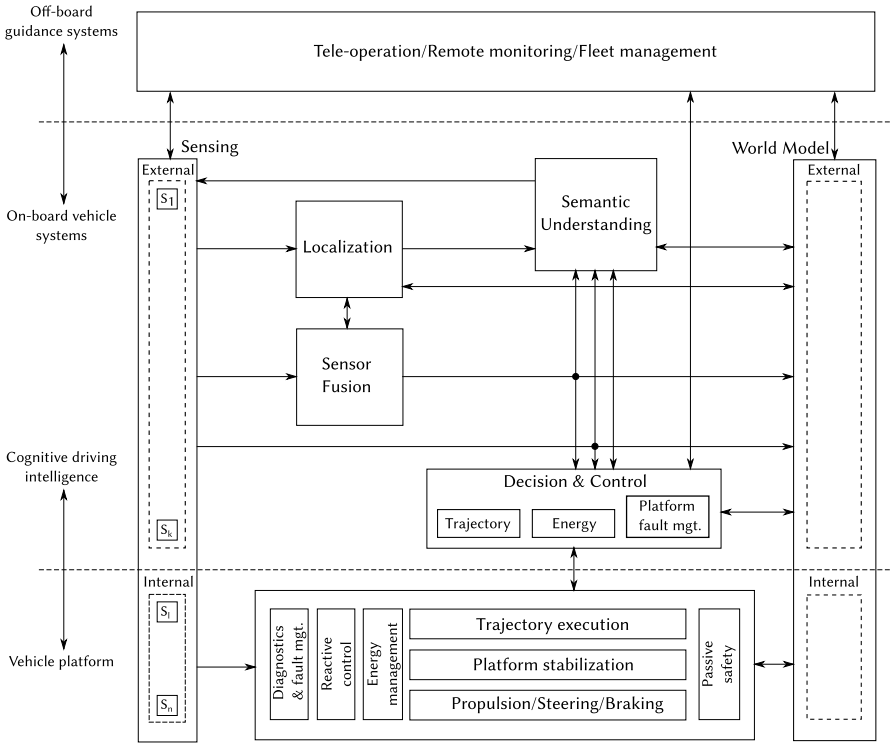

Fig. 1.4 A functional architecture for autonomous driving architecture

ues (trajectory fragments) rather than individual requests, because knowing the anticipated future setpoints is helpful for achieving more optimal control of the actuators. It is also feasible to include two different sets of trajectory fragments: one which takes the vehicle to the desired destination, and the other which takes it to a safe(er) state in case of system errors. The safe(er) state trajectory is computed periodically and should ideally be executable by the vehicle platform in an open loop fashion. The output of the Localization function contains at least the 2.5 dimensional vehicle pose consisting of the location in two dimensional space, as well as the heading. In practice, a lot more data and meta-data is provided which includes altitude, the latitude/longitude coordinates, the same information in a variety of coordinate spaces, detected known landmarks, number of satellites in the GPS fix, estimated accuracy etc. The output of the sensor fusion function depends heavily on the actual sensors and algorithms being used. Since lidars are used very often it is not uncommon to see point clouds with associated meta-information. The addition of a camera leads to the inclusion of extracted image features, and possibly colored point clouds. The latter are especially useful for classification in the semantic understanding component. The usage

of automotive grade radar is more interesting. Typically, automotive radar sensors come with an associated ECU which directly outputs (possibly classified) objects and their properties like relative velocity and distance. The radar outputs can be used for cross-correlation and plausibility testing of the data from the lidar and camera sensors. However, in projects with a strong emphasis on sensor fusion, the raw radar information may also be requested. It can be seen in Figure 1.4 that the output of a functional component goes to other components as well as the World Model. This is usually for performance reasons. In an ideal implementation, it would be possible to have a 'star' data topology where all the components exchange data only with the world model. This is a simplifying abstraction, leading to a single source of information and other benefits. However, it raises concerns on performance, usually latency, as well as increasing the extra-functional requirements on the world model, such as robustness, reliability and availability. The reference architecture permits both topologies and each instantiation may tune the amount of information each component receives from the world model, or directly from other components. The component interfaces are further refined in the technical architecture, considering the constraints imposed by the allocation of components to ECUs and the bandwidth of inter-component communications.

The on-board autonomous driving architecture admits a variety of distribution possibilities for the functional components. We choose to encourage a strong isolation between the vehicle platform and the rest of the driving intelligence. This isolation is beneficial from a number of perspectives. Firstly, from a legacy viewpoint, most automotive OEMs already have fairly sophisticated functionality in their existing vehicles for management of the vehicle motion. This includes features like (adaptive) cruise control, traction management, brake management including possible regeneration (for hybrid and electric vehicle platforms), and in the case of heavy commercial trucks, additional features like control of multiple axles, external brake requests, and overall powertrain control. One of the most convenient ways to introduce autonomous driving functionality is to introduce an additional system (the cognitive driving intelligence) which generates the kind of operational setpoints that the various vehicle controllers are already setup to receive. These setpoints are typically in the form of commanded acceleration, velocity, and vehicle deceleration. Secondly, even if legacy is not a concern, isolating the vehicle platform enables a clean separation of concerns and system partitioning. The cognitive driving intelligence needs to generate desired vehicle motion in some world coordinate system. Thus, its concern is to specify the global motion parameters which the vehicle platform should fulfill. To do this, the cognitive driving intelligence requires only a minimum model (abstraction) of the vehicle dynamics and the vehicle platform configuration parameters. Consequently, the entity responsible for answering the question, "Where and how should the vehicle move in the next N units of time?" need not have an intimate knowledge of the various vehicle propulsion mechanisms and

their continuous control. In turn, the vehicle platform is not required to have knowledge of how and why the motion requests are generated. Its responsibility is to fulfill the commanded motion requests while assuring the safety of the vehicle platform in terms of basic vehicle dynamics (limiting longitudinal and lateral accelerations, anti-lock braking, electronic stability control etc.). This sort of encapsulation of functionality and separation of concerns reduce the cognitive complexity of the architecture and are recommended best practices in the field of systems architecting. Finally, the isolation also facilitates product and platform variability management. Especially in the domain of heavy commercial vehicles, it is quite common to find extreme variability in each manufactured vehicle, since it is specifically configured for each customer's needs. By separating the cognitive driving intelligence from the vehicle platform, it can be reused on different vehicle platforms.

The off-board layer, depending on its functionality, needs to tap into differing parts of the on-board architecture. In our experience, the maximum amount of data exchange occurs with the World Model, since it holds practically all useful information needed by the off-board layer. This includes information regarding the current state of the on-board systems, the perceived external environment, as well as any upcoming motion decisions that may be in the execution pipeline. At the remote end, all received information is typically accumulated in a database, which in turn feeds application specific views of the gathered data. Active teleoperation [46] is foreseen in use-cases where a fleet of autonomous vehicles is overseen by a command-and-control center. In such use-cases, the vehicle may be able to "call home" when it gets stuck, or the remote center may actively claim control in potentially hazardous situations. In these cases, the tele-operation part of the system architecture needs to communicate with the Decision and Control part of the on-board systems. The commands sent are usually brief motion requests relative to the current location of the vehicle (in case the vehicle is stuck), or reprogrammed destinations. In our experience, the remote commands are directed at the cognitive intelligence and have relatively low bandwidth requirements. Direct control of the components in the vehicle platform requires significantly higher bandwidth and stricter timing constraints, which is rarely possible over large distances with existing wireless communication technologies.

1.5.2.1 Comparison with similar architectures

Comparisons of the reference architecture can be made with the architectures of Junior - Stanford's entry in the 2007 DARPA Urban Driving Challenge, the HAVE-IT project, and a Mercedes Benz autonomous car. These architectures are relevant because they represent a steady improvement of functionality and implementation, over the past decade. Junior is a successful example of a self-driving vehicle from the early days of the technology, and a largely academic

proof-of-concept. The HAVE-IT project consortium had strong representations from OEMs and Tier 1 suppliers from the automotive domain, as well as independent research institutes and universities - the project focused on highly automated driving and advanced driver assistance systems. The Mercedes Benz autonomous car development had the automotive OEM Daimler AG as the majority stakeholder. The intent of the comparison is to highlight similarities and differences, rather than make claims of which architecture is "better". An architecture needs to be evaluated in its context, because the context imposes unique constraints with associated implications on the design. Thus, we choose to believe that every architecture that works has merits in its own context, and that there is rarely a definitively best solution to any given architectural problem.

Stanford University's DARPA Urban Challenge entry, Junior [65], provides an early example of an autonomous driving architecture. The interface to the VW Passat vehicle is via steering/throttle/brake controls, rather than direct longitudinal and lateral acceleration demands. This can probably be explained by the assumption that the autonomous driving architecture was designed exclusively for tight integration with one particular vehicle, which lacked general vehicle dynamics interfaces for setting acceleration and deceleration setpoints. The architecture is divided into five distinct parts for sensor interface, perception, navigation, user interface, and the vehicle interface. The localization is integrated into the perception part, and there seems to be no effort to classify detected obstacles. This architecture also explicitly includes a component/layer for 'Global Services' dealing with functionality like file systems, and inter-process communication. We do not describe these services because they do not strictly fit into an architecture's functional view. The architecture is not strictly divided into layers, nor is there an explicit component to abstract the view of vehicle platform.

The layered approach to architectures and their description is also found in the European HAVE-IT project [15], which had its final demonstrations in June 2011. This project architecture consists of four layers: 'Driver interface', 'Perception', 'Command' and 'Execution'. The Perception layer consists of environmental and vehicle sensors, and sensor data fusion. There is no mention of localization, perhaps because the system operates in close conjunction with a human driver. The Command layer contains a component named 'Co-Pilot', which receives the sensor fusion data and generates a candidate trajectory. A 'mode selection' component in the Command layer then switches between the human driver and the 'Co-Pilot' as a source of the trajectory to be executed. The selected trajectory is then handed to the Execution layer in the form of a motion control vector. The Execution layer consists of the Drivetrain control, which in turn controls the steering, brakes, engine, and gearbox. This execution layer corresponds closely to our vehicle platform layer in that it pertains to drivetrain control and *"..to perform the safe motion control vector."* [15]. Also similar is the usage of a motion control vector as an interface to the vehicle platform/execution layer. Our architecture ad-

ditionally incorporates energy management as an explicit part of the decision and control component, which is especially valuable for electric and hybrid drivetrains, since then considerations of estimated range can be incorporated in the long term trajectory planning. The HAVE-IT architecture evolved in the context of Advanced Driver Assistance Systems (ADAS) with a strong reliance on the human driver and emphasis on driver state assessment components in the command layer; it remains unclear how well it can be adapted to L4 autonomous systems, where a human driver may not be present.

Close comparisons can be made with the architectural components of Bertha, the Mercedes Benz S-class vehicle, that recently(2014) completed a 103 mile autonomous drive from Mannheim to Pforzheim [86]. In the system overview presented in [86], components like perception, localization, motion planning, and trajectory control are clearly identified. These agree well with the components we have described in this paper, however this is hardly surprising. Every autonomous driving system requires these functional components and they are likely to show up in practically every architecture for autonomous driving. The system overview in [86] does not explicitly acknowledge the existence of components for semantic understanding, world modeling, energy management, diagnostics and fault management, and platform stabilization. It is possible that some or all of these were present, but not mentioned. This is especially true for diagnostics and fault management. A part of semantic understanding, related to classification of detected objects can (and often is) put in the Perception component, as in [86], but we see benefits in the explicit separation of sensor fusion and semantic understanding advocated by our architecture. The isolation of semantic understanding from raw sensor fusion enables faster and more independent iterations and testing of newer algorithms, without affecting the rest of the system. This is directly relevant to the engineering stakeholder concerns of independent development of individual subsystems, as well as their virtualized simulation and testing. Further, the raw object data from sensor fusion is still of value to the Decision and Control components, despite a lack of accompanying semantic understanding. This is because, although knowledge of whether a detected object is a pedestrian or motorcycle is useful for optimized path planning, collision with the object still needs to be avoided regardless of its classification. In a similar vein, incorporating a distinct component for world modeling, enables incremental sophistication in internal representations while retaining (backwards compatible) interfaces. The existence of a distinct world model components makes it easier to answer questions like, "How will the world evolve if I perform action X instead of action Y?" Although [86] mentions the existence of a 'Reactive Layer', it makes no mention of any other layers in the architecture and how the components are distributed across them. In our paper, we make a clear distinction between the vehicle platform and cognitive driving intelligence layers and provide a rationale for our proposed component distribution.

Comparison of these and a few other architectures with our proposed architecture leads us to believe that the explicit recognition of semantic understanding, world model, and vehicle platform abstraction components are unique to our architecture. This is not entirely coincidental, since our incorporation of these components is, to some extent, a deliberate action to resolve the short-comings we perceived during our early state of the art surveys. Furthermore, our architecture has been applied to a larger variety of vehicles (commercial trucks, passenger cars, as well as novel, legacy free designs) and therefore necessarily incorporates features related to greater isolation of functionality into distinct components and abstraction of vehicle interfaces. The aggressive partitioning of architectural components provides significant freedom to the component developers to modify and test new algorithms, without affecting the rest of the system. It also reduces the cognitive complexity of the system, and makes it relatively easy to foresee potential pitfalls and debug causes of objectionable behavior.

1.6 Systems Engineering

Systems engineering concerns cover the methodology and artifacts related to the engineering processes used throughout the development of the system. Ideally, these concerns commence with an investigation of the needs which the system intends to address, and then cover the entire development lifecycle until the system is deployed. Beyond development, systems engineering also looks at aspects related to the system's operation, maintenance, potential upgrades and eventual decommissioning. In this chapter we restrict the scope to the development activities only. Within this restricted scope, we discuss the modeling steps and associated classes of models that are important for the development of autonomous driving systems. They are important not because the nature of autonomy directly necessitates an increased focus on systems engineering, but because autonomy requirements increase the complexity of the system being designed. The complexity, together with the safety critical nature of the system, requires careful attention to all aspects of the development processes, which is facilitated by systems engineering and its associated methodologies, tools and artifacts. Thus, the topics covered in this section are *not exclusive* to autonomy, but they have a significantly increased importance within the context of autonomy.

In model based systems engineering (MBSE), the engineering processes are supported by an increasing set of models, some of which become increasingly detailed. To guide the systems engineering process, there exist a number of lifecycle development models and engineering methodologies. Most lifecycle models are grounded in one of three seminal models [42]: The Waterfall model [71], the Spiral model [27], and the V-model [47, 48]. Of these, the V-model and its variations have been extensively applied to systems engineering and

development. These lifecycle models are leveraged by various MBSE methodologies, such as OOSEM, RUP-SE, Harmony-SE, JPL State Analysis etc. An overview of these prominent methodologies is given in [42], which includes further information, including additional references for each methodology. More methodologies have since been identified and they are gathered and described online at the MBSE Wiki Page [16]. Beyond these relatively general methodologies, there are also approaches to specific parts of systems engineering that are more focused on embedded and automotive systems. These deal with topics like requirements engineering [31], formal semantics [35], multi-view modeling [34] etc. Among other things, the methodologies define a number of development activities. Example activities are the definitions of stakeholder needs, system requirements, logical architecture, and system validation and verification. In some methodologies, the execution of these activities may be referred to as 'phases'. Each methodology involves a slightly different grouping and sequencing of activities and phases. The MBSE methodologies may or may not recommend a specific language or tool framework, however, they all involve the creation of models or model views supporting each engineering phase. In this section, we propose four classes of models, the contents of which are independent of methodology, but which can be mapped to different activities within a selected MBSE methodology. Similarly, a project specific lightweight MBSE methodology may be created by selecting specific models from each class and specifying the sequence(s) of their creation using particular tools. Thinking in terms of model classes is valuable because classes easily encapsulate a diversity of modeling formalisms and tools. The diversity is necessitated, at the very least, due to the state of practice in modeling technologies - at the moment, there exists no comprehensive modeling formalism that can completely capture all relevant systems engineering stakeholder concerns. Therefore, the various concerns need to be captured using heterogeneous sets of possibly domain specific models.

Establishing and maintaining links between all the assorted models is an active area of research [80], but in the meantime, project specific inter-model links need to be selected manually by the practicing systems engineer, and these need also need to be manually maintained by the engineering team. The links represent relations like refinement, allocations, correspondence etc. between the model elements. An active risk to be guarded against is that in fast moving projects systems engineering models are assigned secondary importance. Therefore, either models are not created, or the models and links are not updated as the implementation changes. Over time, there is significant divergence between the implementations and the models. This leads to accumulation of systems engineering "debt". The debt accumulation can be partially mitigated by via two interrelated choices: the methodology, and the tools used for the systems engineering process. Flexibility in the methodology and automation support from the tooling reduce the "cost" or effort of model creation and maintenance, making it more likely that the

engineering team views systems engineering as less of a "burden" and more of a benefit.

Below, we discuss some key steps in systems modeling, sequenced as they would occur in an ideal development processes. Realistically, it is very important that the methodology and associated tools permit the engineering team to begin with modeling multiple, arbitrary concerns, which can later be extended and linked to other models in all directions. Thus it should be possible to start modeling the logical architecture of the system, or a software component, without having explicitly modeled the preceding requirements, actors and their interactions. Such knowledge may be implicitly known/assumed by an engineer who is itching to sketch out an architectural solution - forcing her/him to explicitly model requirements and behavior first merely results in annoyance and rebellion.

The main systems engineering model classes that support the steps we describe are shown in Figure 1.5. It should be noted that there may exist a variety of different models within each model class. The first step is to analyze and represent the system's users, their operational needs, and the usage scenarios. Who are the actors involved in interacting with the system? Actors can be human users as well as other entities involved in autonomous driving like other road users, road infrastructure like traffic lights, etc. Once the actors are defined, the next task is to determine what they want the system to do. What behavior and services do they expect from the system? What are the contexts within which the actors will interact with the system? What will be the modes of interaction, and how are they related to the operational contexts? Depending on the modeling languages and tools used, there can be a variety of diagrams, process definitions, allocation of actors to process artifacts etc. for capturing and representing the ideas at this phase. This phase helps to identify the principal behavioral requirements expected from the system and its usage by the actors involved.

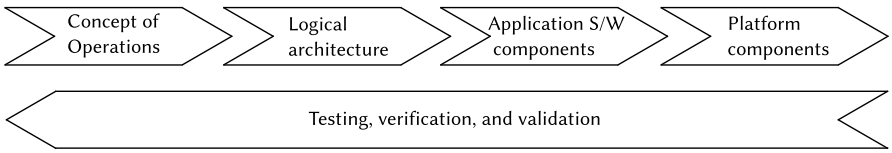

Fig. 1.5 Systems engineering model classes

The second step is to identify the main system components, their contents, relationships, hierarchies, properties, and behavior. This constitutes the logical or functional architecture of the system and it excludes concerns related to technological implementation of the identified components. However, it does take into consideration all major extra-functional constraints (like safety, security, performance, reliability, etc.) so as to find a suitable compromise between them (to the extent this is possible without getting into

implementation specific detail). The logical architecture defines the components and their interfaces, including formalization of all logical views and how these views are accounted for in the component designs. The behavior requirements from the previous step are refined and allocated to the identified architectural components. Links between requirements, operational scenarios, actors, and components are also established. The entire process of logical architecture can be applied repeatedly within the boundaries of a single logical component or its sub-components. Thus, the logical architecture may have multiple and differing "levels of zoom" for each of the components and their connections. Depending on the tools used, the logical architecture and its properties may be represented using a multitude of models, modeling tools and languages, or with a unified, all encompassing model, within which various related concerns are represented using specific views. Given a logical architecture including behavioral views, one could use tools for static analysis (e.g. checking interface compatibility) and perform behavioral analyses. These may include simulation and model checking to assure specific system properties like absence of deadlocks, reachability of specific states etc. The specific analysis techniques of interest may very well dictate the choice of architecture representation languages. For example, it is not uncommon that a modeled architecture or its subcomponent needs to be re-modeled using a different language, just so that it can be verified by using a particular model checker.

The third step is the modeling and grouping of application software components within the system. The grouping may be in terms of individual software applications, or for the purpose of simulation in tools like Simulink. These models include representations of the behavior of the software components, their resource requirements, runtime characteristics, interfaces, properties and attributes, and communication specifications. At this stage it is increasingly common to see the usage of so called "executable software models". Executable software models are those which can directly be transformed into compilable source code, with guarantees that when the source code is executed under assumed conditions, it should have the same behavior as the modeled software component. Dependencies on expected platform capabilities are usually explicitly mentioned for each software component model.

The final step is the modeling of the platforms on which the application software components execute. The platforms consist of a 'stack' incorporating the computing silicon (micro-processors/controllers), an optional operating system which may or may not provide realtime guarantees or alternatively, a native language runtime for the chosen silicon, an optional middleware that abstracts the operating system and its services, and any libraries, daemons and other services provided by the operating system and hardware. Multiple application software components may execute on the same or similar platforms, and in cases of advanced experimental architectures, application software components may migrate between similar platforms. In all cases, it is more convenient and useful to model a platform and its services as a whole,

rather than embed this information into each application software component. However, we emphasize that this is largely a matter of preference.

Associated with all the four modeling steps is a continuous refinement of requirements, test cases, safety viewpoints, and documentation artifacts. A comprehensive taxonomy of models associated with each step is beyond the scope of this chapter, but each step must introduce additional models representing requirements and test cases relevant to that step. Requirements must be allocated to models that assure them, and both requirements and test cases need to be assigned to unique members of the engineering team. Safety considerations are usually generated by following processes established by safety standards like ISO26262. Additional models/views like functional safety architecture will be introduced, along with their refinements to technical safety architectures and associated redundancies and switching modes for application software and platform components. The number of models may grow and shrink at each step as the systems engineering process is iteratively applied during system development. Ultimately, the artifacts ideally exist as a web of interconnected models at each step and across the steps. Preserving the links between the models, and keeping the models up to date with the implementation is a significant challenge. One way to do this is via increasing toolchain automation, but given the relative lack of production ready tools, it becomes the responsibility of the architecting and systems engineering team to select and minimize the number of models used to represent the system. This in turn, depends on a variety of technical and non-technical factors like the nature of the project and its maturity level, available tools, the importance attached to systems engineering by project management, the skills and qualifications of the people involved etc. One recommendation based on our experience is to always synchronize the software and platform models with the actual technologies being utilized in the project. These will change as the project moves through stages of proofs-of-concept, prototyping, to certifiable implementations. The models need to be updated correspondingly. For example, during the early prototyping phase, if the entire application software is modeled in Simulink and the execution platform is a rapid prototyping system like the dSpace MicroAutoBox, it makes little sense to model the system in terms of software threads, tasks, operating system runtimes etc. because the modeling concepts and technical implementation concepts just do not synchronize in terms of relevance. Here, it is better to restrict the models to Simulink and coarse dSpace specifications. When an eventual move is made to AUTOSAR or similar infrastructure, the associated models dealing with finer platform details may be created.

One of the conflicts that MBSE can help resolve is the need for fast development cycles, while guaranteeing consistency of the various process and product artifacts. In our opinion, this necessarily requires the support of advanced tooling and as such, the details are highly tool specific. But the general underlying patterns involve inter-model links, co-simulations, and model checkers. The inter-model links typically represent relationships be-

tween the linked models, for example, B "realizes" A or Q "is derived from" P etc. An example of this is a facility offered by several commercial toolsuits wherein requirements can be linked to specific blocks in a Simulink diagram. When either of the models changes (the requirement or the Simulink block), a flag is raised to signify that the status of the link requires investigation. Depending on the tool support, additional data on what has changed may also be presented. Even a simple facility such as this helps to prevent undetected changes from propagating through the design. Prior to a release, the MBSE process may require that no inter-model links have unaddressed 'change flags'. Similarly, with the advent of technologies like Open Services for Lifecycle Collaboration (OSLC) [9], the concept of "round-trip" flows is gaining ground. In a round-trip, a model created using some particular modeling technology can be "exported" to a different tool, where it is enriched or analysed, and the results can be sent back to the original tool. Techniques like OSLC or Functional Mockup Interface (FMI) [6] enable model exchange and co-simulation of heterogenous models. Such co-simulations can be used to simulate an entire vehicle, comprised of different types of linked models. Thus, it becomes possible to easily examine the impact of changes deep within one particular model, on the overall vehicle behavior. For individual models, the usage of model checkers can help assure that desired properties are retained (for example, absence of deadlocks and unreachable states) after model modification.

Based on the modeling steps described, and the classes shown in Figure 1.5, a partial view of modeling artifacts and their allocation links is shown in Figure 1.6, where the rightmost column describes some of the functionality provided by the models. The dotted, curved arrows crossing the vertical layers represent allocation links. Thus arrows between the architecture representation and application software components show the mapping between logical architecture elements and particular application software components. For the sake of clarity, the continuous refinement and accumulation of requirements and tests is not shown in the diagram. Examples for specification, structuring, and refinement of requirements can be found in [85, 84]. The tests description must include the tools and methods of conducting the tests, templates for data logging during testing, component behaviors and data values/ranges that constitute an acceptable test result. Figure 1.6 represents a concise takeaway of the contents of this section on systems engineering.

1.7 Technical implementation

In this section, we briefly discuss some selected technologies that have proven useful in our experiments for implementing autonomous driving systems. The emphasis is on the early prototyping phase, since that is usually when it is possible to experiment with novel technologies in a low risk manner. Similar

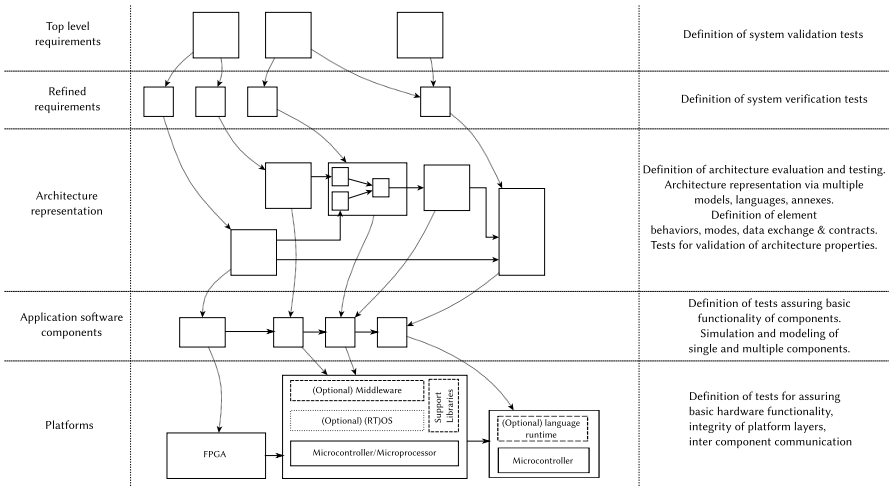

Fig. 1.6 Partial view of modeling artifacts and allocation links

to Section 1.6, the topics and technologies covered in this section are *not exclusive* to autonomy, but they have a magnified importance in the context of the development of autonomous driving systems, which makes them worthy for consideration.

Since the last 15 years, there has been a proliferation of various Architecture Description Languages (ADLs) in academia and industry. A systematic overview is provided in [52], which discusses 102 ADLs with 33 from Industry and 69 in Academia. Each has its own specific meta-model, notations, tools, and domain applicability, with little chance of interoperability between them. Moreover, the ADLs themselves have often undergone complete changes or remodeling between versions. Capturing the vast variety of stakeholder concerns within a single notation is exceedingly impractical, as is the aim of creating a "universal notation". Consequently, domain specific ADLs have emerged, which focus on the properties of a particular domain, and specific types of analyses and modeling environments [59]. When different concerns need to be modeled in differing languages, the individual models need to be synchronized, such that changes in one model are propagated and reflected in the others. There are efforts for tool based and automated synchronization but practically, this is still a manual process. Thus, domain specific ADLs need to strike a sweet spot where they are expressive enough to model all relevant stakeholder concerns in the domain, while minimizing syntax and remaining usable. This needs to be complemented with excellent tools, preferably those which also enable bi-directional synchronization not just between different models, but also between models and executable implementations. This is a steep challenge. We have identified three candidates that are broad and deep enough to address a non-trivial number (but not all!) of stake-

holder concerns, in the field of automated driving: EAST-ADL2 [40], AADL [43], and ARCADIA [12]. Of these three, AADL and ARCADIA are generally applicable within the domain of embedded systems architecture, while EAST-ADL2 is specifically intended for automotive systems. All three enable the representation of requirements, behavior, structure, mapping to technical implementations as well as tracebility links among them. Both EAST-ADL2 and AADL have reference tooling to import Simulink models to form architecture representations. This is important, because Simulink is the de facto simulation and modeling tool in the automotive industry. EAST-ADL2 is supported by a language specific core methodology [2] which can be complemented by additional extensions related to requirements traceability [21], formal analysis and verification [54] etc. Similarly, the ARCADIA method is embedded strongly in the recently open-sourced tool Capella [14]. In our opinion, Capella is the most mature, comprehensive, and user-friendly open-source tool available for model based systems engineering at the time of this writing. It is built on the Eclipse ecosystem, and it been in use internally at the Thales company for over five years. The tool guides the user through a series of top-down modeling activities defined by the ARCADIA method. It also offers a series of default viewpoints for modeling, and within each viewpoint modeling can be done by means of a number of diagram types. Definition of custom viewpoints is also possible within Capella. Note that although all these tools have the word 'architecture' in their names, their actual functionality goes beyond traditional architecture and into the various systems engineering modeling activities we have covered.

An important consideration in the selection of tools and languages used for systems engineering and other modeling, is the support they have for the platforms and programming languages used in the system implementation. Automotive domain specific languages like EAST-ADL2 have built-in support for AUTOSAR concepts, enabling an easier mapping from the models to their implementation platforms.

For technical implementation of autonomous driving systems, we propose a setup represented in Figure 1.7. The setup consists of aggregations of components, connected by a publish-subscribe bus with Quality of Service (QoS) filters. The aggregations are represented by the boxes in Figure 1.7 and agree well with the architectural layers presented in Section 1.5. The overall intent is to support development of functionality in simulations and on target implementations, with the ability for seamless transitions. Even when certain functionality has been implemented on target hardware, it is still valuable to switch back-and-forth between simulations and propagate changes in both directions. The setup consists of a physical platform (a.k.a drivetrain) along with a simulation of it running on a general purpose computer (the "soft" simulation) and a version running on a hardware-in-the-loop (HIL) rig. Similarly, the various drivetrain controllers exist as soft simulations and implemented on target hardware. The cognitive driving intelligence layer is aggregated in an implementation, in addition to a synthetic 3D environment

setup for example, within a 3D gaming engine like Unreal [19]. Within the synthetic environment, techniques like ray-tracing are used to emulate radar, lidar, and camera sensors. The environment incorporates physics engines for calculations of solid body dynamics, collisions etc. The setup allows for combinations like driving the real, physical vehicle in a real environment, driving a virtual vehicle in virtual environment using the real, implemented driving intelligence, testing the implemented drivetrain controllers on a simulated drivetrain, etc. Of course, depending on the constraints and resources of a particular project, not all aggregations need to be present.

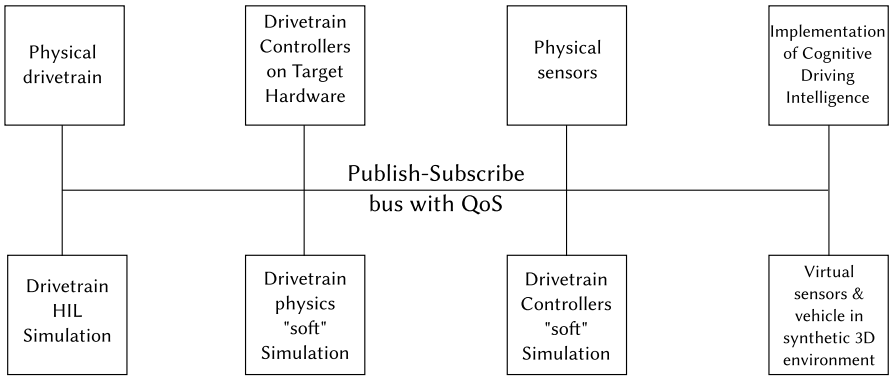

Fig. 1.7 Proposed implementation setup for autonomous driving

The technical implementation platforms for the cognitive driving intelligence and the virtual environments are typically (during prototyping) commercial off-the-shelf (COTS), general purpose, multi-core computers with best-of-breed COTS graphics processing units (GPUs). GPU vendors also provide a variety of graphics and parallel processing libraries to leverage the capabilities of their platforms. The computers usually execute the Linux operating system, with middleware like ROS [70] or OROCOS [36]. Of these, ROS is more popular, with a large, open-source community. It provides facilities for computation and communication across processes running on the same and different computers. ROS does not support hard realtime operation, but for prototyping this is usually not a limitation, provided the hardware has sufficiently fast processors and enough memory. Hard realtime requirements are usually more critical for the drivetrain control, compared to the cognitive driving intelligence. However, Linux does provide some hardened timing guarantees with the help of the `PREEMPT_RT` kernel patch [18] or dual kernel approaches like Xenomai [20]. The OROCOS middleware provides a component framework that can leverage the hardened realtime properties and it also supports inter-component communication on the same and different computers. The programming language of choice for the cognitive driving intelligence is usually C++, a choice often mandated by the used middleware and useful

libraries[2] like OpenCV [30] and the Point Cloud Library (PCL) [72]. The usage of Java for high level tasks is not uncommon either, although there is a lingering perception that it suffers from performance problems in comparison with C++. A modern alternative to Java is the Scala programming language [67], which in our opinion, deserves greater interest. Scala allows for strict functional programming, object-oriented procedural programming, as well as a free combination of the two. It runs on top of the Java Virtual Machine (JVM) enabling it to leverage existing Java libraries. The functional programming paradigm emphasizes stateless, side-effect free functions as building blocks, which leads to greater ease of side-effect free composition of functionality.

The "soft simulations" of the drivetrain and its controllers are usually made in Simulink, typically running on the Windows operating system on general purpose COTS computers. The reason for choosing Windows here is that these simulations are often executed on rapid prototyping systems like the Simulink RealTime and dSpace autobox, the tooling for which is often Windows-only.

There is a greater variety of platforms for the drivetrain controllers, where constraints of realtime, scheduling, input-output, and computing resources are more prominent. Relevant operating system standards here are OSEK/VDX [17] , parts of which are standardized in ISO 17356, and AUTOSAR [13] , which reuses large parts of OSEK. There are a large and diverse number of vendors supplying customized real time operating systems, and middleware stacks with varying capabilities. The programming languages of choice are usually C or C++, where large portions of the code is autogenerated from tools like Simulink. For both languages, safety critical subsets are defined by the Motor Industry Software Reliability Association (MISRA), in the form of approved language usage rules and there are tools to generate and verify C/C++ code against the MISRA rules. In our recent projects, the usage of the Ada 2012 programming language is receiving greater attention. This is partially because of its reputation as the programming language of choice in other safety-critical domains, as well as its syntactic support for notions of contract based programming in the 2012 version. The latest incarnation of the safety critical subset of Ada, SPARK 2014, also deserves rising interest, in our opinion, not least because of its accompanying tooling for code analysis, coverage and testing. Ada provides a number of runtime profiles, including a multi-tasking profile named Ravenscar [37]. The language then provides built-in support for tasking and inter-task communication, often negating the need for an explicit operating system on embedded microcontrollers. The dominant embedded microcontrollers in this domain are based on the ARM platform, which provides both uni- and multi- core processors in 32 and 64 bit configurations. For drivetrain controllers, the matching of the hardware platform with the operating system (OS), and the OS support for

[2] Although some libraries do provide bindings for other languages, in our experience C++ still dominates the scene.

low-level hardware and peripheral drivers is a more important consideration than for the platforms used for the cognitive driving intelligence, since there is less standardization in this area.

Communication middleware has not made significant inroads in the automotive domain, beyond the functionality provided by AUTOSAR with its Virtual Function Bus (VFB) concept. At a lower level, the CAN bus remains the most common technology for connecting distributed vehicle controllers. Autonomous driving systems impose greater bandwidth requirements on the in-vehicle communication links, which is especially true when high resolution sensors like cameras and multi-beam lidars are utilized. The usage of gigabit Ethernet is exceedingly common in autonomous driving projects, especially for connecting the computers executing the cognitive driving intelligence. Considering that these computers use general purpose operating systems and programming libraries, they are able to utilize advanced communication middleware, beyond bare TCP/IP or UDP/IP socket communication in client-server scenarios. Three facilities provided by advanced communication middleware are: automatic de/serialization of data structures into wire representation, high level functions for sending and receiving the data using fine-grained Quality of Service (QoS) guarantees, and automatic routing and service discovery within the network. The middleware uses either the brokered or broker-less architectures and often uses uniform APIs regardless of whether the communication is between threads in a single process, between multiple processes on the same computer or across computers. It also makes available smarter communication patterns like publish-subscribe, push-pull, N-to-M, fan-in, fan-out etc. A relatively recent communication middleware standard becoming increasingly common for critical, distributed realtime systems is the Object Management Group's Data Distribution Service (OMG-DDS) [68]. DDS supports message structure definition using an Interface Definition Language (IDL) similar to CORBA, and thenceforth manages all aspects of data transmission and wire-representation with a large number of QoS settings, using the publish-subscribe pattern. Another efficient communication middleware is ZeroMQ [5], which provides no built-in support for data de/serialization (it transmits given 'binary blobs'), but supports substantially more patterns than just publish-subscribe. ZeroMQ also provides bindings for over 40 programming languages.

1.8 Discussion

In this chapter, we have touched upon some relevant topics on the nature of autonomy, architectures and systems engineering aspects for autonomous driving, as well as implementation technologies for prototype systems. This section presents a synthetic discussion highlighting how these areas affect each other, as well as the how autonomy influences each of them.

1.8.1 A holistic view

A holistic view, relevant for both long term perspectives as well as contemporary engineering is shown in Figure 1.8. The Figure shows that each of Concepts, Architecture, Systems Engineering and Technical Implementation can be divided into two parts. One part is for the relatively low level concepts of contemporary engineering, and the other is for the higher level concepts arising from theories of general autonomous and intelligent systems. The former part is shown in the inner circle of Figure 1.8, while the latter part is shown in the outer circle. Thus, the basic functional components for the architecture, as described in Section 1.5.1, and their inter-connections form part of the inner architecture circle. The higher level concepts like runtime system level reasoning, which would be useful to have, but are currently not included in most contemporary architectures, are shown in the outer architecture circle. Figure 1.8 also shows that the concepts and systems engineering areas drive developments of architectures and technical implementations, for both high and low level concepts. The state of the art in autonomous driving still lies mostly within the inner circle, with very brief and occasional excursions to the higher levels.

The construction of autonomous driving systems today very much follows the Constructionist Design Methodology(CDM) which involves integration of a large number of functionalities that must be carefully coordinated to achieve coherent system behavior. For autonomous driving, the coordination is manual, and notions of coherency are mostly implicit in the heads of the designers. This approach limits how well systems scale with rising system complexity. These limits are already acknowledged in research on AI systems integration [76], which is exploring a fundamental shift from manually designed to self-organizing architectures that can learn and grow [75] - the so-called constructivist approach. However, given the safety, reliability, determinism and real-time requirements of autonomous driving, it is not feasible to abandon the constructionist approach entirely. Therefore, it is necessary to reason on the gap between the two approaches to determine the direction in which autonomous driving architectures need to evolve. One way to explore the gap is to think in terms of 'missing components' and how they can be injected into the existing architecture patters without breaking them. A key missing component is an explicit representation of a 'Self' or 'Ego' within the vehicle, along with a 'Goals' module. Today, the automobile architecture is made up of a variety of basic sub-systems like the engine, the transmission, the brakes, steering etc. and some hierarchically higher sub-systems related to traction control, traffic jam assist and so on. These systems have limited knowledge of the existence, purpose and functioning of the other sub-systems (following the excellent architectural principal of 'separation of concerns') and consequently, in the presence of severe uncertainty or disturbances within the operational environment or due to internal failures, there is no particular sub-system that can perform system-wide reasoning, revise

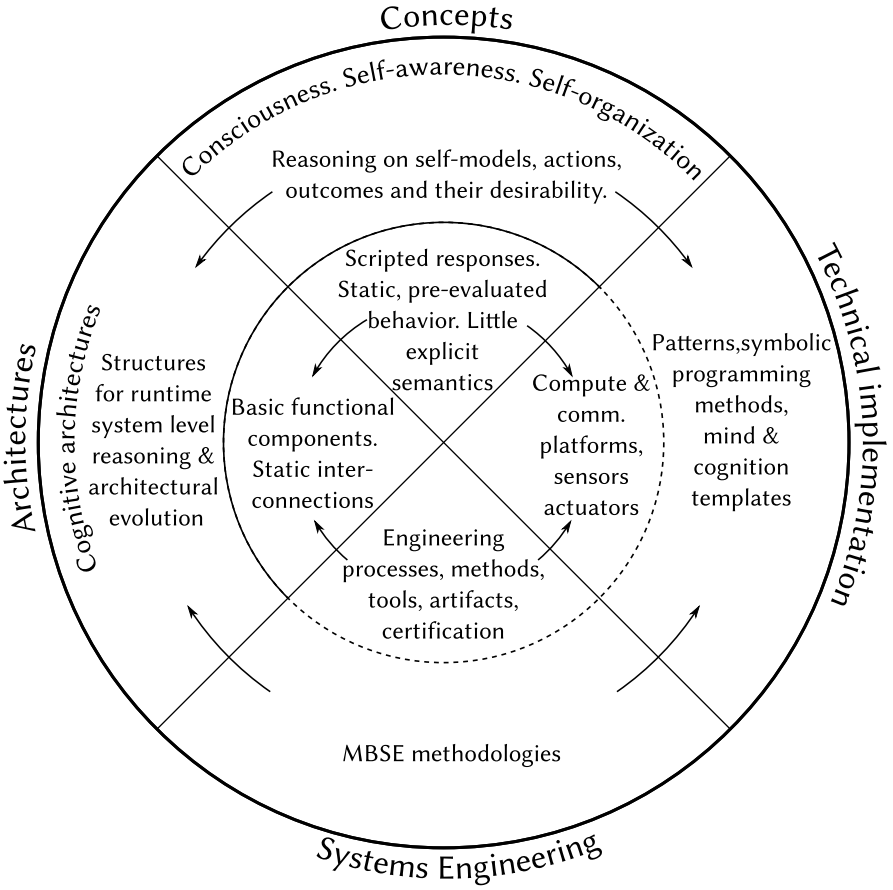

Fig. 1.8 A holistic view of the concepts covered in this chapter

previously held beliefs about the system capabilities and its environment, and act/adapt accordingly. When the operational situation exceeds the limits envisioned by the system's human designers, or when the carefully scripted responses fail, the system as a whole also fails. The presence of explicit 'Ego' sub-systems could be a first step in addressing this problem. The 'Ego' would be aware of the primary behaviors the vehicle is expected to fulfill, as well as the various subsystems present and how they work together to generate the desired behavior. The architecture of the 'Ego' is an interesting topic in itself: it could range from a "thick central" ego component, to a hierarchical network of ego components in each subsystems. The self-awareness would be continuously represented and maintained in the form of a self-model. The set of self-models can be 'seeded' at the time of the architecture deployment and the 'Ego' would be free to evolve them and synthesize other models throughout vehicle operation. Preliminary methods for automatic synthesis

of multiple internal models already exist (see for example [28]) and have been proven to successfully increase machine resilience [29]. This directly contributes to characteristics like fail-operational behavior and survivability, which are desirable in autonomous driving. Initially, the operation of the Ego sub-system would be limited to (de)activation of particular subsystems, dynamic reconfigurations of the connections between the systems, transfer of functional responsibility between the sub-systems, and triggering specific modes in individual subsystems. With today's technological means, all permissible system configurations need to be proven for coherence, consistency and safety in advance. The eventual transition to constructivist architectures creates the possibility of runtime reasoning in the Ego components, leading to runtime verification of the desired properties. The AI domain is already reporting some progress towards agents, design methodologies and programming paradigms following constructivist approaches [39, 69, 66]. We feel that this is a good time to engage with researchers in the domain of AI systems integration, in order to highlight the requirements relevant to practical embedded systems. This is intended to prevent a repetition of the problem seen in the area of cognitive architectures, wherein real world concerns are rarely accounted for in theoretical development [78].

As noted in the outer circle of Figure 1.8, there have been efforts to create higher level concepts like consciousness and cognitive architectures, that mimic some aspects of human decision making. However, corresponding developments in systems engineering are conspicuously missing. We anticipate that an expansion to the scope of existing systems engineering to constructivist architectures will be adequate, without needing a revolution of the underlying methodologies. However, particularly strong challenges still exist within the area of model based systems engineering. These challenges are in two general directions: maintaining consistency between various models/views, as well as between the models and implemented artifacts. The usage of constructivist AI designs in upcoming architectures will also impact the testing, verification, and validation aspects of the systems engineering processes. This is because, if the system behavior is partly synthesized at runtime, and could take forms not anticipated during design, it would be challenging to find testing and verification methods that demonstrate correctness of behavior prior to deployment.

In our experience, the availability of suitable implementation platforms and technologies is not a strong limitation in autonomous driving, with a possible exception being in the area of sensors for perception and localization. The capabilities of state of the art silicon, programming tools, communication technologies are adequate to implement the elements found in existing and proposed autonomous driving architectures. This is not to say that better tools for analyzing code, model checking, translation of models to executables etc. are not desired. All of this holds true within the constructionist approach. The shift to a constructivist approach however introduces paradigm shifts in implementation, notably for capabilities of programming languages. The pro-

gramming of autonomy requires language constructs supporting autonomous knowledge acquisition, realtime and any-time control, reflectivity, learning, and massive parallelization [66]. Such constructs are missing from the most common programming languages currently employed in the development of safety critical embedded systems.

The four steps for systems engineering proposed in Section 1.6 are complementary to the systems engineering processes based on the V-model, or those recommended by ISO26262. The former chiefly recommends activities, the latter introduces views and requirements to analyze and assure functional safety. Models, views and other artifacts underlie both. In our experience, systems engineering and associated modeling is usually given secondary importance during the prototyping phase. Once a functional prototype is available, the focus shifts to writing extensive requirements and acceptance criteria, which are then off-loaded to vendors and suppliers for process adherent development. The usage of early lightweight modeling, with subsequent refinement and additional models/views has the potential to keep development consistent not only during the prototyping phase, but also for making a smoother transition to high maturity implementations.

1.8.2 The influence of autonomy

For autonomous driving, the **impact of autonomy on architecture**, is mostly related to the need for specific functional components. These components are responsible for perception of the external world and its internal representation, localization of the vehicle with respect to some coordinate system, finding collision free trajectories between detected obstacles while staying on a driveable surface, and reacting to unexpected external and internal events with the intention of reaching a safer state. Also present are more "traditional" components governing basic motion of the vehicle viz. lateral and longitudinal accelerations and deceleration. Finally, overall energy management of the entire vehicle is also a concern. Beyond functional components, the architecture is driven by needs of safety, survivability, fault tolerant operation, and cost. This includes aspects of redundancy management, systematic degradation of available functionality in the presence of faults, movement of software functionality between similar computational platforms, and dynamic changes to inter-component data-flows depending on context and failure conditions. Among these, while redundancy has been a well-established pattern in the architecture of safety critical systems, the rest are not yet established. For example, it is common in AUTOSAR to statically specify inter-component dataflows and their contents. In case an architecture needs to modify the dataflow routing and content at runtime, all the possible combinations need to be determined and statically specified, along with the conditions under which each combination will occur. Then, all the possible

combinations and their transitions would need be proven safe, to meet certifiability and standards requirements. This is an excellent example of the "careful, manually crafted" characteristic of the constructionist approach. It is also the approach taken by the next generation Integrated Modular Avionics (IMA-2G) architectures, where an architecture reconfiguration is intended to be triggered by a component referred to as the 'reconfiguration supervisor' [26]. The approach of prior static definition of all possible combinations, is easier to certify, compared to the case where the reconfiguration supervisor contains algorithms to determine a configuration's safety at runtime, and uses this to develop potentially new configurations at runtime, which were not statically evaluated during the system design phase. This latter approach of evaluating circumstances and generating viable new solutions "on-the-fly" is a characteristic of human level reasoning, emulated by experimental constructivist AI, but is still a far cry from being implemented in domains like autonomous driving.

Given the rapid development of autonomous driving technologies and ever shortening time-to-market needs, a key challenge for automotive architects is to incorporate newer learning and keep the architecture relevant for the expected lifetime of the vehicle. Continuous deployment has hitherto not been a great concern in the automotive domain, but autonomy is pushing the envelope in this regard. A big obstacle to continuous deployment is the verification of any proposed changes, to assure system level safety properties are retained. To some extent, accelerated testing via virtualization techniques is one mitigating solution, but techniques for correctness-by-construction via contract based design, modularization, and composability are needed to first reduce the amount of testing and verification required for any given change.

For autonomous driving, we see the **impact of autonomy on systems engineering** to be mostly in the areas of testing, verification, and validation, and corresponding requirements management. This is driven by three characteristics of autonomous driving architectures: Newer types of perception sensors, growing system state space, and complexity of expected operational scenarios. Perception sensors like cameras, lidars, radars etc. each have multiple failure modes under various operating conditions. Complex, probabilistic sensor fusion guards against full perception failure to some extent, but in turn leads to a more complex assurance process and surprising common points of failure. For example, in one of our projects, similarity in filter time constants between pre-processing algorithms of two different types of sensors was questioned as a potential common failure mode. Such information is sometimes not even available to the systems integrator. The growth of system state space and complexity of operational scenarios implies that capturing requirements comprehensively is an additional challenge. Traditional testing and verification methods can not yield sufficient test coverage within reasonable timeframes. One solution is accelerated testing by means of 3D virtual worlds and scripted unit tests. For example, the evaluation of a new trajectory planning algorithm was conducted in one of our industrial projects

by letting the algorithm drive a virtual vehicle in a synthetic 3D world. An indication of usefulness of this technique is that approximately a thousand tests were executed in parallel on a GPU based computing cluster, within a few hours. Each test executed a different scenario from a scenario library (for example, traffic intersections, unexpected pedestrians running in front of the vehicle etc.) and metrics were gathered for each test. The metrics utilize parameters like resulting minimum distance to obstacles, accelerations within the cabin etc. and help in rapid evaluation of the planning algorithm. To the best of our knowledge, such testing infrastructure is not common in traditional automotive OEMs. Beyond testing and verification, we see little immediate influence of autonomy on systems engineering processes, although autonomy may demand stricter discipline in their execution. For example, we see comparatively lower influence to the process of gathering, analyzing, structuring, and documenting requirements, beyond a rigorous enforcement of the process itself. Autonomous driving may however give an added push to development of Intelligent Transport Systems (ITS) and connected systems-of-systems, increasing the number of stakeholders in the systems engineering process.

The **impact of autonomy on technical implementation** tools for autonomous driving is principally the introduction of technologies and platforms from other domains to the automotive industry. This introduction would, in turn, drive various ways to make the technologies and platforms more robust, in line with demands of the automotive domain (e.g. greater emphasis on verification, diagnosis, error handling etc.). The introduction of Linux in the form of Android is already occurring in the area of automotive infotainment, but its use for propulsion guidance, navigation, and control purposes is a relatively new phenomenon. The Automotive Grade Linux (AGL)[3] is a Linux Foundation workgroup with over 50 members including OEMs, Tier 1s, and system integrators. Although Linux may not be the final implementation of choice for safety critical ADAS functions, it is the de facto platform for prototyping, and also making inroads into solutions for HMI and telematics, which are crucial support functions for automated driving. The usage of silicon (GPUs), programming libraries, and 3D engines from the computer gaming industry are instrumental for accelerated testing and verification tooling. Computation middleware like OROCOS and ROS from the robotics domains, and communication middleware like DDS and ZeroMQ used in distributed information systems are also useful implementation technologies for autonomous driving. These middleware enable concrete component based software engineering techniques, as well as smarter communication patterns like push-pull, router-dealer, N-to-M, fan-in, and fan-out. This in turn introduces more options for technical architecture implementations.

1.8.3 Concluding remarks and future work

This chapter attempted to provide an overview of the architecture and systems engineering for autonomous driving, aimed towards the ambitious practitioner. We started by placing autonomous driving within the greater context of general purpose intelligent, autonomous systems, and highlighted some philosophical and practical gaps between two in terms of approaches and architectures. Despite the fact that the automotive industry practices a bottom-up approach to the engineering of autonomous driving systems, we believe it is valuable for practitioners to get ideas from the theories of Artificial Intelligence, Theory of Mind, Machine Consciousness, and Self-Awareness.

We then presented a functional architecture for autonomous driving, by introducing the key functional components and a way of connecting them together in an example architecture. Supporting the architecting efforts are practices for model based systems engineering. We introduced four modeling steps to aid the systems engineering process, each of which needs to be supported by requirements and test cases for verification and validation. Finally, we touched upon some key technologies used to prototype autonomous driving systems.

Future work lies along two dimensions: scientific and documentary. The scientific part needs to elaborate on the theories and algorithms for adapting concepts from different domains to autonomous driving, keeping in mind concerns of safety, certifiability, and engineering processes. The documentary part should add to our ambition for eventually creating a 'Handbook for Autonomous Driving'. This then needs to provide greater and in-depth treatment of model taxonomies, listings of architectural and technological options, as well as guidance for selection and application.

The engineering of Human Machine Interfaces was left unaddressed in this chapter. From an engineering viewpoint, the functionality offered by the system is somewhat distinct from the way it interacts with its users (HMI). But from the viewpoint of the system's users, the HMI **is** the functionality, making HMI issues of critical importance to autonomous driving. The topics of design space exploration and extra-functional metrics for architecture evaluation, like cost, reliability, performance, flexibility also deserve an in-depth treatment.

Finally, the importance of standards and certification of autonomous driving systems can not be underestimated. Existing functional safety standards like ISO26262, as well as industry specific norms like MISRA C, AUTOSAR etc. need to be re-evaluated and upgraded to meet the requirements of autonomous driving.

References

[1] (2007) INCOSE: Systems Engineering Vision 2020, Document No. INCOSE-TP-2004-004-02, Version 2.03

[2] (2010) Methodology Guidelines When Using EAST-ADL2. Public deliverable 5.1.1 of the ATESST2 project. URL http://www.atesst.org/home/liblocal/docs/ATESST2_Deliverable_D5.1.1_V1.1.pdf

[3] (2015) Automotive Grade Linux. https://www.automotivelinux.org/, URL https://www.automotivelinux.org/

[4] (2015) Biologically Inspired Cognitive Architectures (BICA) Society. The MAPPED Repository. http://bicasociety.org/mapped/, URL http://bicasociety.org/mapped/

[5] (2015) Distributed Messaging with Zero MQ. http://zeromq.org/, URL http://zeromq.org/

[6] (2015) FMI: Functional Mock-up Interface. https://www.fmi-standard.org/, URL https://www.fmi-standard.org/

[7] (2015) IEEE Spectrum. How Google's Autonomous Car Passed the First U.S. State Self-Driving Test. http://spectrum.ieee.org/transportation/advanced-cars/how-googles-autonomous-car-passed-the-first-us-state-selfdriving-test, URL http://spectrum.ieee.org/transportation/advanced-cars/how-googles-autonomous-car-passed-the-first-us-state-selfdriving-test

[8] (2015) National Highway Traffic Safety Administration - Preliminary Statement of Policy Concerning Automated Vehicles. http://www.nhtsa.gov/staticfiles/rulemaking/pdf/Automated_Vehicles_Policy.pdf, URL http://www.nhtsa.gov/staticfiles/rulemaking/pdf/Automated_Vehicles_Policy.pdf

[9] (2015) OSLC: Open Services for Lifecycle Collaboration. http://open-services.net/, URL http://open-services.net/

[10] (2015) Oxford Dictionaries. http://www.oxforddictionaries.com/definition/english/consciousness, URL http://www.oxforddictionaries.com/definition/english/consciousness

[11] (2015) SAE J3016: Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems. http://standards.sae.org/j3016_201401/, URL http://standards.sae.org/j3016_201401/

[12] (2015) The ARCADIA MBSE method for systems, hardware and software architectural design. https://www.polarsys.org/capella/arcadia.html, URL https://www.polarsys.org/capella/arcadia.html

[13] (2015) The AUTOSAR Platform. http://www.autosar.org/, URL http://www.autosar.org/

[14] (2015) The Capella Graphical Modeling Workbench. https://www.polarsys.org/capella/index.html, URL https://www.polarsys.org/capella/index.html

[15] (2015) The HAVE-it EU project. Deliverable D12.1 Architecture document. http://haveit-eu.org/LH2Uploads/ItemsContent/24/HAVEit_212154_D12.1_Public.pdf, URL http://haveit-eu.org/LH2Uploads/ItemsContent/24/HAVEit_212154_D12.1_Public.pdf

[16] (2015) The OMG MBSE Wiki page on Methodology and Metrics. http://www.omgwiki.org/MBSE/doku.php?id=mbse:methodology, URL http://www.omgwiki.org/MBSE/doku.php?id=mbse:methodology

[17] (2015) The OSEK-VDX portal. http://www.osek-vdx.org/, URL http://www.osek-vdx.org/

[18] (2015) The PREEMPT_RT patch for the Linux kernel. Real-Time Linux Wiki. https://rt.wiki.kernel.org, URL https://rt.wiki.kernel.org

[19] (2015) The Unreal gaming engine. https://www.unrealengine.com/, URL https://www.unrealengine.com/

[20] (2015) The Xenomai solution for real-time Linux. http://xenomai.org/, URL http://xenomai.org/

[21] Albinet A, Boulanger JL, Dubois H, Peraldi-Frati MA, Sorel Y, Van QD (2007) Model-Based Methodology for Requirements Traceability in Embedded Systems. In: Proceedings of 3rd European Conference on Model Driven Architecture Foundations and Applications, ECMDA'07, Haifa, Israel, URL https://hal.inria.fr/inria-00413488

[22] Albus J (1991) Outline for a theory of intelligence. Systems, Man and Cybernetics, IEEE Transactions ... 21(3):473–509, DOI 10.1109/21.97471, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=97471http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=97471

[23] Behere S, Törngren M (2014) Architecture challenges for intelligent autonomous machines: An industrial perspective. In: Proceedings of the 13th international conference on intelligent autonomous machines, Springer-Verlag, IAS-13

[24] Behere S, Törngren M (2015) A functional architecture for autonomous driving. In: Proceedings of the First International Workshop on Automotive Software Architecture, ACM, New York, NY, USA, WASA '15, pp 3–10, DOI 10.1145/2752489.2752491, URL http://doi.acm.org/10.1145/2752489.2752491

[25] Behere S, Törngren M, Chen D (2013) A reference architecture for cooperative driving. Journal of Systems Architecture 59(10, Part C):1095 – 1112, DOI http://dx.doi.org/10.1016/j.sysarc.2013.05.014, URL http://www.sciencedirect.com/science/article/pii/S1383762113000957, embedded Systems Software Architecture

[26] Bieber P, Boniol F, Boyer M, Noulard E, Pagetti C (2012) New Challenges for Future Avionic Architectures. Aerospace Lab (4):1–10

[27] Boehm BW (1988) A spiral model of software development and enhancement. Computer 21(5):61–72

[28] Bongard J, Lipson H (2005) Automatic synthesis of multiple internal models through active exploration. In: AAAI Fall Symposium: From Reactive to Anticipatory Cognitive Embodied Systems

[29] Bongard J, Zykov V, Lipson H (2006) Resilient machines through continuous self-modeling. Science 314(5802):1118–1121

[30] Bradski G (2000) The OpenCV Library. Dr Dobb's Journal of Software Tools

[31] Braun P, Broy M, Houdek F, Kirchmayr M, Mller M, Penzenstadler B, Pohl K, Weyer T (2014) Guiding requirements engineering for software-intensive embedded systems in the automotive industry. Computer Science - Research and Development 29(1):21–43

[32] Broy M (2006) Model-driven architecture-centric engineering of (embedded) software intensive systems: modeling theories and architectural milestones. Innovations in Systems and Software Engineering 3(1):75–102, DOI 10.1007/s11334-006-0011-y, URL http://link.springer.com/10.1007/s11334-006-0011-y

[33] Broy M (2007) Two Sides of Structuring Multi-Functional Software Systems: Function Hierarchy and Component Architecture. 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007) pp 3–12, DOI 10.1109/SERA.2007.129, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4296910

[34] Broy M (2012) Software and system modeling: Structured multi-view modeling, specification, design and implementation. In: Hinchey M, Coyle L (eds) Conquering Complexity, Springer London, pp 309–372

[35] Broy M, Huber F, Paech B, Rumpe B, Spies K (1998) Software and system modeling based on a unified formal semantics. In: Broy M, Rumpe B (eds) Requirements Targeting Software and Systems Engineering, Lecture Notes in Computer Science, vol 1526, Springer Berlin Heidelberg, pp 43–68

[36] Bruyninckx H (2001) Open robot control software: the OROCOS project. Proceedings 2001 ICRA IEEE International Conference on Robotics and Automation (Cat No01CH37164) 3:2523–2528, DOI 10.1109/ROBOT.2001.933002, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=933002

[37] Burns A (1999) The ravenscar profile. Ada Letters XIX(4):49–52, DOI 10.1145/340396.340450, URL http://doi.acm.org/10.1145/340396.340450

[38] Chalmers D (1996) The Conscious Mind: In Search of a Fundamental Theory. Oxford paperbacks, OUP USA

[39] Chella A, Cossentino M, Seidita V, Tona C (2010) An approach for the design of self-conscious agent for robotics. In: An A, Lingras P, Petty S, Huang R (eds) Active Media Technology, Lecture Notes in Computer Science, vol 6335, Springer Berlin Heidelberg, pp 306–317

[40] Cuenot P, Frey P, Johansson R (2011) The EAST-ADL Architecture Description Language for Automotive Embedded Software. In: Model-based engineering of embedded real-time systems, pp 297–307
[41] Davidson D (1990) Turing's Test. In: Said K (ed) Modelling the Mind, Oxford University Press
[42] Estefan JA, et al (2007) Survey of model-based systems engineering (mbse) methodologies. Incose MBSE Focus Group 25:8
[43] Feiler PH, Lewis BA, Vestal S (2006) The SAE Architecture Analysis and Design Language (AADL) a standard for engineering performance critical systems. 2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control DOI 10.1109/CACSD-CCA-ISIC.2006.4776814
[44] Ferris T (2009) On the methods of research for systems engineering. Annual Conference on Systems Engineering Research 2009(April), URL http://cser.lboro.ac.uk/papers/S10-62.pdf
[45] Ferris T (2012) Engineering Design as Research. In: Mora M, Gelman O, Steenkamp AL, Raisinghani M (eds) Research Methodologies, Innovations and Philosophies in Software Systems Engineering and Information Systems, IGI Global, DOI 10.4018/978-1-4666-0179-6, URL http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-4666-0179-6
[46] Fong T, Thorpe C (2001) Vehicle teleoperation interfaces. Autonomous Robots 11(1):9–18
[47] Forsberg K, Mooz H (1992) The relationship of systems engineering to the project cycle. Engineering Management Journal 4(3):36–43
[48] Forsberg K, Mooz H (1995) Application of the vee to incremental and evolutionary development. Systems Engineering in the Global Market Place pp 801–808
[49] Frost CR (2011) Challenges and opportunities for autonomous systems in space. In: Frontiers of Engineering:: Reports on Leading-Edge Engineering from the 2010 Symposium, National Academy of Engineering
[50] Gamez D (2008) Progress in machine consciousness. Consciousness and Cognition 17(3):887 – 910
[51] Henzinger T, Sifakis J (2006) The embedded systems design challenge. In: Misra J, Nipkow T, Sekerinski E (eds) FM 2006: Formal Methods, Lecture Notes in Computer Science, vol 4085, Springer Berlin Heidelberg, pp 1–15
[52] Hussain S (2013) Investigating architecture description languages (adls) a systematic literature review. Master's thesis, Linköpings universitet, Sweden
[53] Juarez Dominguez AL (2008) Feature Interaction Detection in the Automotive Domain. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, IEEE, pp 521–524, DOI

10.1109/ASE.2008.97, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4639390

[54] Kang EY, Enoiu EP, Marinescu R, Seceleanu C, Schobbens PY, Pettersson P (2013) A methodology for formal analysis and verification of east-adl models. Reliability Engineering and System Safety 120:127 – 138, DOI http://dx.doi.org/10.1016/j.ress.2013.06.007

[55] Kopetz H (2008) The Complexity Challenge in Embedded System Design. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), IEEE, pp 3–12, DOI 10.1109/ISORC.2008.14, URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4519555http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4519555

[56] Lee EA (2009) Computing needs time. Communications of the ACM 52(5):70–79, DOI 10.1145/1506409.1506426, URL http://doi.acm.org/10.1145/1506409.1506426

[57] Maes P (1987) Concepts and experiments in computational reflection. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, ACM, New York, NY, USA, OOPSLA '87, pp 147–155, DOI 10.1145/38765.38821, URL http://doi.acm.org/10.1145/38765.38821

[58] Maier MW, Rechtin E (2000) The Art of Systems Architecting (2Nd Ed.). CRC Press, Inc., Boca Raton, FL, USA

[59] Malavolta I, Muccini H, Pelliccione P, Tamburri D (2010) Providing architectural languages and tools interoperability through model transformation technologies. Software Engineering, IEEE Transactions on 36(1):119–140, DOI 10.1109/TSE.2009.51

[60] Mårtensson J, Alam A, Behere S (2012) The Development of a Cooperative Heavy-Duty Vehicle for the GCDC 2011: Team Scoop. IEEE Transactions on Intelligent Transportation Systems 13(3):1033–1049, DOI 10.1109/TITS.2012.2204876, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6236179http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6236179

[61] McCarthy J (1995) Making Robots Conscious of Their Mental States. In: Working Notes of the AAAI Spring Symposium on Representing Mental States and Mechanisms, Menlo Park, CA

[62] McDermott D (2007) Artificial intelligence and consciousness. In: Zelazo PD, Moscovitch M, Thompson E (eds) The Cambridge Handbook of Consciousness, Cambridge University Press, pp 117–150, URL http://dx.doi.org/10.1017/CBO9780511816789.007, cambridge Books Online

[63] Metzger A (2004) Feature interactions in embedded control systems. Computer Networks 45(5):625–644, DOI 10.1016/j.comnet.2004.03.002, URL http://linkinghub.elsevier.com/retrieve/pii/S138912860400043X

[64] Minsky M (1968) Matter, mind, and models. M L Minsky (ed) Semantic Information Processing

[65] Montemerlo M, et al (2008) Junior: The Stanford entry in the urban challenge. Journal of Field Robotics

[66] Nivel E, Thrisson K (2013) Towards a programming paradigm for control systems with high levels of existential autonomy. In: Khnberger KU, Rudolph S, Wang P (eds) Artificial General Intelligence, Lecture Notes in Computer Science, vol 7999, Springer Berlin Heidelberg, pp 78–87

[67] Odersky M, Spoon L, Venners B (2011) Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition, 2nd edn. Artima Incorporation, USA

[68] Pardo-Castellote G (2003) Omg data-distribution service: Architectural overview. In: Proceedings of the 2003 IEEE Conference on Military Communications - Volume I, IEEE Computer Society, Washington, DC, USA, MILCOM'03, pp 242–247

[69] Perotto F, Vicari R, Alvares L (2004) An autonomous intelligent agent architecture based on constructivist ai. In: Bramer M, Devedzic V (eds) Artificial Intelligence Applications and Innovations, IFIP International Federation for Information Processing, vol 154, Springer US, pp 103–115

[70] Quigley M, Conley K, Gerkey B, Faust J, Foote TB, Leibs J, Wheeler R, Ng AY (2009) ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software

[71] Royce WW (1987) Managing the development of large software systems: Concepts and techniques. In: Proceedings of the 9th International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA, ICSE '87, pp 328–338

[72] Rusu RB, Cousins S (2011) 3D is here: Point Cloud Library (PCL). In: 2011 IEEE International Conference on Robotics and Automation (ICRA), IEEE, pp 1–4, DOI 10.1109/icra.2011.5980567

[73] Samsonovich AV (2010) Toward a unified catalog of implemented cognitive architectures. In: Proceedings of the 2010 Conference on Biologically Inspired Cognitive Architectures 2010: Proceedings of the First Annual Meeting of the BICA Society, IOS Press, Amsterdam, The Netherlands, The Netherlands, pp 195–244, URL http://dl.acm.org/citation.cfm?id=1893313.1893352

[74] Sarma S, Dutt N, Gupta P, Nicolau A, Venkatasubramanian N (2014) On-chip self-awareness using cyberphysical-systems-on-chip (cpsoc). In: Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis, ACM, New York, NY, USA, CODES '14, pp 22:1–22:3, DOI 10.1145/2656075.2661648, URL http://doi.acm.org/10.1145/2656075.2661648

[75] Thórisson KR (2009) From constructionist to constructivist ai. In: AAAI Fall Symposium: Biologically Inspired Cognitive Architectures

[76] Thórisson KR (2012) A New Constructivist AI: From Manual Methods to Self-Constructive Systems. DOI 10.2991/978-94-91216-62-6_9

[77] Thórisson KR (2012) A new constructivist ai: From manual methods to self-constructive systems. In: Wang P, Goertzel B (eds) Theoretical Foundations of Artificial General Intelligence, Atlantis Thinking Machines, vol 4, Atlantis Press, pp 145–171

[78] Thórisson KR, Helgason HP (2012) Cognitive Architectures and Autonomy: A Comparative Review. Journal of Artificial General Intelligence 3(2):1–30, DOI 10.2478/v10229-011-0015-3, URL http://dx.doi.org/10.2478/v10229-011-0015-3

[79] Thórisson KR, Benko H, Abramov D, Arnold A, Maskey S, Vaseekaran A (2004) Constructionist design methodology for interactive intelligences. AI Magazine 25(4):77

[80] Törngren M, Qamar A, Biehl M, Loiret F, El-khoury J (2014) Integrating viewpoints in the development of mechatronic products. Mechatronics 24(7):745 – 762

[81] Turing AM (1950) Computing machinery and intelligence. Mind 59(236):433–460, URL http://www.jstor.org/stable/2251299

[82] Van Gulick R (2014) Consciousness. In: Zalta EN (ed) The Stanford Encyclopedia of Philosophy, spring 2014 edn

[83] Wallmark O, et al (2014) Design and implementation of an experimental research and concept demonstration vehicle. In: Vehicle Power and Propulsion Conference (VPPC), 2014 IEEE, pp 1–6, DOI 10.1109/VPPC.2014.7007042

[84] Westman J, Nyberg M (2013) A Reference Example on the Specification of Safety Requirements using ISO 26262. In: ROY M (ed) SAFECOMP 2013 - Workshop DECS (ERCIM/EWICS Workshop on Dependable Embedded and Cyber-physical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security, France, p NA, URL https://hal.archives-ouvertes.fr/hal-00848610

[85] Westman J, Nyberg M, Törngren M (2013) Structuring safety requirements in iso 26262 using contract theory. In: Bitsch F, Guiochet J, Kaniche M (eds) Computer Safety, Reliability, and Security, Lecture Notes in Computer Science, vol 8153, Springer Berlin Heidelberg, pp 166–177

[86] Ziegler J, et al (2014) Making bertha drive: An autonomous journey on a historic route. Intelligent Transportation Systems Magazine, IEEE 6(2):8–20, DOI 10.1109/MITS.2014.2306552

[87] Zima HP, James ML, Springer PL (2011) Fault-tolerant on-board computing for robotic space missions. Concurrency and Computation: Practice and Experience 23(17):2192–2204, DOI 10.1002/cpe.1768, URL http://dx.doi.org/10.1002/cpe.1768