A Generic Architecture Style for Self-Adaptive Cyber-Physical Systems

Doctoral Thesis

(Dissertation)

to be awarded the degree of

Doctor of Engineering

(Dr.-Ing.)

submitted by

Meng Zhang

from Hefei, China

approved by

the Faculty of Mathematics/Computer Science and Mechanical Engineering,

Technische Universität Clausthal

Date of Oral Examination: March 1, 2023

Dissertation Clausthal, ISSE-Dissertation 26, 2023

Chairperson of the Board of Examiners

Prof. Dr. Thorsten Grosch

Chief Reviewer

Prof. Dr. Andreas Rausch

2. Reviewer

Prof. Dr.-Ing. Gert Bikker

3. Reviewer

Prof. Dr. Christian Siemers

To my parents

Acknowledgment

Throughout the writing of this dissertation, I have received a great deal of support and assistance.

I would first like to thank my supervisor, Prof. Dr. Andreas Rausch, whose expertise was invaluable in formulating the research questions and methodology. Additionally, I would like to thank the other supervisors, Prof. Dr.-Ing. Gert Bikker and Prof. Dr. Christian Siemers, for the patient support and feedback about this dissertation. Insightful feedback from both of you pushed me to sharpen my thinking and brought my work to a higher level.

In addition, I would like to thank my parents for their wise counsel and sympathetic ear, particularly for their continuous support and understanding when undertaking my research and writing my dissertation. Your understanding was what sustained me this far.

I could not have completed this dissertation without my dear former and current colleagues at the institute like Mirco Schindler, Dirk Kluge, Marco Körner, Jörg Grieser, Tim Warnecke, Karina Rehfeldt, Peter Engel, and Andreas Vorwald. They provided stimulating discussions and happy distractions to rest my mind outside of my research.

Finally, I would like to thank all other colleagues, who have not been listed in previous sections, but always ensured a pleasant working time and supported me with advice and contributions during the preparation of this dissertation.

Clausthal-Zellerfeld, March 1, 2023 Meng Zhang

Abstract

Current concepts of designing automatic control systems rely on dynamic behavioral modeling by using mathematical approaches like differential equations to derive corresponding functions [1], and slowly reach limitations due to increasing system complexity. Along with the development of these concepts [1]–[5], an architectural evolution of automatic control systems is raised.

This dissertation defines a taxonomy to illustrate the aforementioned architectural evolution relying on a typical example of control application: adaptive cruise control (ACC). Current ACC variants, with their architectures considering control theory, are analyzed. The analysis results indicate that the future automatic control system in ACC requires more substantial self-adaptation capability and scalability. For this purpose, more complicated algorithms requiring different computation mechanisms must be integrated into the system and further increase system complexity. This makes the future automatic control system evolve into a self-adaptive cyber-physical system and constitutes significant challenges for the system's architecture design.

Inspired by software engineering approaches for designing architectures of software-intensive systems, a generic architecture style is proposed. The proposed architecture style serves as a template by following the developed design principle to construct networked architectures not only for the current automatic control systems but also for self-adaptive cyber-physical systems in the future. Different triggering mechanisms and communication paradigms for designing dynamic behaviors are applicable in the networked architecture.

To evaluate feasibility of the architecture style, current ACCs are retaken to derive corresponding logical architectures and examine architectural consistency compared to the previous architectures considering the control theory (e.g., in the form of block diagrams). By applying the proposed generic architecture style, an artificial cognitive cruise control (ACCC) is designed, implemented, and evaluated as a future ACC in this dissertation. The evaluation results show significant performance improvements in the ACCC compared to the human driver and current ACC variants.

Table of Contents

List of Figures

List of Tables

Used Abbreviations

ACC	**A**daptive **C**ruise **C**ontrol
ACCC	**A**rtificial **C**ognitive **C**ruise **C**ontrol
ADAS	**A**dvanced **D**river **A**ssistance **S**ystem
A-FL	**A**daptation **F**eedback **L**oop
AI	**A**rtificial **I**ntelligence
BEV	**B**attery **E**lectric **V**ehicle
CLZ	**Cl**austhal-**Z**ellerfeld
CO-FL	**C**ontrol **O**bjectives **F**eedback **L**oop
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
CTCU	**C**ycle-**T**ime-based **C**ontrol **U**nit
DAG	**D**irected **A**cyclic **G**raph
DDS	**D**ata **D**istribution **S**ervice
DP	**D**ynamic **P**rogramming
ECU	**E**ngine **C**ontrol **U**nit
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
GPU	**G**raphics **P**rocessing **U**nit
GS	**G**o**s**lar
HMI	**H**uman **M**achine **I**nterfaces
ICE	**I**nternal **C**ombustion **E**ngine
IDM	**I**ntelligent **D**river **M**odel
IPDI	**I**ncreasing **P**recision with **D**ecreasing **I**ntelligence
KDE	**K**ernel **D**ensity **E**stimator
M-FL	**M**onitoring **F**eedback **L**oop

MIAC	**M**odel **I**dentification **A**daptive **C**ontrol
MIMO	**M**ultiple-**I**nput and **M**ultiple-**O**utput
ML	**M**achine **L**earning
MPC	**M**odel **P**redictive **C**ontrol
MQTT	**MQ** **T**elemetry **T**ransport
MRAC	**M**odel **R**eference **A**daptive **C**ontrol
NARX	**N**onlinear **A**uto**r**egressive Network with E**x**ogenous
NIST	**N**ational **I**nstitute of **S**tandard and **T**echnology
OS	**O**perating **S**ystem
QoS	**Q**uality **o**f **S**ervice
RAU	**R**oute-based **A**daptation **U**nit
RL	**R**einforcement **L**earning
ROS	**R**obot **O**perating **S**ystem
RPC	**R**emote **P**rocedure **C**all
RSAU	**R**oute-**S**egment-based **A**daptation **U**nit
SC	**S**ensor of **C**ontrolled Variable
SE	**S**ensor of **E**nvironment Variable
SISO	**S**ingle-**I**nput and **S**ingle-**O**utput
SP	**S**ensor of Controlled **P**lant
SR	**S**ensor of **R**eference Variable
SU	**S**ensor of **U**ser
SUMO	**S**imulation of **U**rban **Mo**bility
TD	**T**emporal **D**ifference
TraCI	**Tra**ffic **C**ontrol **I**nterface

1 Introduction

This chapter presents a short introduction to the whole dissertation. First, the motivation behind the dissertation is presented. The objective of the dissertation is also included in this chapter. Subsequently, the concrete contributions of the dissertation are summarized. Finally, this chapter ends with the content and structure of the dissertation.

1.1 Motivation

Automatic control systems have been increasingly deployed in diverse applications covering almost every area in daily life, ranging from an ordinary oven in the kitchen to significant industrial devices like production machines and mobilities like cars and drones [6]–[8]. Generally, the automatic control system is designed to fulfill predefined control targets through automated control of dynamic technical processes, particularly against different disturbances and influences from the surrounding system environment [9].

Traditional design approaches to such automatic control systems, like classical and modern control theory, rely on mathematical modeling using differential and state-space equations to derive dynamic behavioral functions of the controlled technical processes [10]. In such approaches, development engineers must have the explicit domain knowledge to model the controlled processes at the design time. Thus, they can parametrize a controller to keep it working as expected within a limited working range, considering bounded uncertainties like environmental disturbances. After the system design, the controller parametrization remains static at run time.

Due to the limited working range, static parametrization of the controller limits the control system's performance in diversified operating situations at run time, which hinders system flexibility. Thus, the reconfigurable controller is applied to enable the system to work appropriately in more diversified operational cases [9][11]. For this purpose, a secondary control loop relying on the technology of variable monitoring is deployed on top of the primary control loop to estimate the current situational context of the controlled

processes and accordingly make decisions to adapt the controller parametrization at run time [12].

However, the diversity required in the operational cases of automatic control systems is still increasing. Rapidly developing sensor technologies increase context data for the controlled technical processes, and accompanying complicated algorithms for data interpretation on higher levels of abstraction are integrated into the secondary control loop to make the system operate appropriately in diversified operating situations [13][14]. In this sense, the complexity of automatic control systems is increasing continuously.

Such a development trend constitutes significant challenges for the design of automatic control systems. As discussed earlier, the field of control theory initially focuses on the control flow of dynamic processes. With this idea in mind, system design focuses on the conception of a pure embedded system, and takes scheduling for fulfilling real-time interaction with the physical world as the most crucial constraint. Thus, required real-time interaction with the system's surrounding physical world generates a hard timing constraint for the cycle time of the whole control loop. All system parts on the control loop perform under the consideration of this timing constraint. This constraint may become a bottleneck that limits the complexity of the system during the overall system design.

For example, it may conflict with the high complexity of the previously mentioned complicated algorithms due to their long (and even possibly non-deterministic) computation time. Thus, a tradeoff between algorithm complexity and the required cycle time to guarantee reliable system scheduling must be considered while designing the system. Along with the increasing system complexity in the future, such tradeoffs will increasingly arise in system design and become much more difficult to manage. Thus, traditional design approaches of control theory that primarily focus on modeling the controlled technical processes to derive dynamic behavioral functions with mathematical equations have reached their limitations.

Against such a background, system architecture design from the software engineering research field that considers system construction and makes the software-intensive system's complexity manageable, becomes crucial [15]. Unlike control theory, software engineering focuses on organizing fundamental software building blocks, including their

functionalities and behavioral interactions in an overall system architecture [16][17]. Instead of a simplified superposition of the primary and secondary control loops, hierarchical architectures with multiple layers are proposed [2][15][18][19]. These architectures roughly derive an implicit design paradigm about the responsibility assignment of different hierarchical layers. However, a generic concept of architecture design on a higher meta-level that can be applied to systematically design different kinds of automatic control systems with high complexity is still missing.

Without such a generic concept, the architecture design of automatic control systems with increasing complexity is becoming more challenging. Integrating algorithms with high complexity (e.g., by using heuristic, linguistic, or artificial intelligence (AI) approaches) would deteriorate the system's scheduling issues. In addition, such integration may make the system include heterogeneous parts with different expected computation mechanisms. Thus, the automatic control system evolves from a pure embedded system into a so-called cyber-physical system with a hybrid construction [20][21]. A typical cyber-physical system is a vehicle’s autonomous driving system [22]. It includes the system parts with interpretation on a high semantic level like perception and decision-making. But it also includes system parts processing on a low data level, like real-time feedback control [23]. Along with further development in the future, the requirements of such cyber-physical systems with higher complexity will increase more progressively than in the past.

The evolution of the automatic control system as the cyber-physical system causes the system to acquire features of other software-intensive systems [15] (e.g., the self-adaptive system with a so-called self-adaptation capability established by the software engineering field [24][25]). In this sense, it can be said that control theory and software engineering are converging due to the similarity of their investigated systems’ features (cf. Section 4.1). Such a background motivates the reconsideration of the architecture design concept of automatic control systems from the view of software engineering, particularly in the case of future self-adaptive cyber-physical systems. Thus, the system architecture design can benefit from mature software engineering approaches for the architecture design of software-intensive systems.

1.2 Objectives of this Dissertation

As discussed in the previous section, system architecture design will become a meaningful and increasingly important research topic in the future. Thus, this dissertation takes as a main objective the development of a generic concept for the architecture design of different automatic control systems, particularly for future automatic control systems with high complexity designed as self-adaptive cyber-physical systems [20]. For this purpose, it is necessary to investigate current design concepts of automatic control systems, primarily aiming to analyze their established underlying architectures. This dissertation takes a vehicle's original adaptive cruise control (ACC) and its current variants by applying the previously mentioned automatic control concepts as concrete application examples to analyze the current technical limitations and derive a functional vision for the future.

The functional vision of the ACC would constitute further challenges for the architecture design of future automatic control systems, which the expected generic concept of architecture design must address. Since the expected concept will be applicable to different automatic control systems, particularly for the future self-adaptive cyber-physical system, this dissertation aims to define a generic architecture style. This generic architecture style will be applicable as a template to derive the appropriate architectures of different automatic control systems while considering their boundary conditions in concrete applications. In particular, the architecture style shall also cover the use case of designing self-adaptive cyber-physical systems. In this architecture style, architectural paradigms from different perspectives as further sub-objectives of the dissertation will be defined.

Considering a static view of system construction, a pattern of the fundamental component structure must first be developed, defining a set of components and their corresponding functionalities as basic building blocks in the overall architecture. Subsequently, a paradigm of system construction must also be determined to build more complicated system architectures based on the fundamental component structure. In addition to the static view of system construction, the dynamic view that focuses on the run-time behaviors of the components and their interactions is also crucial for developing the generic architecture style. For this purpose, use cases for the whole architecture

must be defined, describing communication paths between the components in corresponding scenarios. To realize concrete communication between related components on the communication path, it is also still necessary to define potential concrete communication paradigms, which requires an investigation of common communication architecture patterns (cf. Section 2.2.7).

After developing the generic architecture style, this dissertation shall evaluate its feasibility and generalization potential. For this purpose, it is essential to derive instance architectures for the current ACC variants by applying the developed generic architecture style. The derived instance architectures must be compared with previous architectures concerning the control flow of dynamic processes from the view of control theory to check the generalization potential of the architecture style based on architectural consistency. In addition, an example architecture for a potential future ACC variant named artificial cognitive cruise control (ACCC), with improvements for overcoming the previously analyzed technical limitations, must be derived. This will be used to evaluate the feasibility of the generic architecture style. As a further objective in this dissertation, the mentioned technical limitations of current ACC variants must be overcome by the ACCC, and this must be evaluated quantitatively.

1.3 Contributions of this Dissertation

The main contributions of this dissertation are as follows:

- A comprehensive architectural taxonomy of current automatic control concepts is proposed considering the relationship between the included technical system and its managed physical system (primarily referring to controlled dynamic processes in the physical world). Based on this comprehensive taxonomy, current ACC variants on the market as concrete application examples of the control concepts are categorized and analyzed to derive a functional vision for the future.
- Based on the functional vision, existing challenges are summarized and a vision for the architecture design of future automatic control systems is presented. Artificial cognitive control is defined as a control concept for the

next generation of automatic control systems that must be designed as self-adaptive cyber-physical systems.

- As the most significant contribution of this dissertation, a generic architecture style including a fundamental component structure and accompanying paradigm for the system construction is proposed to address the pain points of current system architecture design. This architecture style can be applied as a template to design architectures for different automatic control systems, particularly for future self-adaptive cyber-physical systems.
- As a concrete potential future ACC variant following the concept of artificial cognitive control, an innovative so-called artificial cognitive cruise control (ACCC) that works by learning and satisfying a single driver's driving preferences is proposed. In addition, the ACCC also learns the experienced historical context of driving environment.
- Recommendations are made for meaningful future research directions to improve the system's scalability and cognition or, more generally, to improve intelligence capability.

1.4 Content and Structure

This dissertation consists of six chapters. Chapter 1 briefly introduces the framework of this dissertation, including its motivation, objectives, and contributions. Chapter 2 focuses on the state of the art in relevant fundamental theories, these are, generally divided into control theory, software engineering for architecture design of software-intensive systems, generic communication architecture patterns, and application of underlying AI-based technologies in the dissertation's implementation.

The third chapter deals with a case study about the architectural evolution of the automatic control systems mentioned above with the help of concrete examples of ACC variants. In the case study, the functionalities of these ACC variants and their technical limitations (particularly their underlying architectures) are systematically analyzed to review the architectural evolution of recent years. Along with this evolution, a functional vision of ACC in the future is then illustrated based on two postulated future variants, which would make the original pure embedded systems become self-adaptive cyber-

physical systems with significantly higher architectural complexity. Thus, existing challenges for architecture design of underlying automatic control systems are constituted. Finally, a future concept of the automatic control system, which aims to fulfill expected features of the ACC variants, is roughly defined and named artificial cognitive control.

Chapter 4 introduces a generic architecture style serving as a template for the architecture design of different automatic control systems. This architecture style particularly considers the artificial cognitive control system designed as a self-adaptive cyber-physical system to fulfill the previously mentioned existing challenges. Fundamental component structure and different design paradigms are introduced as parts of the architecture style. The proposed architecture style is applied to instantiate example logical architectures of current automatic control systems by applying software engineering approaches, particularly considering the previously presented concrete examples of ACC variants. The instantiated logical architectures are then compared with architectures from the view of control theory to evaluate the architecture style's generalization potential by checking the architectural consistency.

To further evaluate the feasibility of the proposed architecture style, a future ACC variant called artificial cognitive cruise control, which realizes both previously postulated future ACC variants in the functional vision, is designed and implemented by applying the architecture style, as presented in Chapter 5. A systematic performance evaluation of ACCC is also included in this chapter.

Finally, a summary of this dissertation is provided in Chapter 6. In this chapter, the limitations of the dissertation are also discussed, which can be taken as indications of future research activity. In addition, some recommendations indicating potential future research directions in the long term (after this dissertation) are also summarized.

2 State of the Art

This chapter introduces all related theoretical fundamentals in the scope of this dissertation, which mainly involves three fields: control theory, architecture design of software-intensive systems, and generic communication architecture patterns for defining dynamic system behaviors. Additionally, related underlying AI-based technologies utilized in the practical application for evaluating the proposed architecture style in this dissertation are also presented in this chapter.

2.1 Control Theory

The research field of control theory focuses on systems for automatic control of technical processes (e.g., within industrial machines or applications) which have been widely applied almost everywhere in the world. For this purpose, models and algorithms must be developed to regulate the system inputs so that the system outputs can be maintained within a desired state, considering additional factors like time delay, system stability, and optimality in parallel [26][27].

Since this dissertation focuses on developing a generic architecture style that serves as a template for the architecture design of automatic control systems, a taxonomy covering different automatic control systems is required. However, current well-known taxonomies like classical[1] and modern[2] control theory focus intensively on underlying mathematical approaches instead of system architectures [10]. Thus, this dissertation takes as a reference another taxonomy derived from a stage model proposed by Iwanek within his dissertation and further developes this taxonomy [28].

In Iwanek's proposed stage model, different functional areas are defined and then used as different perspectives to categorize intelligence levels of diverse mechatronic systems. In these functional areas, it is emphasized that "control and feedback control"

[1] Classical control theory relies on mathematical approaches like Laplace or z-transform to convert differential equations for modeling the physical system from the time domain into the frequency domain, aiming to reduce mathematical complexity.

[2] Modern control theory relies on time-domain analysis and converts differential equations into state equations based on state variables, which can be further processed by using linear algebra approaches.

is one of the most important, which points precisely to the topic of automatic control and thus is of interest for this dissertation. Based on the stage model of Iwanek, other researchers have attemped to assign automatic control approaches to different intelligence levels within the stage model, as shown in Figure 2.1.

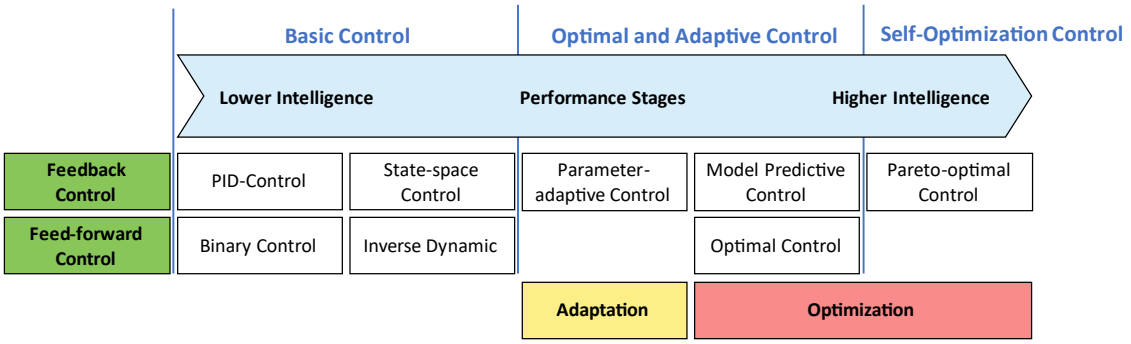


Figure 2.1: Exemplary Assignment of Automatic Control Systems to the Stage Model and its Derived Categories [11]

Based on the assignment results, different automatic control systems, including feed-forward and feedback control, are divided by Trächtler and Gausemeier [11] into three categories along with increasing performance stages regarding the level of intelligence: (a) fundamental control, which is called *basic control* in this dissertation, (b) *optimal and adaptive control*, and (c) *self-optimization control*. Since this dissertation primarily focuses on feedback control systems, more details of these systems will be provided in the following sections. In this dissertation, the categories based on the increasing performance stages are taken as a reference to derive the taxonomies of this dissertation from the perspective of system architecture evolution, which will be discussed in Chapter 3.

2.1.1 Basic Control

The first category in the lowest performance stage refers to basic control systems. Control engineering aims to realize the automated goal-oriented influence of a dynamic process during operation. In this case, the behaviors of the focused dynamic process as a controlled system must usually be explicitly known in advance and thus modeled using mathematical differential equations to derive corresponding functions. The designed control system takes the current value of the system's output variable as the controlled

variable (y*(t)*). It aims to maintain the controlled variable as a given set value of the reference variable (w*(t)*) as much as possible, as shown in Figure 2.2.

Since the behaviors of the controlled system can be influenced by disturbance variables (*z(t)*) from the system environment, the control system also focuses on compensation for environmental disturbances. If disturbances from the environment and system behavior are precisely known, the technologies of feed-forward control can be applied to set the value of the reference variable. However, in the case of unknown disturbances and uncertain parameters, the approaches of control engineering, like feed-forward control with an opening chain of effect, become insufficient [11]. Instead, feedback control is able to compensate for uncertainties by relying on a closed control loop. It calculates a deviation (so-called control error: *e(t)*) between the current value of the controlled variable (y*(t)*) and the set value of the reference variable (w*(t)*). A controller subsequently determines a control activity aiming to minimize this deviation. Based on the controller activity, the final control element manipulates the value of the manipulated variable (*u(t)*) to influence the dynamic processes of the controlled system. Figure 2.2 shows the process flow of such a closed control loop with a block diagram.

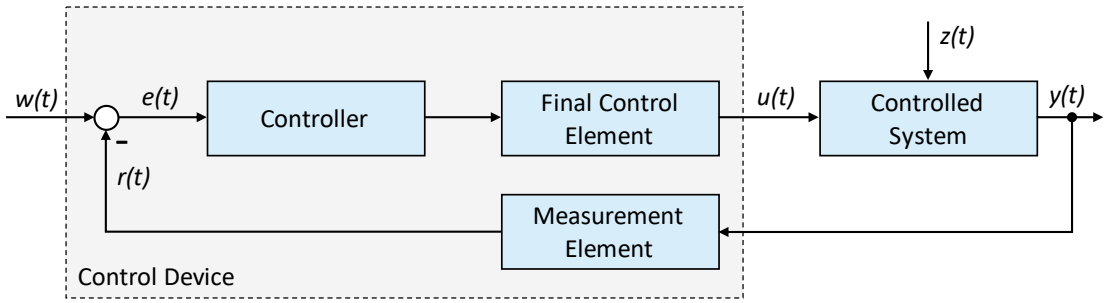


Figure 2.2: Block Diagram of Fundamental Feedback Control Loop [11]

As shown in Figure 2.2, the most crucial component in the feedback control system is the controller. In the design of the controller, it is necessary to model the dynamic processes of the controlled system, which can be accomplished using mathematical differential equations, as discussed previously.

In this design process, the Laplace- or z-transform (appropriate for continuous and discrete systems, respectively) for reducing the complexity of the design problem is often used to investigate system behaviors in the frequency domain. These transforms

convert the differential equations in the time domain into a transfer function (*G(s)*) in the frequency domain, which describes the relationship between the input (*U(s)*) and output (*Y(s)*) of the controlled system with algebraic equations [11]. In the case that the behavior of the controlled system is linear and time-invariant, the differential equation in the time domain would also be linear with constant coefficients, for which standard transfer functions such as a PT1- or PT2-element with application parameters like a so-called time constant can be used [9]. These time constants play an essential role in the controller's design while determining the optimized values of corresponding application parameters.

The controller design strongly influences the performance of the whole control system. During the design stage, different criteria like the system's reaction time to the environmental change, damping of the system's response, and its stationary accuracy after the damping process must be considered [11]. In particular, system stability must be evaluated. Otherwise, unstable oscillations in unfavorable cases may even cause the failure of the whole control system. For this purpose, different approaches addressing system stability, such as NYQUIST and HURWITZ, have been developed [29][30].

The most significant and well-known approach is the proportional-integral-derivative controller (PID controller), which works based on a parallel connection consisting of a proportion-element (*P*), an integrator (*I*), and a differentiator (*D*), as shown in the following equation. The P-element is responsible for adjusting the value of the manipulated variable (*u(t)*), aiming to align with the control error (*e(t)*), while the integrator is responsible for stationary accuracy. The differentiator arranges for the reaction time and the response damping of the controlled system. The control effects of the P-element, the integrator, and the differentiator are influenced by the value configuration of corresponding application parameters: K_p, K_I, and K_D (cf. Equation (2.1)). In some cases, certain parts of the PID controller can also be removed to build variants like the P controller, PI controller, or PD controller [11].

$$U(s = jw) = \left(K_P + K_I \frac{1}{s} + K_D \frac{s}{1 + T_N s}\right) \cdot Y(s) \tag{2.1}$$

$$= K_R \frac{(1 + T_{R1}s)(1 + T_{R2}s)}{s(1 + T_N s)} \cdot Y(s)$$

The first step in designing a controller is to decide on an appropriate variant. Subsequently, it is also essential to determine the parametrization of the controller by adjusting the values of corresponding parameters: T_{R1}, $T_{R2,}$ and K_R. In this case, T_{R1} and T_{R2} as time constants can be applied to compensate for the slowest sub-processes in the controlled system, and K_R is used to realize an expected response damping [11].

The control approach based on a conversion from the time domain into the frequency domain is appropriate for single-input and single-output (SISO) systems. However, it reaches its performance limit in the case of multiple-input and multiple-output (MIMO) systems. Thus, an intermediate parameter between the input and output variables of the system is developed to describe the system's internal state, which is named state variable (*x*). In this case, a complicated high-order differential equation can be avoided by decomposing it into multiple differential equations of the first order. Thus, a state-space Equation (2.2) and an output differential Equation (2.3), including multiple matrices, are constituted. The matrices in the equations covering a state matrix (*A*), a feedthrough matrix (*D*), an input (*B*), and an output matrix (*C*) are used to describe the behavior of the controlled system, as shown below:

$$\dot{x}(t) = Ax(t) + Bu(t); x = [x_1, x_2, \dots, x_n] \tag{2.2}$$

$$y(t) = Cx(t) + Du(t); y = \left[y_1, y_2, \dots, y_q\right] \tag{2.3}$$

The eigenvalues (*λ*) of the state matrix (*A*) can be calculated to analyze the dynamics of the controlled system. Further, they can also be used as specifications to design the state controller represented by a controller matrix (*R*), following the rules included in the following Equation (2.4). Thus, an updated system matrix (A_R) for the whole system can be constituted as follows:

$$u = -Rx \tag{2.4}$$

$$A_R = A - BR \tag{2.5}$$

Controllability and observability are two additional system properties that must still be considered for designing an appropriate state space control. Controllability implies that the state variable (x) can be steered by adjusting the manipulated variable (u) from an initial value to any final value within a finite time duration. Observability refers to the fact that the system's internal state, represented by the non-measurable state variable (x), can be reconstructed by measuring the output variable (y). For this purpose, the concept of state space control based on the so-called Luenberger-observer is developed, as shown in Figure 2.3.

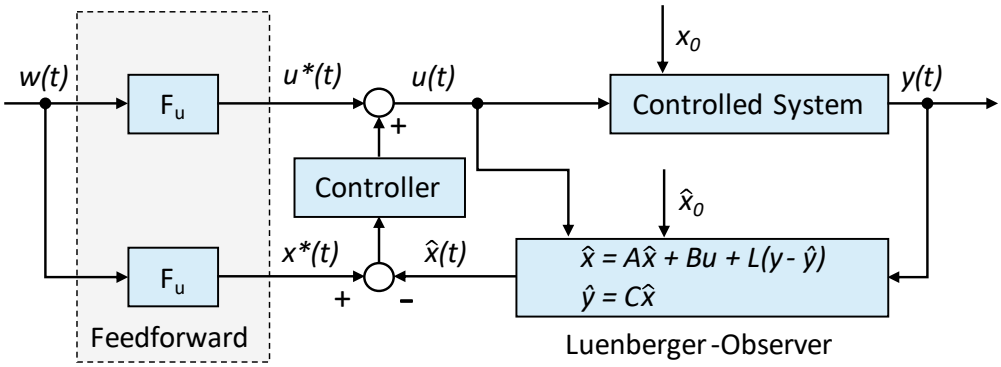


Figure 2.3: Block Diagram State-Space Control with State Observer [11]

In this concept, the observer aims to determine the state variables as a vector ($\hat{x}$). Thus, a model of the controlled system's dynamics can be constructed relying on the observed input and output variables of the controlled system (*u(t)* and *y(t)*). The observer compares the actual output of the controlled system and the model's output to balance the deviation due to unknown disturbances from the system environment and initial system state so that the estimation error by the model can be minimized.

2.1.2 Optimal and Adaptive Control

The second category in the following performance stage refers to control-engineering concepts, which either rely on optimization approaches to optimize the control performance, or on adaptation of the control system to work against changeable environmental conditions [11]. A typical example approach in this category is optimal control. The controller's design in previously mentioned control concepts primarily relies on the human engineers' experiences, particularly in state-space control. In this case,

human engineers must iteratively adjust controller parametrization to guarantee the expected system performance represented by the eigenvalues of the state matrix. Unlike such an approach, optimal control relies on a cost function consisting of multiple criteria and corresponding weights, as shown in equation (2.6).

$$J = \frac{1}{2}\int_0^\infty (x^T(t)Qx(t) + u^T(t)Su(t))dt \tag{2.6}$$

Different criteria for evaluating control performance, such as high response speed and low oscillation (which also means less energy consumption), are considered and represented by different terms in the equation. Additionally, corresponding weighting matrices (*Q* and *S*) are also included in the equation. By minimizing the final cost function (*J*), the optimal control parametrization can be determined, and thus the design of the controller becomes an optimization problem. Since the solution of this optimization problem leads to a non-linear matrix equation, which is also called an algebraic Riccati equation, the controller designed by following such an approach is called a Riccati controller [29]. In this approach, control performance is represented by the weighting matrices (*Q* and *S*) instead of the eigenvalues of the state matrix.

In addition to optimal control, another example concept included in the category of optimal and adaptive control is model predictive control (MPC). MPC relies on the idea of prediction of system behavior by a time-discrete model of the dynamic processes included in the linear or non-linear controlled system [11], as illustrated in the following Equation (2.7) and Equation (2.8).

$$x(k+1) = Ax(k) + Bu(k) \tag{2.7}$$

$$y(k) = Cx(k) \tag{2.8}$$

Unlike the Riccati controller's offline optimization, which means that the optimization process must be completed before system operation, MPC includes online optimization that is live during system operation. Its block diagram is illustrated in Figure 2.4.

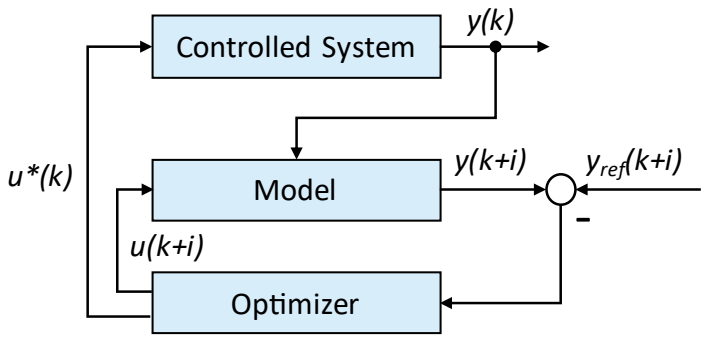


Figure 2.4: Block Diagram of Model Predictive Control [11]

In MPC, the optimizer is initialized by an initial value of the system's input and output. The optimizer in MPC performs an optimization process, aiming at minimizing a cost function (*J*) with consideration of a finite time horizon. The model is responsible for predicting the future outputs (*y*) of the controlled system until the end of the considered finite horizon, which is then compared with the reference trajectory of the output variable (y_{ref}) to estimate the deviation. The optimizer takes the deviation as its input. Thus, the optimizer can determine the optimal system input for the current time point (*u*(k)*) and forward it into the controlled system.

As discussed above, a cost function is required for the optimizer in MPC to identify the optimal solution for the controlled system's input variable (*u**). For this purpose, considering the prediction model, the cost function should describe the system state within the finite time horizon. A typical example cost function is shown in Equation (2.9), in which the weighting matrices (*Q* and *S*) are applied again. In this case, the optimizer attempts to determine the optimal system input (*u*) considering the accuracy of the output variable's predicted trajectory compared to the reference trajectory and the oscillation of the system input, which also relates to the required energy consumption.

$$J\big(Y(k), U(k)\big) = (Y(k) - Y_{ref}(k))^T Q\big(Y(k) - Y_{ref}(k)\big) + U^T(k) S U(k) \tag{2.9}$$

MPC shows remarkable advantages by considering constraints of the system state and the system's input and output simultaneously, covering the controlled system's technical and physical limitations [11]. However, from the opposite perspective, such an approach also strongly increases the non-linearity of the optimization problem and thus makes MPC possibly require greater computational effort.

The previously presented control concepts all rely on modeling of the controlled system, which requires precisely describing the system's dynamic behaviors but does not involve consideration of time-dependent behavioral changes. Instead, the control engineering approaches with an adaptation of the controller parametrization at runtime are developed to deal with the controlled system with time-dependent behavioral changes, which are also categorized as adaptive control in the control theory [30] (e.g., the parameter-adaptive control in Figure 2.1).

Different strategies for adaptation of controller parametrization have been developed in the parameter-adaptive control, depending on whether feedback of system dynamics is available or not available, respectively. In the case of unavailable feedback, the adaptation process can be understood as a pure feed-forward control task. In this case, an approach like gain scheduling can be applied, which linearizes the nonlinear model of the controlled system under different boundary conditions as operating points with corresponding assumed linear behaviors. Subsequently, the controller parametrizations such as gains are determined and assigned to the operating points. Thus, gain scheduling can be completed with the help of a scheduling variable, which aims to identify the system's state or the output. In this case, since the stability of the nonlinear system cannot be analytically guaranteed, a simulation-based evaluation of the system stability is required [11].

If time-dependent environmental disturbances or system dynamics must be considered in the adaption of the controller parametrization, feedback for continuous variable monitoring of the system is required. For example, in model reference adaptive control (MRAC), a reference model is deployed in parallel to estimate a reference output of the system so that controller adaptation can be completed, aiming at approximating the reference output and the actual output of the controlled system. Unlike MRAC, model identification adaptive control (MIAC) relies on model-based estimation approaches to identify the system parameters' states and thus adjust the controller parametrization.

2.1.3 Self-Optimization Control

The final category in the highest performance stage defined by Trächtler and Gausemeier [11] is self-optimization control. The self-optimization system is defined as

a system with a capability for automatic adaptation to changes in the environment and user profile. The intelligent mechatronic system is defined as a system with an automatic adaptation of its system structure and parameters at runtime in changeable environments or under different operating modes [11]. Based on these two definitions, the self-optimization control is defined by Trächtler and Gausemeier [11] as a control system with the capability for situational adaptation of its system configuration to the optimal setup with the help of a controller, which is further developed based on the concept of adaptive control [28].

To realize self-optimization control, the approaches of multi-objective optimization and reconfiguration of the controller are essential. Multi-objective optimization is responsible for identifying the optimal system configuration considering corresponding constraints and criteria, which are typically in conflict. The second required approach for reconfiguring the controller aims to guarantee the stability and functionality of the fault-tolerant system by adapting the system structure or the controller's application parameters. A typical example of self-optimization control is the concept of Pareto-optimal control based on the so-called approach of Pareto sets [31], which has been listed as an example included in the category of self-optimization control in Figure 2.1. A block diagram of Pareto-optimal control is illustrated in Figure 2.5.

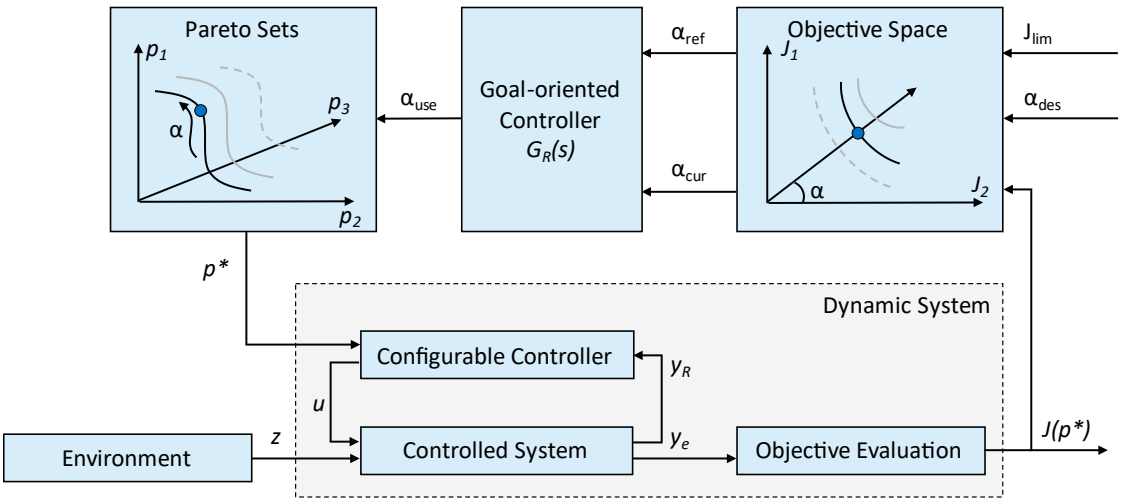


Figure 2.5: Block Diagram of Pareto-Optimal Control [11]

In Figure 2.5, it is indicated that Pareto-optimal control has a two-layered hierarchical architecture, including two feedback control loops, on which a goal-oriented controller

and a configurable controller are deployed. The lower layer focuses on realizing the adaptation of the configurable controller's parameters and structure so that the controller can guarantee the stability of the system and its expected functionalities.

Unlike the lower layer, the higher layer in the hierarchical architecture, including the goal-oriented controller, aims to figure out the optimal parameter setup for the (re-)configurable controller on the lower layer without solving the multi-objective optimization problem at runtime during the system operation and thus is different from the MPC [11]. Instead of the online optimization at runtime, characteristic diagrams illustrated as Pareto fronts are defined and deployed in the components *Pareto Sets* and *Objective Space* in advance. The feedback control loops on the higher and lower layers are discussed in detail in the following sections.

2.1.3.1 Feedback Control Loop on Higher Layer

As illustrated in Figure 2.5, unknown environmental disturbances (z) influence the system's output (y_e) and subsequently influence the output of a predefined cost function ($J(p^*)$). This cost function is used to evaluate the fulfillment of the control objective. The parameter p^* stands for the current configuration of the configurable controller (e.g., the value setup of the controller's application parameters). Due to the mentioned multi-objective optimization, the cost function is defined as a vector consisting of a set of sub-functions in parallel, which can respectively provide corresponding evaluation results ($J_1(p^*), J_2(p^*), \ldots, J_n(p^*)$) from the perspectives of different objectives.

As discussed earlier, the Pareto-optimal control needs to realize a situational adaptation of its system configuration to the optimal setup. For this purpose, the higher layer aims to figure out the optimal parameter setup for the reconfigurable controller on the lower layer in different situations, which also means that the functionalities of situation identification and corresponding mode change are required.

To realize the situation identification and corresponding mode, a relative weight (α), representing the relative importance of different objectives, is applied in the component objective space on the higher layer. Figure 2.6 shows an example of the objective space considering the case with two different objectives. The calculated evaluation results of the fulfillment of the objectives ($J_{1,cur}$, $J_{2,cur}$) are taken as inputs of the objective space.

Additionally, maximal limited values of the cost function ($J_{1,lim}$ in Figure 2.6) and a desired relative weight (α_{des}) are given in advance as constraints by external sources (e.g., predefined by the user). They are also taken as inputs of the objective space.

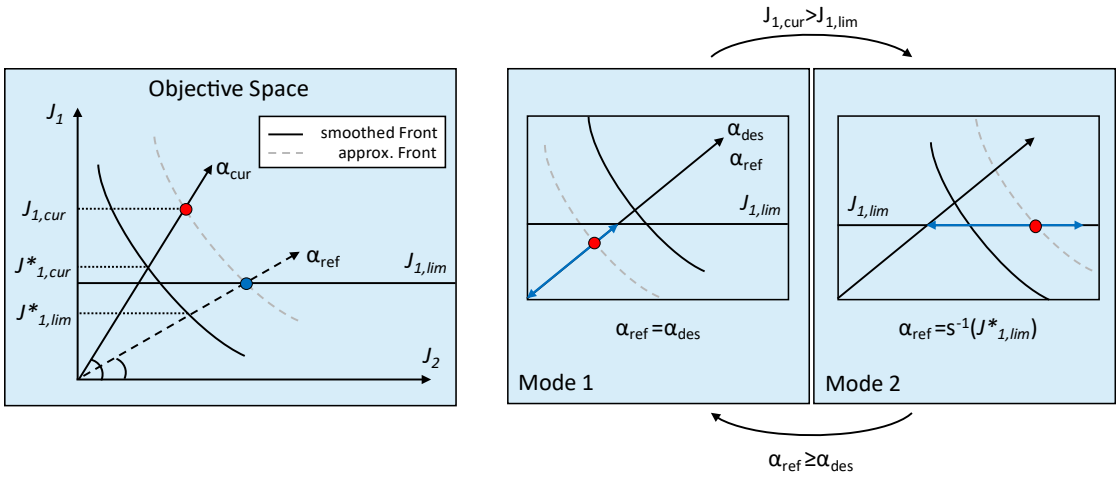


Figure 2.6: Determination of Reference Relative Weight α_{ref} (a) and Logic of Mode-Switch (b) in Objective Space [11]

Based on the cost function's current output ($J_{1,cur}$, $J_{2,cur}$), it is possible to identify a red point in the objective space, as shown in Figure 2.6. The red point is assumed to be located on an approximated Pareto front illustrated by a dotted line. Based on the position of the red point, the current relative weight (α_{cur}) can be determined, which is understood as an angle in the objective space. As presented earlier, characteristic diagrams referring to the smoothed Pareto fronts in the diagram (a) of Figure 2.6 are predefined in the objective space. Thus, a target value ($J^*_{1,cur}$) located on the smoothed Pareto front can be found based on the identified red point and the current relative weight (α_{cur}).

In the case of great environmental disturbances (z), the current value of the cost function would become very high to work against the disturbances. The determined target value ($J^*_{1,cur}$) with current relative weight would become higher and violate the constraint of the limited value ($J_{1,lim}$), which triggers the objective space to switch into Mode 2. In this case, the current relative weight cannot be taken as the referenced relative weight (α_{ref}).

Thus, it is required to determine a new referenced relative weight, which is under the constraint regarding the limited value of the cost function. For this purpose, a value of $J^*_{1,lim}$ for determining the referenced relative weight located on the smoothed Pareto front as shown in diagram (a) of Figure 2.6, can be calculated following Equation (2.10).

$$J^*_{1,lim} = J_{1,lim} \frac{J^*_{1,cur}}{J_{1,cur}} \tag{2.10}$$

Subsequently, the referenced relative weight (α_{ref}) under the constraint of the maximal limited value of the cost function ($J^*_{1,lim}$) can be determined as follows:

$$\alpha_{ref} = f(J^*_{1,lim}) \tag{2.11}$$

If the environmental disturbances (z) become weaker, the current value of the cost function ($J_{1,cur}$) will be lower than the maximal limited value ($J_{1,lim}$). Thus, the referenced relative weight will equal or be greater than the desired relative weight. In this case, the current relative weight can be directly taken as the referenced relative weight, and the objective space will switch into Mode 1. With such a switch mechanism, the mentioned situational change of operating modes can be realized by corresponding trigger conditions, as shown in diagram (b) of Figure 2.6.

Based on the calculated current and referenced relative weights from the objective space as inputs, the goal-oriented controller (e.g., one possibly designed as a linear PI controller) determines a current relative weight (α_{use}). In this case, the optimal candidate setup of the controller parameters (p^*) can be selected from the Pareto sets to adapt the parametrization of the configurable controller on the lower layer. It is emphasized that the reliability of the parametrization solution is not explicitly considered during the selection of the optimal controller parameters. Instead, it is guaranteed by the previously defined value range of the candidate controller parametrizations included in the Pareto sets, with additional consideration of fulfillment of the stability criteria of the feedback control loop on the higher layer [11].

Subsequently, the controller parametrization setup will be forwarded into the configurable controller ($G_R(s)$) on the lower layer. This forwarding activity is triggered by the mode change on the higher layer. An individual Pareto set is activated for each

operating mode to identify the corresponding optimal parametrization setup. In this case, the stability of the feedback control loop on the lower layer as a reconfigurable system is not considered. Thus, the stability of individual systems on the lower layer after each reconfiguration must still be proven [11]. Generally, it can be understood that the guarantee of the system stability on the lower layer is wholly delegated to the lower layer itself in the Pareto-optimal control system.

2.1.3.2 Feedback Control Loop on Lower Layer

As discussed earlier, the feedback control loop on the lower layer in the Pareto-optimal control is responsible for realizing the system's expected control functionality and stability, which refers to the feedback control loop on the lower layer, including the configurable controller. In this case, reconfiguration of the controller is an effective technique in fault-tolerant control systems to maintain stability and functionality [32].

An approach for reconfiguring a configurable controller with a stability guarantee of the fault-tolerant control system is introduced by Trächtler and Gausemeier within a critical example of a broken actuator [11]. In this approach, the adaptation of the control structure or the controller parametrization can be performed to eliminate the negative influence of the broken actuator.

For example, in the case of a linear modeled system, differential equations[3] can be formulated as follows to describe the dynamic processes of the controlled system:

$$\dot{x_f} = Ax_f + B_f u_f + Ez \tag{2.12}$$

$$y_f = Cx_f \tag{2.13}$$

In this case, the failure of an actuator influences the input matrix (B_f), which causes a column of the input matrix to consist entirely of zeros. In contrast, the system matrix (A) stays the same without influence. In this case, it is required to keep the trajectory of the system state (x_f) in compliance with Equation (2.14) to realize the expected controller

[3] The subscript *f* refers to the dynamic processes in the controlled system with failures.

reconfiguration with a guarantee of system stability. A reconfiguration matrix *K* is taken as the solution to Equation (2.14), added to the input matrix as stated in Equation (2.15).

$$B_f u_f = Bu \tag{2.14}$$

$$B_f K = B \tag{2.15}$$

Thus, the reconfiguration matrix *K* can be integrated between the nominal controller and the faulty model of the controlled system, as shown in the following Equation (2.16):

$$u_f = Ku \tag{2.16}$$

The control vector u_f includes the reconfigured desired values of the operative actuators [32], which can compensate for the negative influences caused by the failed actuator and let the nominal controller remain unchanged in the reconfigured control loop.

2.2 Architecture Design of Software-Intensive Systems

A taxonomy of different control concepts in the research field of control theory proposed by Trächtler and Gausemeier [11], including basic control, optimal and adaptive control, and self-optimization control, was presented in previous sections. Based on this taxonomy, it is clear that the complexity of control systems is continuously growing.

Faced with increasing system complexity, traditional design approaches of control theory that focus on modeling the controlled dynamic processes and deriving accompanying functions becomes insufficient. Instead, the architecture design approaches of software-intensive systems [15] become crucial for managing increasing complexity. Clear proof of such a trend can be found in the previously presented self-optimization control, constructed with a naive hierarchical software architecture by simply superposing two concurrent closed control loops. In the following sections, related works about architecture design of different software-intensive systems will be discussed in detail, with a particular focus on intelligent control systems in control theory [33]–[35] and the self-adaptive systems in software engineering [25][36].

2.2.1 Tree-Structured Architecture of Saridis

The most famous hierarchical architecture is proposed by Saridis [1], focusing on the design of hierarchical control systems by following a principle of so-called increasing intelligence with decreasing precision, or increasing precision with decreasing intelligence (IPDI). The phrase increasing intelligence refers to the highly symbolic methods and fewer numeric-algorithmic methods utilized on higher than on lower levels, along with degrees of abstraction. Thus, the highest levels can directly interact with a human user (e.g., with the help of an expert system). The phrase decreasing precision means that the higher levels focus on the plan by considering a larger contextual horizon with more information (e.g., a longer time horizon with a lower interaction frequency with the physical world), which also means a lower system or decision rate [37].

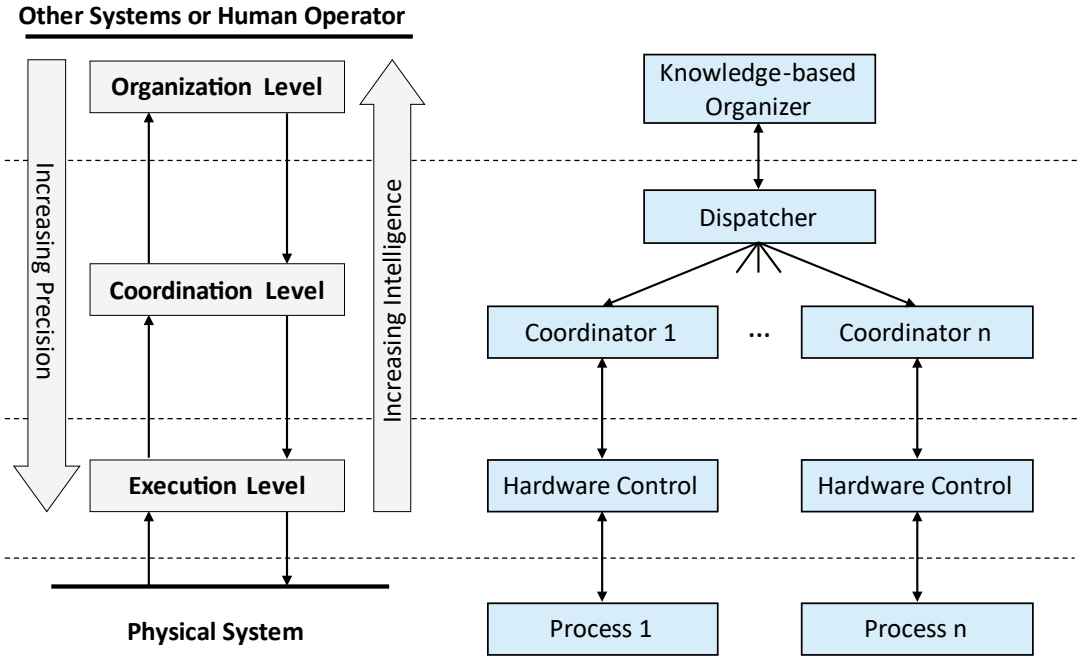


Figure 2.7: Tree-Structured Architecture of Saridis [1][2]

Following the IPDI principle, the control system is hierarchically decomposed into different levels of control (e.g., the levels of organization, coordination, and execution, which are also presented by Saridis in his work, as shown in Figure 2.7) [2]. Along with the different levels of control, the high-level control task is hierarchically decomposed into distinct subtasks on the next level down. A successive delegation of duties exists

from the upper to lower levels. The number of distinct tasks continually increases from the top to down, with an accompanying decrease in the hierarchical levels [37].

In this case, the whole control system can be seen as an integrated unit consisting of mathematical and linguistic methods and algorithms applied to corresponding subsystems and processes. This approach separates the control system into different levels, following the IPDI principle. Thus, the system may contain more than one layer of tree-structured functions, including components like knowledge-based organizers, dispatchers, corresponding coordinators, hardware controls, and the physical system. The physical system includes a set of dynamic processes in the controlled plant and its environment (cf. Figure 2.7).

The level of organization performs operations like planning and high-level decision-marking based on long-term memory with high-level information processing such as the probabilistic model or knowledge-based system using artificial intelligence algorithms. For example, the learning of large quantities of knowledge would be performed on this level, which can be deployed in the knowledge-based organizer. The coordination level as an intermediate structure works like an interface between the organization and execution level, which includes decision-marking and learning in short-term memory. Due to high-level control task decomposition, multiple coordinators responsible for the decomposed independent subtasks can be deployed on the coordination level. Subsequently, the execution level involving a set of hardware controllers is then responsible for the fundamental control functions (e.g., feedback control loops in other control concepts with lower levels of intelligence).

2.2.2 NASREM Reference Model for Telerobot Control System Architecture

NASREM is another famous hierarchical reference architecture of the real-time control system for applications like robots and intelligent machines developed by the National Institute of Standards and Technology (NIST) of the United States [3]. In the NASREM reference model, the control system is designed as a three-legged hierarchy of computing modules serviced by a communication system and a global memory, as shown in Figure 2.8.

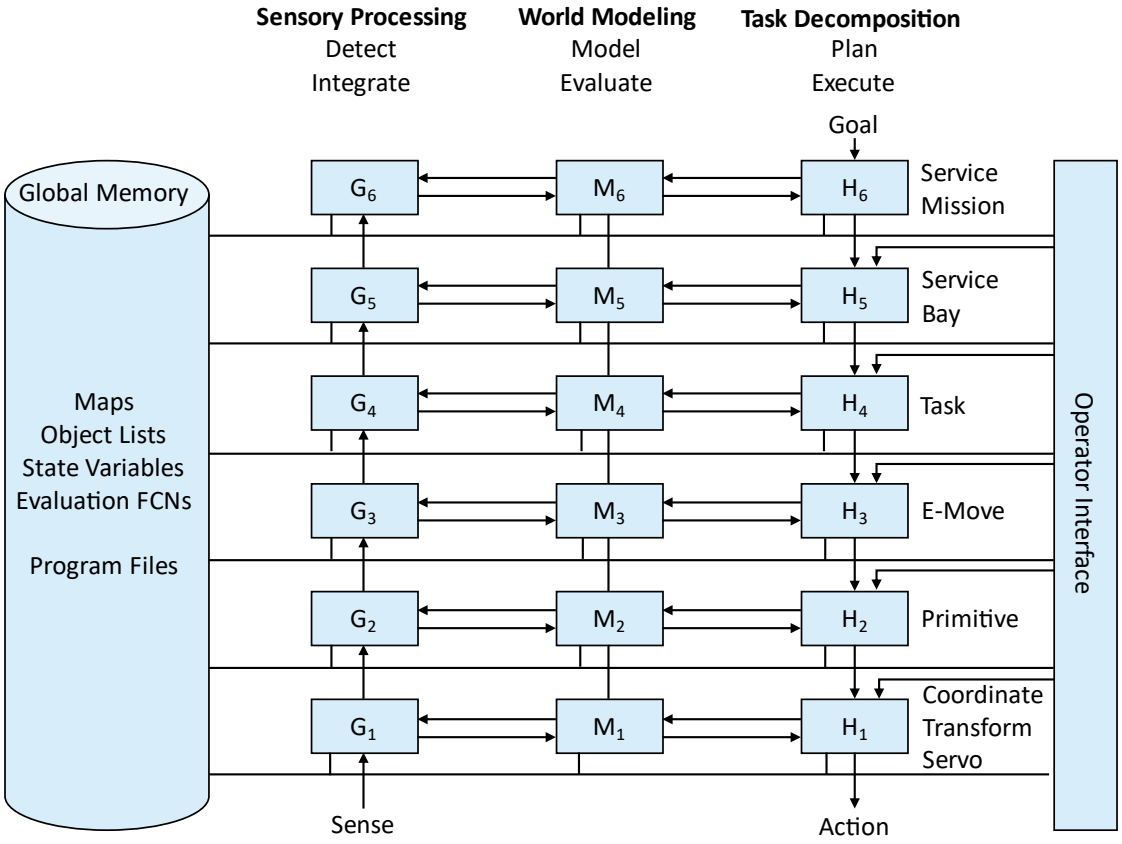


Figure 2.8: NASA/NBS Standard Reference Architecture for Telerobot Control System [3]

The first leg of the hierarchy, *Task Decomposition*, is responsible for the real-time planning and monitoring of the task, which relies on the modules H_x to plan and execute the spatial and temporal decomposition of high-level goals into low-level actions. In this case, spatial decomposition refers to the task division as concurrent actions by corresponding subsystems. Temporal decomposition means that the task is divided over time into sequential actions. Each task decomposition module at each level includes a job assignment manager and a set of planners and executors.

The second leg of the hierarchy is *World Modeling*, which consists of the modules M_x for modeling and evaluating the world, focusing on its historic, current, and possible future states, including the states of the controlled system. For this purpose, the modules of world modeling work together with a knowledge base involved in the global memory (cf. Figure 2.8), in which maps, the lists of objects, events and their attributes, and the state variables are included. Based on the provided information of observed facts from

the sensory processing modules (G_x), the world model maintains the knowledge base of the global memory and delivers its predictions of expected sensory data back into the sensory processing modules. Additionally, the world model interacts with the planner and executor deployed in the task decomposition module at each level and tries to answer questions like “What is?” and “What if?” [3].

The third leg of the hierarchy, *Sensory Processing*, consists of the modules G_x and is responsible for pattern recognition and event detection through checking correlations and differences between the world model’s predictions and observed facts of sensory data. Additionally, the processes of sensory data processing like filtering and integration are included in these modules. Thus, newly detected or recognized events, objects, and relationships will be integrated into the database of the global memory, and objects or relationships that no longer exist will be removed [3]. The confidence factors and probabilities of the identified events and statistical estimates of the state variables are also computed within the modules.

In addition to the three-legged hierarchy, an operator interface is also included in the NASREM reference architecture. It interacts with the human operator to intervene in the control system at any level and time. For example, some specific interventions are monitoring a process, inserting information, interrupting automatic operation, or even taking over control of a task to realize semi-autonomous control.

Following the principle of hierarchical levels in the architecture, the range of time scale and thus the planning horizon and historical event summary interval under the consideration decrease exponentially along with the hierarchical levels from top to bottom in the NASREM architecture. At each level, the planners inside the task decomposition modules divide the task commands into strings of planned subtasks for execution. The strings of sensed events are summarized and integrated into single individual events at the next level up. In this case, each plan is constituted by at least two and, on average, ten subtasks. The planning horizon is extended to the future, considering an additional input command interval. During system operation, replanning can be triggered by an emergency condition or deterministic cycle time, which requires that the cycle time is an order of magnitude less than the planning horizon [3].

Instead, the executors inside the task decomposition modules react to feedback for every control cycle interval. Once the feedback detects the failure of any planned subtask, the executor will immediately launch a preplanned emergency subtask. The planner will generate an error recovery sequence to replace the previous failed plan.

In Figure 2.8, it is indicated that the lowest level is called a coordinate transform servo. This level can be called the servo level for short. Its design relies on the technology of basic feedback control. The primitive level is designed to generate a sequence of command specifications, which is a kind of trajectory and taken as input for the servo level. Once the planner at the servo level receives a new command specification, it transmits the information about an attention function to the world model so that the world model knows where to concentrate its efforts. The world model estimates the state of the manipulator (e.g., including values for variables like position, velocities, and joint torques). While the executors perform the specified commands, relevant information will also be transmitted to the sensory processing modules at the primitive level to monitor the trajectory execution [38].

2.2.3 Nested Hierarchical Architecture of Meystel

Saridis' tree-structured architecture was presented in the previous section. In this architecture, the high-level control task is derived as low-level subtasks, which different concurrent coordinators and hardware controls can perform. The Saridis' architecture [1][2] focuses more on the multi-actuator control system, consisting of a tree-structured hierarchy including different levels of intelligence. On different levels, several units for decision-making are deployed. By coordinating the actions of these units, Saridis' architecture [1][2] optimizes the process of goal achievement [39].

Unlike Saridis' architecture, Meystel's [39] work has proposed another nested architecture for intelligent control, which focuses on the case of single actuator systems instead of on multi-actuator systems, especially for the use cases of autonomous control systems without any human involvement. Unlike the tree-structured architecture in Saridis' concept, the control system is designed in Meystel's concept as a nested hierarchy consisting of layers with different resolutions [5][18], as shown in Figure 2.9. In this case, each layer corresponds to an individual resolution level.

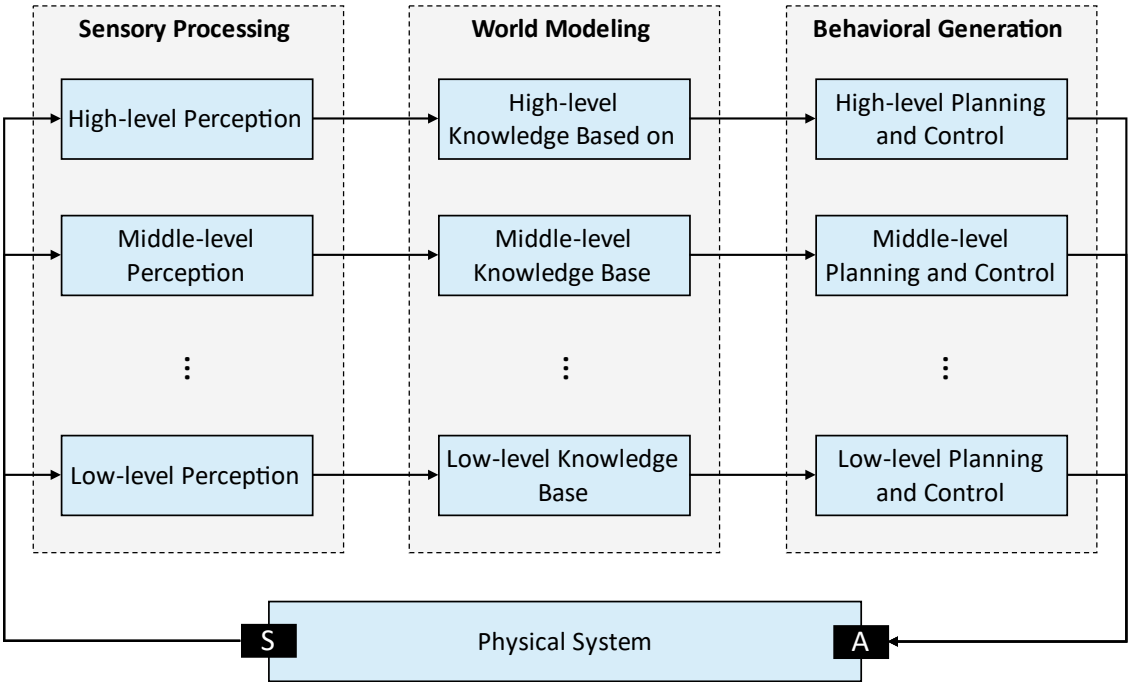


Figure 2.9: Nested Hierarchical Architecture of Meystel [18][39][40]

In Meystel's architecture, computing processes are independently distributed on the hierarchical layers, on each of which a feedback control loop is deployed. Thus, each layer represents a different domain of the overall system. The loops on the upper and lower layers correspond to each other. Thus, the system behavior results from a superposition of the actions on every resolution level generated by similar algorithms. The hierarchy of knowledge representations evolves from linguistic at the top level to analytical at the bottom level. It is emphasized that the knowledge bases here are relatively independent but can communicate to realize a knowledge exchange [5].

A typical application example of Meystel's architecture is the intelligent control system for an autonomous mobile robot, which usually includes components of planning and control at four levels instead of three levels, also called planner, navigator, pilot, and execution. The planner focuses on finding and carrying out a rough plan consisting of time profiles of the input variables, which are used to guarantee the expected time profile of the output variable. The navigator refines the initial plan and plans a more concrete motion trajectory. The pilot is used to realize online motion control tracking, considering deviation between the expected situation in the plan and the current local surrounding situation observed by onboard sensors. Finally, the execution level is responsible for the

execution and compensation of the plan delivered by the planner, the navigator, and the pilot [39].

2.2.4 Behavior-Based Subsumption Architecture of Brooks

In the previously presented architectures, the control task is decomposed into subtasks on different levels of abstraction. Such a decomposition is realized by a series of vertical slices, including sequenced functional modules (e.g., perception, modeling, planning, and execution, deployed on concurrent layers in the architecture; cf. Figure 2.9) [41]. Thus, the slices form a chain to build up a closed feedback loop with information flows at different levels.

Unlike such an approach with so-called vertical decomposition, another principle focuses on horizontally decomposing the problem into task-achieving behaviors. Each behavior means a mapping from the sensory inputs to a pattern of actuator outputs that aim to complete certain tasks. Thus, the whole control system is designed as a reflex system. The perception and the action are tightly coupled within the behavior without using abstract representation or temporal planning, which is an approach very similar to the current end-to-end learning system for autonomous driving [42].

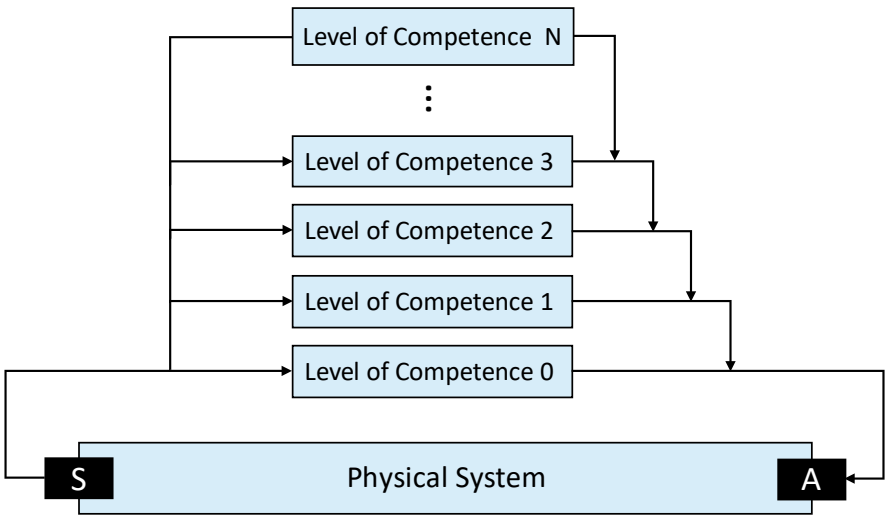


Figure 2.10: Behavior-Based Layered Subsumption Architecture by Brooks [41]

The problem decomposition in the behavior-based architecture performs based on desired external manifestations of the control system. A typical example of behavior-

based architecture following such a decomposition principle is the subsumption architecture developed by Brooks [41] for applications in autonomous robots, as illustrated in Figure 2.10. In Brooks' architecture, different layers represent so-called levels of competence. Each level of competence represents an informal specification of the desired class of valid behaviors for the autonomous robot in all potential operating environments. In the system architecture, the level of competence is represented by a single layer.

Compared to the lower-level layers, higher-level layers with higher levels of competence imply more specific desired classes of behaviors with inhibition mechanisms. They can subsume the roles of lower levels by suppressing their outputs. However, lower levels continue to function as higher levels are added [41]. Each level of competence includes its lower levels of competence as a subset. As presented earlier, a level of competence defines the desired class of valid behaviors. Thus, the level of competence above it can also be understood as an additional constraint of the valid behavior class.

2.2.5 LAAS Architecture of Alami

Another well-known architecture concept was developed by Alami et al. [43] in their research work, called the LAAS architecture due to the name of their laboratory. In LAAS architecture, the control system comprises three levels: a decision level, an execution control level, and a functional level, as shown in Figure 2.11. The logical system is designed as an independent logical level, which works as an interface between the technical and physical system to make the functional level as hardware-independent as possible.

In LAAS architecture, the decision level is designed to take charge of the high-level decision-making with the requirement of deliberative capability and reaction to incoming events. For this purpose, the decision level is designed as a system triggered by goal and event, which includes a multi-layered structure, depending on concrete applications. The planners and supervisors are deployed on different layers, as shown in Figure 2.11.

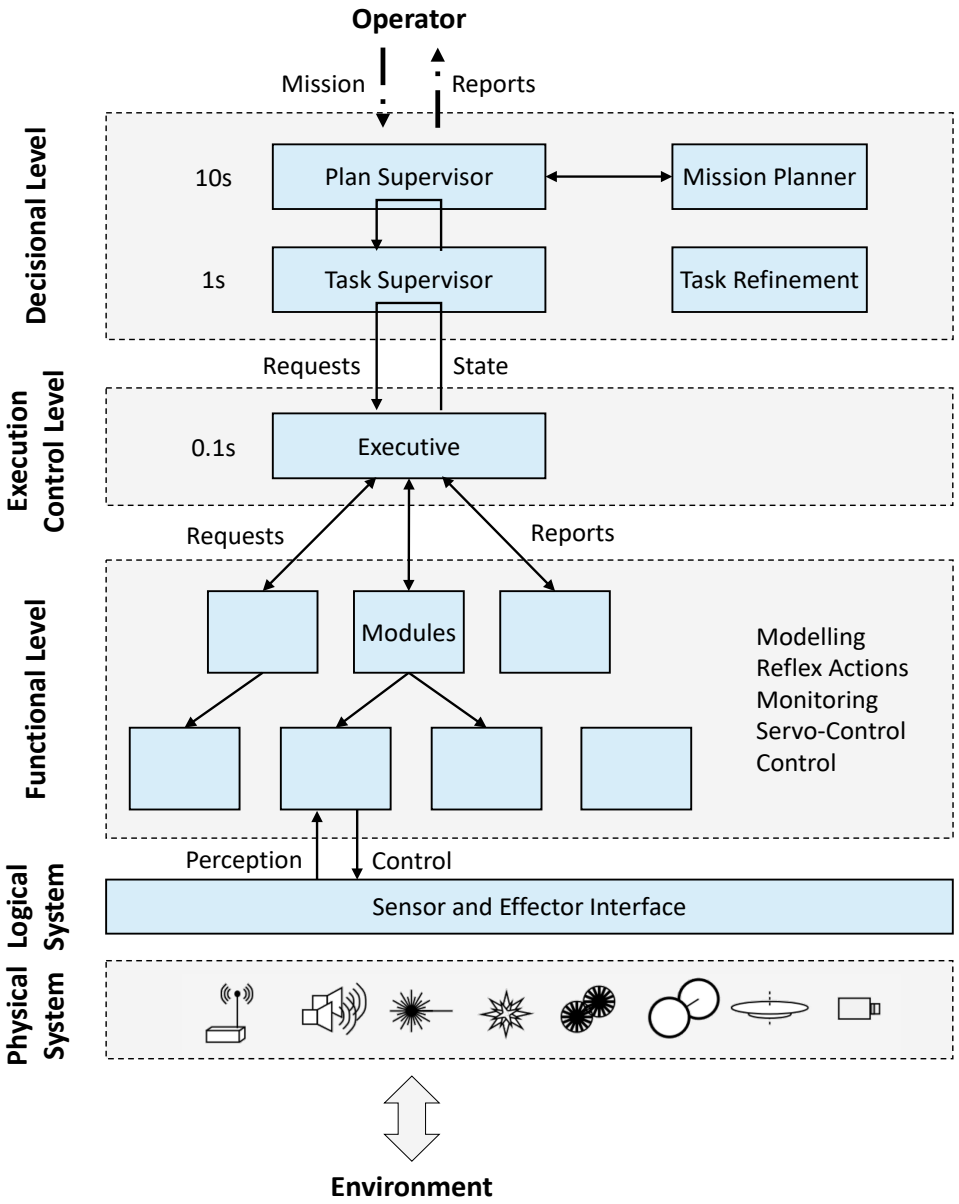


Figure 2.11: Reference Structure of LAAS Architecture [43]

The execution control level plays a role as the interface between the time-consuming symbolic processing at the decision level and numerical data computation with a high frequency at the functional level. It is designed as a purely reactive system without considering the predictive horizon. Additionally, it takes the decided sequence of actions as input to correspondingly select, parametrize, and synchronize the appropriate functions at the lower functional level, depending on the task and current state of the

system, which is determined based on submitted requests and returned replies from the functional level. It is emphasized that replies can trigger requests delayed by the *Executive* component. As output, a report about the current state will be submitted to the upper decision level to enable plan supervision and choice of subsequent actions [43]. The executive component takes over the responsibility of control and data flow simultaneously.

Generally, the functional level can be seen as a library of functions that activate elementary robot actions, task-oriented activities like motion planning, vision, and localization, or reflexes relying on predefined condition-reaction-policies. These functions are embedded in a set of modules. These modules are taken as the fundamental unit at the functional level to build dynamic networked interactions, depending on the task being executed and the environmental state. Communication between the modules is built based on the so-called request-response pattern, relying on the call of specified services provided by server modules to corresponding client modules (cf. Section 2.3.1). In this case, the server module does not know its clients in advance. Instead, the relationship between the client and server modules is established dynamically, which means that the modules can be arbitrarily included or removed from the system. Additionally, it is also permitted that the services of one module are used by other modules, relying on the design of advanced services by combining primitive services, which are general and reusable.

As discussed earlier, the functional level is designed with the principle of the distributed system. In this case, the malfunction of any modules may lead to a failure of the whole system. Thus, each module is designed to check its request parameters' validity and applicability of the required actions. As a critical case, concurrent activities may require the same shared resource and thus cause a conflict of resource allocation. Thus, the latest request is always defined to have higher priority than previous requests to eliminate this critical case.

2.2.6 Hybrid Control Architecture of Yavuz and Bradshaw

A further well-known architecture approach to the control system is the hybrid control architecture proposed by Yavuz and Bradshaw [44]. They have argued that the

capability to work against uncertainty is one of the leading design constraints for the real-time control system in different applications such as mobile robotics, mainly due to incomplete and time-variant prior knowledge about the environment. Thus, the control system is required to complete its reasoning based on current information about the state of the controlled system. It is also crucial that the control system quickly and appropriately responds to an unexpected event. These expected properties require a vital adaptability function and a quick reaction capability for the control systems. Thus, a point for the system design was made by Yavuz and Bradshaw: the adaptability function must be based on the reaction behaviors [44]. Following this point, a concept of hybrid hierarchical architecture was proposed, as shown in Figure 2.12.

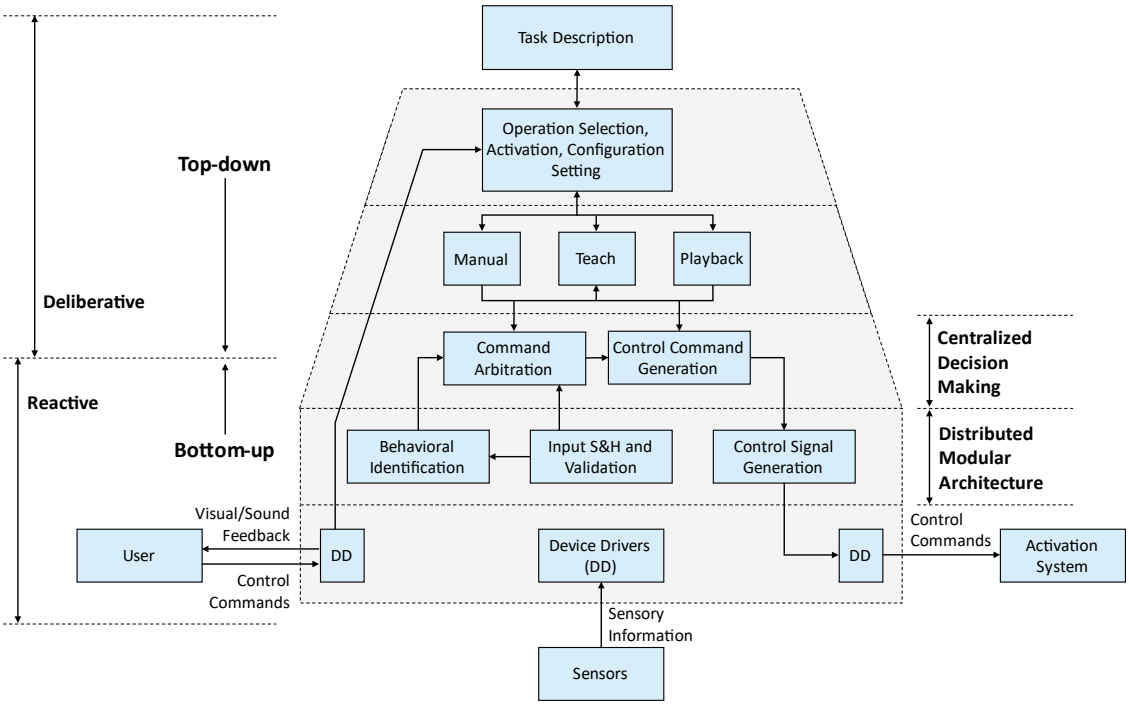


Figure 2.12: Hybrid Control Architecture for Mobile Robot [44]

System parts following different principles are systematically integrated into the hybrid hierarchical architecture. On the top of the hierarchical architecture, deliberative modules are deployed on upper layers, which are responsible for high-level planning and decision-making, relying on the benefits of included world models. In this case, the control task is hierarchically decomposed into different levels of abstraction. Thus, each

level controls the level beneath it and assumes that its commands will be executed as anticipated [44], following a so-called principle of top-down manner.

Unfortunately, it is utopic that the commands can always be precisely performed as expected, particularly in the case of unstructured and dynamic environments. In addition, the deliberative modules rely on a complicated world model to complete the planning tasks, which causes time delay due to extensive computation effort and thus possibly violates the required time constraints of the control system within a concrete application like a mobile robot. Thus, reactive modules are deployed at the bottom of the hierarchical architecture instead of the deliberative modules. These reactive modules are responsible for reflection of and response to environmental stimuli, relying on compiled procedural knowledge without high-level knowledge in the world model [44].

In contrast to the top-down manner, the control task in the lower part of the hierarchical architecture is horizontally decomposed into independent behaviors. Due to the independencies between the behaviors, the control system can be developed in a bottom-up, evolutionary manner. In this case, each behavior only concerns its relevant context information and thus does not require a complicated world model. Relying on sensing at a rate high enough for impact limitation of the false sensory readings to work against the uncertainty in perception, the mentioned disadvantage of the deliberative modules thus can be compensated for [44]. Additionally, due to the loose coupling of the deliberative modules and the physical system by isolation of reactive modules, the presented critical point about time constraints can also be eliminated.

Since the architecture of Yavuz and Bradshaw is mainly designed for a mobile robot, several operating modes, including manual, teaching, and playback, are included in their concept. The manual mode refers to teleoperation entirely controlled by the human operator. In mobile robot applications, a map with high complexity is usually required for the deliberative modules to plan the path for the robot. In the approach of Yavuz and Bradshaw, a simple sequential task-information set containing simple steps of a navigational pattern that direct the robot towards the goal (or the achievement of the task pattern) is applied to replace the complicated map. Thus, the robot samples and records the task pattern in the teaching mode, solving the task-information source problem. The playback mode is designed for the self-supervised goal-oriented

autonomous operation of the robot. In this mode, the robot evaluates the inputs from the user, deliberative and reactive modules, and assigns priorities to the requested actions [44].

In this case, the information from different sources with different format (e.g., the task plan step from the deliberative modules, the reflexive behaviors from the reactive modules, and the control commands from the user) must be integrated. The component *Device Drivers* are designed to preprocess the collected raw sensory data, relying on its embedded specific sensor-information-analysis modules. After the preprocessing, each input set is evaluated, decoded, or converted into standard behaviors with specific motion control and activation settings [44]. The component *Command Arbitration* is designed to integrate the sets of information and generate the control commands. The deliberative modules may cause a time delay in response due to intensive computation. Thus, the generated control command is checked against the sensory information before execution to compensate for the time delay and potentially harmful influences. A bypass technology is also utilized in the architecture, which means that the reflexive-control-action requests with a high priority can bypass the control command process and attend to the urgent reflexive behavior control requests [44].

Generally, the presented architecture aims to design a goal-oriented, reactive, and teleoperable system. The design of modular reactive modules in the lower part of the architecture guarantees high flexibility for further development of the system, while the application of the mentioned bypass technology compensates for the disadvantages of the deliberative modules regarding time delay of response. The overall sensory processing is distributed to low-level reactive modules, allowing the developed system to have sensor-specific processing to guarantee its responsiveness, robustness, and flexibility to lower levels. From another perspective, decision-making in centralized modules of the developed system guarantees the system’s straightforwardness, modularity, and efficiency [44]. The operation of deliberative modules follows the top-down approach, while the reactive modules follow the bottom-up approach. The centralized decision-making integrates both modules for the generation of control commands. Subsequently, the generated control commands are forwarded to the activation system.

2.2.7 IBM's MAPE-K for Autonomic Computing

The control systems aim to keep the current value of the controlled variable as a given reference value by adapting the value of the manipulated variable, respectively, depending on the current state of the control error. From the perspective of software engineering, such an adaptation feature can be interpreted as a kind of self-adaptation activity, which is the most significant feature of self-adaptive systems. For this reason, the software engineering approach for designing self-adaptive systems is also considered in this dissertation and taken as a reference to investigate the design of future control system architectures with increasingly high complexity. Since MAPE-K is one of the most well-known architecture concepts for self-adaptive systems in software engineering, it is investigated in this dissertation. In the following sections, it will be presented in detail.

2.2.7.1 Autonomic Computing as a Vision of Self-Adaptive System

In 2001, IBM reported that the crisis due to the increasing complexity of future software systems had become one of the most significant constraints in the further development of the IT industry [45]. This development trend requires millions of well-skilled IT professionals to develop and maintain highly complicated systems. Additionally, it constitutes a significant challenge for integrating heterogeneous systems and the internet in the future. Thus, current innovations that solely rely on programming languages, which have extended the size and complexity of the systems for architects to design, become insufficient.

Concept	Current Computing	Autonomic Computing
Self-Configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time-consuming and error-prone.	Automated configuration of components and systems follows high-level policies. The rest of the system adjusts automatically and seamlessly.
Self-Optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their performance and efficiency.
Self-Healing	In large systems with high complexity, problem determination can take a team of programmers weeks.	The system automatically detects, diagnoses, and repairs localized software and hardware problems.

Self-Protection	Detection of and recovery from attacks and cascading failures is manual.	The system automatically defends against malicious attacks and cascading failures. It uses early warning to anticipate and prevent system-wide failures.

Table 2.1: Autonomic Computing for Strengthening Self-X Properties [45]

In this case, the only remaining solution approach is autonomic computing inspired by the natural biological nervous system that enables self-management for high-level objectives provided by administrators to software systems. Thus, the software systems become self-managing or so-called self-adaptive systems [45]. To make the mentioned self-management clearer, IBM has defined four properties to specify it, including self-configuration, self-optimization, self-healing, and self-protection, summarized as the well-known self-X properties as described in Table 2.1.

Additionally, IBM has developed the autonomic computing adoption model, as shown in Figure 2.13, referring to a methodology for businesses to calibrate the degree of autonomic capability based on the following dimensions: increased functionality, control scope, and service flow. The dimension of functionality describes the automation degree of the IT and business processes with five levels. The first *Manual* level means that the IT professionals are manually responsible for system management. Different system management technologies can be applied at the second level of *Instrument and Monitor* to collect the information from the managed resources. Thus, some administration tasks like monitoring can be eliminated for human administrators. The third level is named *Analysis*, which means that the management functions like pattern recognition, prediction of optimal configuration, and recommendation of corresponding actions are automated instead of being delegated to human administrators. Subsequently, at the level of *Closed Loop*, the management system performs the actions automatically, relying on its available information and knowledge about the managed resources. Finally, the IT systems can understand the high-level objectives and business policies at the final level of *Closed Loop with Business Priorities*. Users interact with the autonomic technology tools to monitor business processes and alter the business processes or objectives [46].

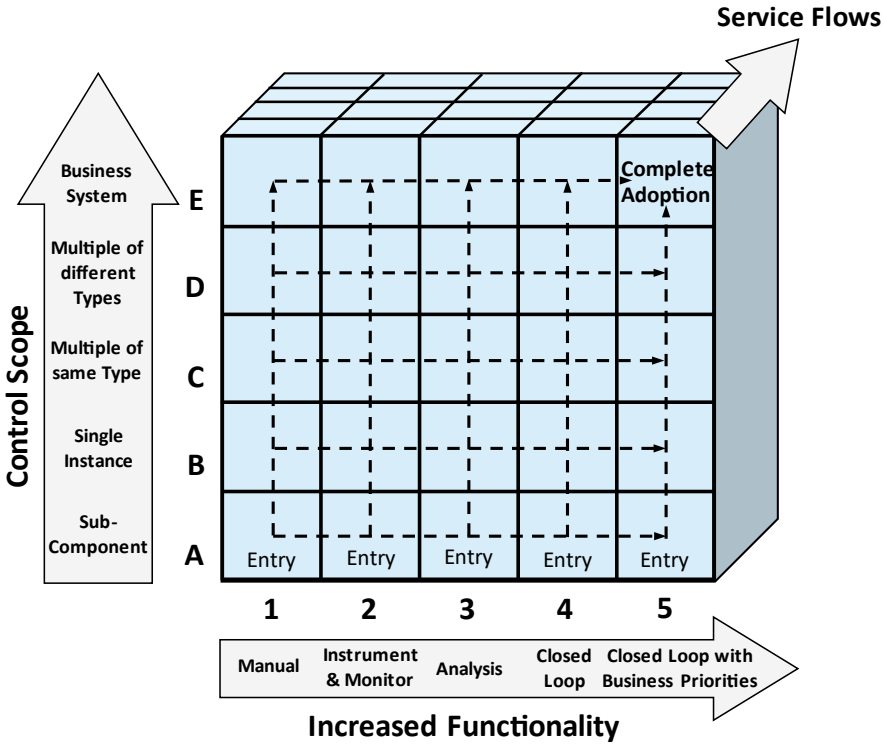


Figure 2.13: The Autonomic Computing Adoption Model [46]

In addition to functionality, the second dimension is the control scope, which refers to coverage of the managed resources. At the first level of the *Sub-Component*, the technical system focuses on the partial system (an operating system on a server or an individual application within an application server) [46]. The second level is the *Single Instance*, in which case the whole standalone resource, such as a server or application server, is automatically managed. The next level focuses on the case with *Multiple Instances of the Same Type* as the managed resources, which means that the managed resources are homogeneous (e.g., a cluster of application servers). Once the managed resources consist of *Multiple Instances of different Types*, such a use case is categorized as the fourth level, which means that heterogeneous resources like servers and databases or routers and storage units are focused on. Finally, the hardware and software resources, which perform the business processes, are automatically managed by the systems at the *Business System* level.

The third dimension of service flow focuses on the so-called autonomic maturity level by combining IT management process activities such as change management, incident

management, and problem management. Autonomic maturity can evolve in three dimensions: (1) automating more functions along with the increase in maturity, (2) applying automated functions to broader resource scope, and (3) automating tasks and activities in various IT management processes [46]. Along with the increasingly higher levels of autonomic maturity, automation is applied, in which case more processes of the managed systems are focused on within an increasingly broader scope.

The developed adoption model enables the incremental adoption of additional autonomic capabilities for the evolution of autonomic computing. Thus, a solution space is provided for the business to determine an incremental action plan for maximizing the benefits of corresponding available autonomic capabilities. To realize autonomic computing, the topic of self-adaptive systems, which are capable of monitoring their operating environments and automatically adapting themselves to changes with the help of the mentioned self-X properties (cf. Table 2.1), has been studied by different researchers [36][47][48]. In this dissertation, MAPE-K, proposed by IBM as one of the most fundamental architecture concepts, is taken as a reference and will be discussed in detail in the next section.

2.2.7.2 Overall Reference Architecture for Autonomic Computing

As discussed earlier, autonomic computing requires the self-management capability of the software systems, which makes them self-adaptive systems. To design such systems, IBM proposed a reference architecture to describe the overall idea for the design of the MAPE-K concept, which is shown in Figure 2.14. This reference architecture consists of multiple layers, with corresponding so-called autonomic computing building blocks, which communicate by relying on the communication pattern of the enterprise service bus (e.g., via standard mechanisms like web services) [46]. Thus, the autonomic computing system is decomposed by building blocks (on different layers) within different classes: touchpoints, knowledge sources, orchestrating and touchpoint autonomic managers, and manual managers.

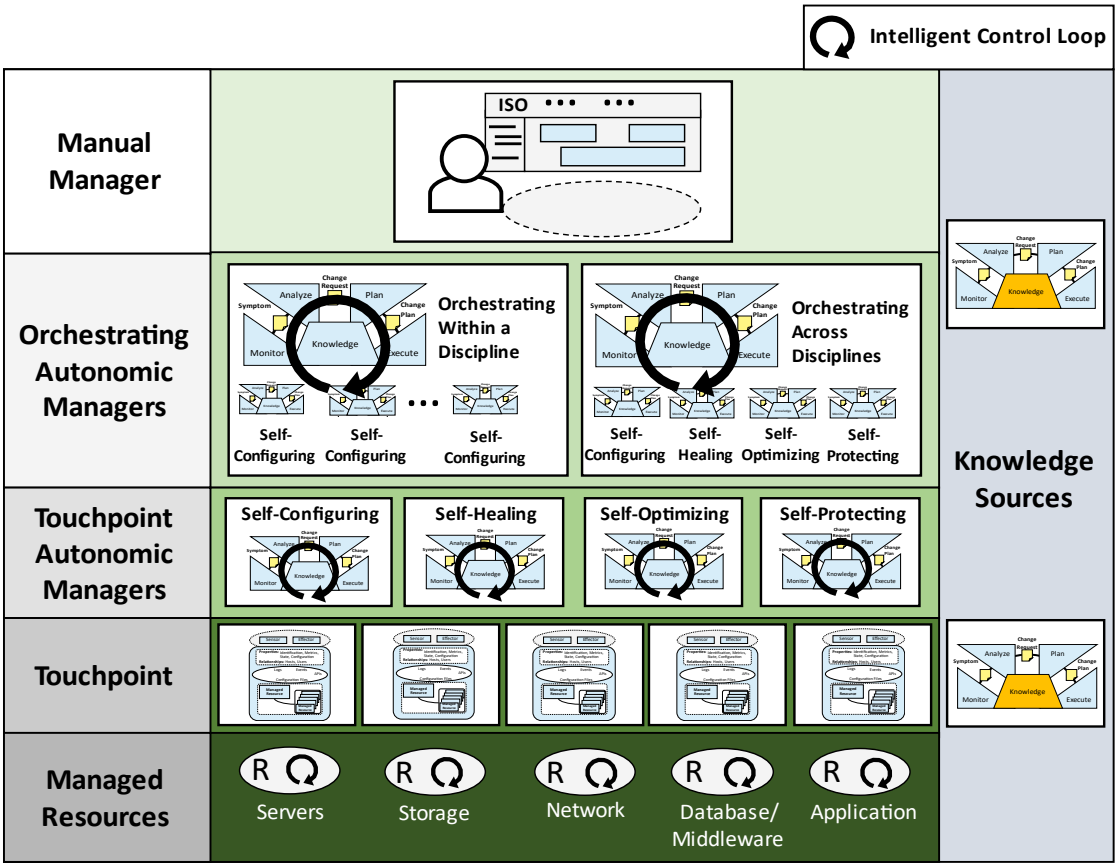


Figure 2.14: Reference Architecture of Autonomic Computing [46]

The managed resources or system components that constitute IT infrastructures are deployed on the lowest layer of the reference architecture, possibly consisting of software or hardware like a server, storage unit, database, or application server. As shown in Figure 2.14, it is emphasized that the managed resource can also contain an embedded intelligent control loop. In this case, the loop can be used to realize the autonomic capability of self-management within the run-time environment, independently of the management by autonomic managers on higher layers. Such an embedded intelligent control loop may be deeply hidden in the managed resources, which are externally invisible and thus inaccessible. In some cases, the intelligent control loop may also be externally visible and thus can be accessed and controlled by autonomic managers via touchpoints and so-called manageability interfaces. In this case, the touchpoints are the interfaces between the autonomic managers and

managed resources. The manageability interfaces are the services provided by the managed resources, about which more details will be provided in Section 2.2.7.3.

Generally, the management task of specific managed resources can be taken over by one or multiple touchpoint autonomic managers. Each touchpoint autonomic manager can be implemented as a control loop to realize specific self-X properties. On a higher layer, orchestrating autonomic managers, each of which is also implemented as a control loop with the same structure as the touchpoint autonomic manager, are deployed to deliver system-wide autonomic capability by coordination of multiple touchpoint autonomic managers. For example, a typical orchestrating autonomic manager could be the workload manager, which, based on specific measurement approaches and policies, optimizes the resource utilization across a pool of managed resources to realize policy-based goal-oriented system management [46]. Thus, the orchestrating can be performed within a single discipline or across disciplines, depending on the self-X properties realized by the touchpoint autonomic managers, as shown in Figure 2.14.

On the top layer of the reference architecture, a manual manager provides a standard system management interface for human IT professionals to perform certain management functions like system setup and configuration or run-time monitoring and control. For this purpose, an integrated management console is used, which is primarily designed to provide a single platform. Thus, the administrator can deal with an overall management solution instead of addressing individual components. The manual manager can cooperate with other autonomic or manual managers either on the same or lower layers, enabling IT professionals to delegate management functions to autonomic managers.

The final kind of building block refers to the knowledge sources as repositories to deliver knowledge access. These knowledge sources are deployed across different layers in the reference architecture to guarantee that different manual and autonomic managers can acquire and share knowledge. The following sections will provide more details about essential building blocks in the reference architecture.

2.2.7.3 Touchpoint and Manageability Interface

As presented before, the touchpoint is an interface component that exposes the state and management operations for the resource to be managed by the autonomic manager. In this case, the autonomic manager communicates with the touchpoint through the manageability interface. Figure 2.15 illustrates a touchpoint as an implementation of the manageability interface for a specific or a set of managed resources, typically for the use case of database server management.

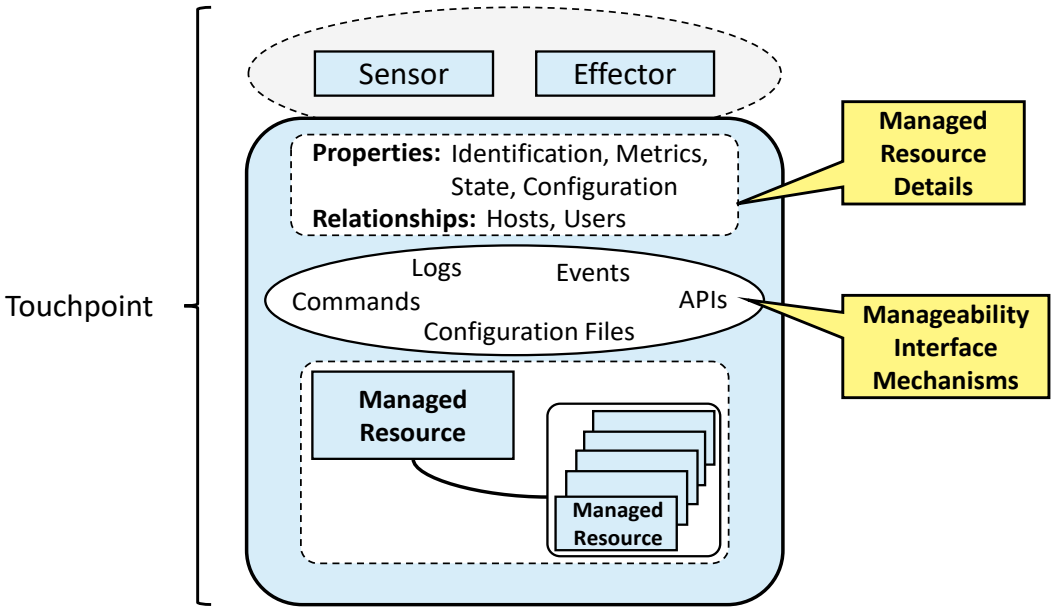


Figure 2.15: Touchpoint as Interface Between Autonomic Managers and Managed Resources [46]

The manageability interface is divided into sensor and effector interfaces to control the managed resource. The touchpoint implements sensor and effector behavior for certain specific managed resources by mapping the standard sensor and effector interfaces to available manageability interface mechanisms of the managed resources, as shown in Figure 2.15. Such an approach massively reduces implementation complexity due to the avoidance of the need to specify diverse interface mechanisms for various types of managed resources [46].

The sensor interface consists of at least one of two parts. The first part deals with accessible properties through a standard “get” operation for exposure of the information about the current state of the managed resources, relying on the interaction mechanism

of request-response. The second part refers to management events that occur in the case of a state change in the managed resource, relying on the interaction mechanism of send-notification.

The effector interface also contains at least one of two parts. The first part deals with a set of “set” operations to change the state of the managed resource, relying on the interaction mechanism of perform-operation. The second part focuses on other operations implemented by autonomic managers to enable the managed resource to send requests, relying on the interaction mechanism of solicit-response.

The sensor and the effector interface are directly linked. Such an approach works as a reflection to enable notification through the sensor interface if the effector interface changes the configuration of the managed resource. The approach relies on the so-called manageability capability of the managed resource, including a logical collection of manageable resource’s state information and operations as detailed properties (cf. Managed Resource Details in Figure 2.15).

For example, the property of identification refers to state information and operations used to identify the instance of a managed resource. Instead, the metrics refer to the state information and operations used to measure the managed resource. For different manageability capabilities, the component on the client-side is forced to be able to acquire and control state data through the manageability interface, which includes three parts: (1) meta details for specifying the managed resource and its configuration, (2) sensor interactions for retrieving of current property values from the managed resource, and (3) effector interactions for changing the state of the managed resource [46].

2.2.7.4 Knowledge Source

The previous section described how the knowledge source is deployed in the overall reference architecture as a repository to access and share knowledge. Knowledge refers to standard data shared among different building blocks, such as autonomic managers. In this case, the knowledge included in the knowledge sources can be used to extend the available knowledge of the autonomic managers. The autonomic managers can load knowledge from one or multiple knowledge sources. Further, the

knowledge can be activated by the managers of these autonomic managers on higher layers.

In the reference architecture of MAPE-K, knowledge can be acquired by following three approaches. The first approach is that the knowledge source directly passes the knowledge to the autonomic manager. The second approach refers to a case where the autonomic manager retrieves the knowledge from an external knowledge source (e.g., specific resource-specific historical knowledge, which needs to be extracted from the log files of a particular component or system). As a final approach, the autonomic manager can also create new knowledge based on current activities by logging the notifications provided by the managed resource [46]. In such a case, the autonomic manager can also update the created knowledge into the knowledge source.

To complete the management tasks, the autonomic managers require different types of knowledge, as summarized in Table 2.2. Each knowledge type must be expressed by standard syntax and semantics [46], categorized in this dissertation as homogenous knowledge. The case of heterogeneous knowledge expressed by different syntax and semantics is excluded here and is discussed in Section 6.3.2.

Knowledge Types	Comments
Solution Topology Knowledge	The solution topology knowledge captures the components' construction and configuration for a solution or business system. Installation and configuration knowledge are captured in a standard installable unit format to eliminate complexity. The plan function of an autonomic manager can use this knowledge for installation and configuration planning.
Policy Knowledge	A policy is a knowledge consulted to determine whether changes need to be made in the system. An autonomic computing system requires a uniform method for defining the policies that govern the decision-making of autonomic managers. By defining policies in a standard way, they can be shared across autonomic managers to enable entire systems to be managed by a standard set of policies.
Problem Determination Knowledge	Problem determination knowledge includes monitored data, symptoms, and decision trees. The problem determination process also may create knowledge. As the system responds to actions taken to correct problems, learned knowledge can be collected within the autonomic manager. An autonomic computing system requires a uniform method for representing problem determination knowledge, such as monitored data (standard base events), symptoms, and decision trees.

Table 2.2: Knowledge Types in Knowledge Sources [46]

2.2.7.5 Autonomic Manager Based on the Reference Model of MAPE-K

The most crucial building blocks in the presented reference architecture are the orchestrating and touchpoint autonomic managers. As discussed earlier, each autonomic manager implements an intelligent control loop. In both the orchestrating and the touchpoint autonomic manager, the intelligent control loop is realized based on a so-called MAPE-K reference model, which consists of five parts with different functions: Monitor (*M*), Analyze (*A*), Plan (*P*), Execution (*E*) and Knowledge Source (*K*), as shown in Figure 2.16 [49].

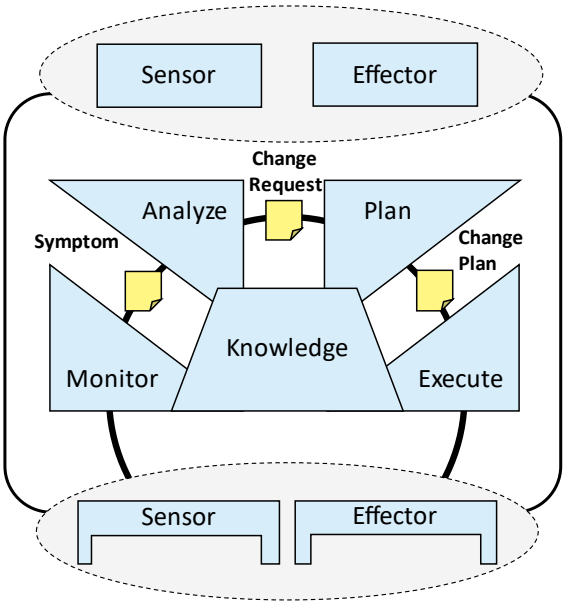


Figure 2.16: Reference Architecture of Autonomic Managers Based on MAPE-K [46][50]

The monitor function is responsible for the collection, aggregation, filtering, and reporting of the managed resource details, including information like topology, metrics, configuration property, state, and the provided capability of the managed resource. After processing the collected data (e.g., filtering), the processed data is used to aggregate and correlate the context of events to generate a symptom related to a particular combination of events. The generated symptom is then forwarded to the analyze function, as shown in Figure 2.16. It is emphasized that the monitor part may create knowledge based on current activities by logging the notifications from a managed resource [46].

The second part of the MAPE-K reference model is the analyze function, which aims to observe and analyze situations to determine whether a change must be performed. For this purpose, the analyze function includes mechanisms to correlate and model the situations. For example, once a particular policy is not being met, the requirement to enact a change is fulfilled [46]. Additionally, in some cases, the analyze function considers the context of the current situation and takes the future context into account during its processing. For this reason, the mechanisms of modeling the situations are also included, which allows the analyze function to complete tasks like time-series predictions or queuing of models.

With the help of these mechanisms, the analyze function can understand the current system state and allow the autonomic manager to learn about the IT environment and help predict future behaviors. Once a change is required, a so-called change of request as a standard description of essential modifications, which the analyze function considers, will be generated and forwarded to the next part: the plan function.

Once the plan function receives the change of request from the analyze function, it enacts a desired alteration in the managed resource by creating or selecting a procedure, which can be performed in diverse forms, ranging from a single command to a complicated workflow [46]. The output of the plan function is called a change plan, which includes a set of desired changes for the managed resource with certain orderings for the achievement of goals and objectives. The plan function uses policy information to guide its work [46].

The generated change plan still must be forwarded to the execute function, which as another part in the control loop of the MAPE-K reference model, is responsible for the execution control of the change plan. For this purpose, the execute function can schedule and perform the desired changes to the system. A single change plan may contain a series of actions that can be performed to modify the state of the managed resources. To perform the actions, the autonomic manager needs to use the effector interface of the touchpoint, which was presented in Section 2.2.7.3. Suppose the monitor part has created new knowledge. In that case, the execution of the generated change plan can also be completed by updating the created new knowledge into the final

knowledge source in the MAPE-K reference model, which was explicitly introduced in Section 2.2.7.4.

The knowledge included in the knowledge source is required during the processes of other system parts, like the monitoring, analyzing, planning, and execution functions. For example, the monitor function can create new solution topology knowledge based on the observed details of the managed resource (e.g., a new configuration or the construction of the system). The analyze function can use the policy knowledge to determine whether a change request is necessary. Following a standard means to define the policy with a uniform method, a single autonomic manager can use the defined policy for its decision-making, which can also be shared across multiple autonomic managers in the whole system. Problem determination knowledge includes monitored data, symptoms, and decision trees [30]. Thus, the monitor and the plan function use this knowledge to generate symptoms and determine the change plan.

2.2.8 DYNAMICO Reference Model

Most implemented approaches as contributions to self-adaptation assume non-mutable adaptation goals and monitoring infrastructures, which strongly limits the applicability of the approaches in the case of systems within highly changing environments. For this purpose, Villegas et al. [51] have proposed a so-called DYNAMICO reference model for engineering adaptive software systems. The general concept of the DYNAMICO reference model is inspired by the MAPE-K and the control theory with the classical feedback control loop. In this sense, the DYNAMICO can also be seen as an application and further development of MAPE-K with the integration of control theory approaches.

Integrating control theory approaches based on the feedback control loop and the self-adaptive software systems is not an entirely novel idea. Its benefits have been investigated by different researchers [25][52]. For example, Müller et al. [53] and Kramer and Magee [54] said that the feedback control loops had been considered a fundamental design element for designing systems with self-adaptation. However, visibility of the adaptation controller and the control loop is still missing, which leads to a lack of explicit methods for analysis, validation, and verification of the system and thus makes the measurement of the effectiveness of the adaptation mechanisms unrealizable [53].

DYNAMICO reference model aims to solve such issues by increasing the visibility of the feedback control loop components and making them analyzable, assessable, and comparable [55].

2.2.8.1 Fundamental Design of DYNAMICO

As discussed with the concept of MAPE-K, the autonomic manager is a component that implements an intelligent control loop [46], which is realized by a set of tight-coupled functions (cf. Section 2.2.7.5). In this case, the separation of concerns between the monitoring process, the adaptation controller, and the management of control objectives (adaptation goals) is still missing. The lack of the separation of concerns makes the consistency guarantee between the adaptation mechanisms and corresponding control objectives while preserving the relevance of context monitoring of the adaptation mechanism exceedingly tricky. Thus, loose coupling is selected in the concept of DYNAMICO, which makes the fundamental elements in the MAPE-K loop independent.

For example, a classical feedback control loop from the perspective of control theory, including corresponding components like controller and controlled system and their input and output, are schematically visualized in Figure 2.17. Since control theory, especially the classical control loop, was presented in Section 2.1.1, detailed discussion of the classical feedback control loop is excluded.

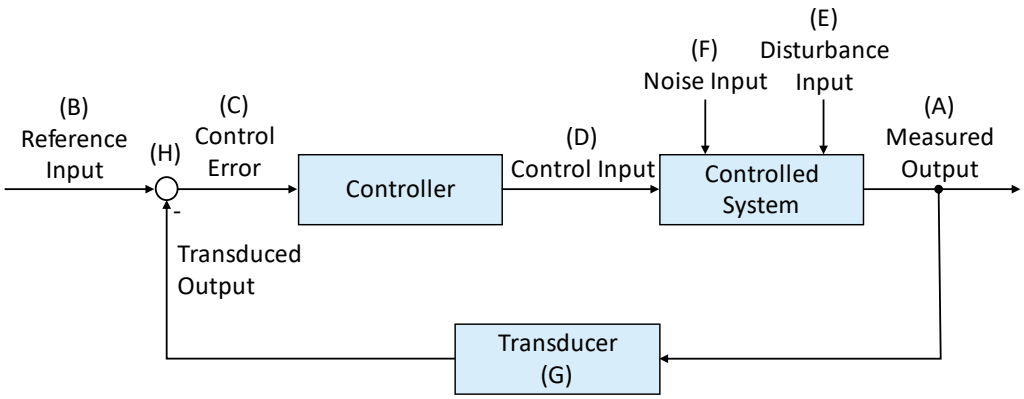


Figure 2.17: Classical Block Diagram of a Feedback Control System [51]

From another perspective, the control loop of the MAPE-K is realized by the monitor, analyze, plan, and execution functions, which were also discussed in previous sections. Based on both the classical feedback control loop and the control loop in the MAPE-K,

a fundamental structure with several general components in the DYNAMICO reference model with loose coupling is developed by merging both previous concepts shown in Figure 2.18.

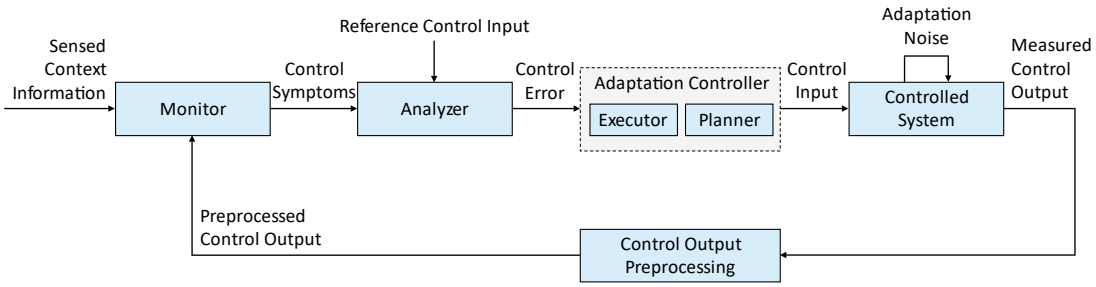


Figure 2.18: Fundamental Structure with General Components of DYNAMICO [51]

2.2.8.2 Hierarchical Architecture Based on Three Levels of Dynamics in DYNAMICO

The context-driven self-adaptation of the software system is also considered to increase the applicability of the DYNAMICO concept. For this purpose, three levels of dynamics in the use case of the self-adaptive system are identified: (1) the management of changing control objectives, (2) the dynamic behavior of the adaptation mechanism controlling the target system, and (3) the management of dynamic context information [51], which influence each other. This means that level (2) and level (3) must be adapted in the case of the change of control objectives at level (1). Considering this from the opposite direction, a change of context situation at level (3) may also require a review of the control objectives to modify the adaptation mechanism at level (2), even once the mechanism is working correctly [51].

Based on the identified three levels of dynamics, a hierarchical architecture is derived following the idea of separation of concerns at different levels, which thus consists of three subsystems implemented as three concurrent feedback loops (CO-FL, A-FL, and M-FL), as shown in Figure 2.19. CO-FL represents the control objectives control loop, which controls the change in adaptation goals and monitoring requirements to guarantee their fulfillment. In comparison, the A-FL as the adaptation feedback loop controls the adaptive behavior of the controlled system and the adaptation mechanism, considering the control objectives from the CO-FL and the monitored context events. Finally, M-FL

represents the dynamic monitoring feedback loop, which manages context information for preserving the context relevance of the adaptation mechanism [51].

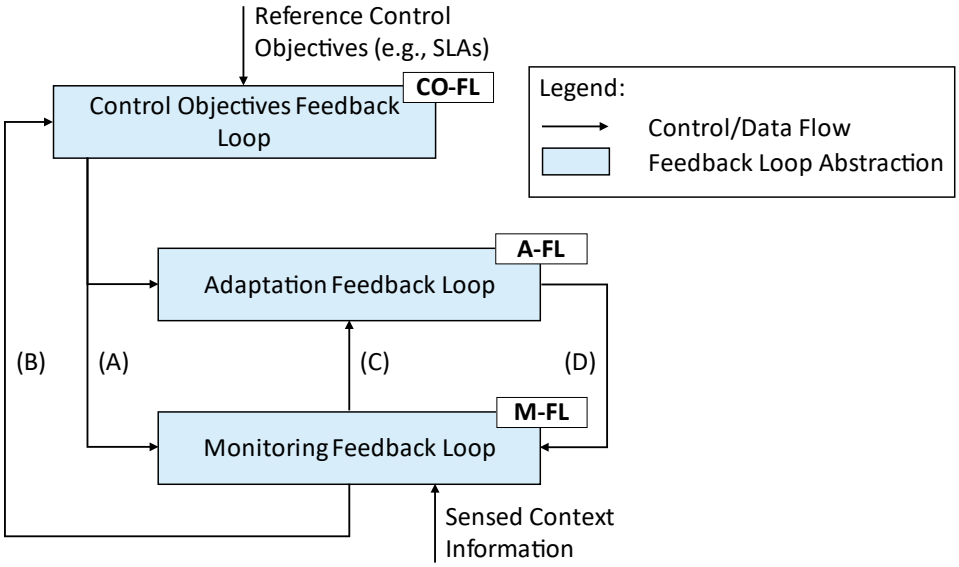


Figure 2.19: Three Levels of Dynamics in Context-Driven Self-Adaptive Software Systems [51]

In Figure 2.19, labels (A), (B), (B), (D) are highlighted interfaces between different control loops, which represent interactions specified depending on the requirements of concrete application. The interaction (A) provides requirements derived from the current control objectives to the M-FL. The interaction (B) supports the CO-FL in deciding on changes in the control objectives if the M-FL detects the requirement to change control objectives. The interaction (C) guarantees that the observed context by the M-FL can be considered during the processing of the A-FL. The interaction (D) represents the flow of sensed internal context of the controlled system included on the A-FL.

For each feedback loop, this fundamental structure with general components can be applied to derive concrete construction on a lower component level instead of on the level of the control loop. Thus, a detailed hierarchical architecture is constituted, including controllers on different levels of feedback loops, as shown in Figure 2.20. Each feedback control is comprised of a series of “MAPE” components. The A-FL and M-FL work together to take over the high-level objective-oriented control, which covers the regulation of requirements satisfaction and the preservation of adaptation properties and is defined as system variables: *control objectives* or *adaptation goals* [51].

To specify the control objectives, the requirements can be either functional or non-functional, which the target system should satisfy. The adaptation properties refer to the inherent properties of the self-adaptive software, which are quantitatively represented by quality attributes and thus can be exposed by adaptation mechanisms. As illustrated in Figure 2.20, the control objectives can also be modified by user-level negotiations at runtime, which must be addressed consistently and synchronized at the level of A-FL and M-FL [51]. In the case of a change of control objectives, the reference control input at the A-FL level and the reference context input at the level of M-FL also must be derived and adapted automatically and fed into corresponding feedback loops.

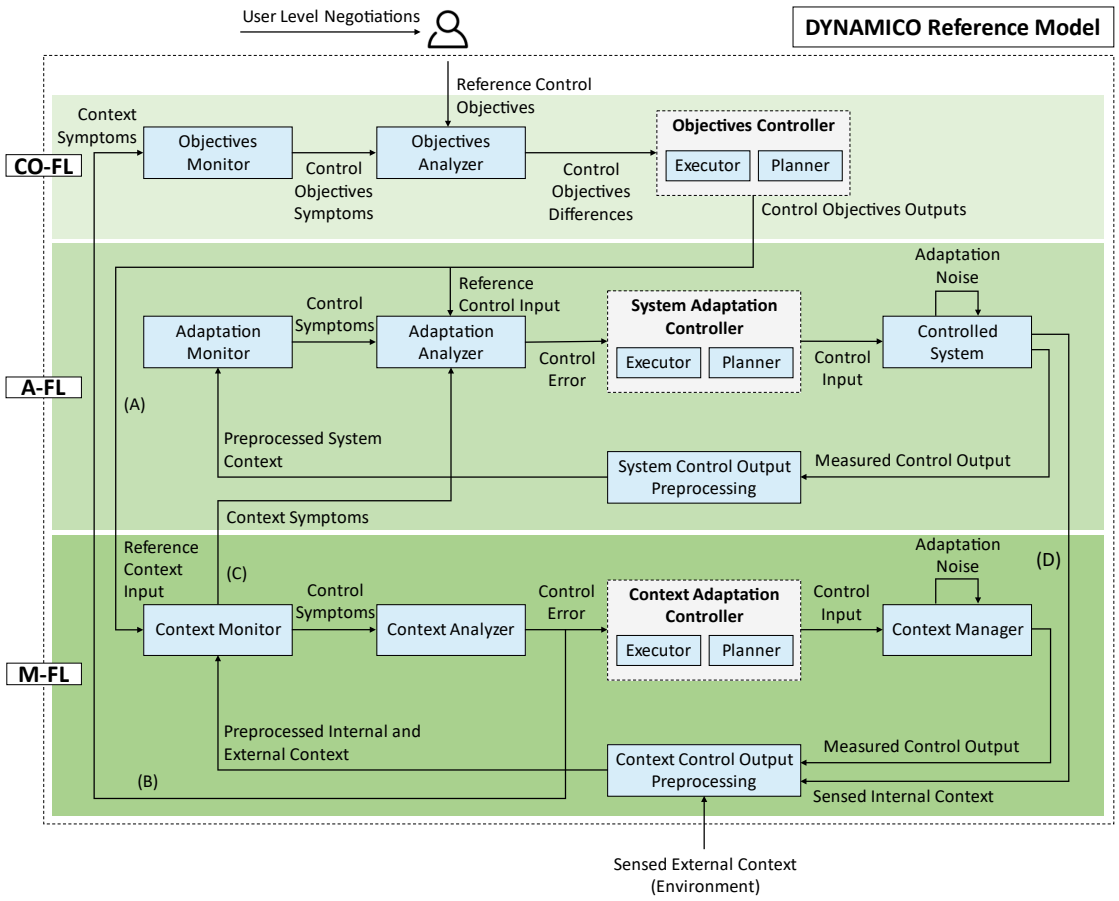


Figure 2.20: DYNAMICO Reference Model with Controllers for the Three Levels of Dynamics [51]

The second feedback loop A-FL follows the mechanism of control theory with quantitative expressions to measure the control error between the current value of the controlled system variables and the set value of the reference control inputs for these

variables. Relying on the context monitor, the A-FL can continually monitor the state of the controlled system, which is analyzed by the component *Adaptation Analyzer*, aiming to determine whether an adaptation is necessary. As in the case of the MAPE-K reference model, the planner and executer in the component *System Adaptation Controller* take over the responsibilities of determining adaptation strategy to fulfill the control objective by eliminating violations and performing concrete adaptation actions included in the strategy.

The M-FL works as an independent feedback control loop focusing on the dynamic nature of the controlled system's context information. The reference context inputs correspond to the reference context management objectives derived from the reference control objectives at the CO-FL level. The *Context Monitor* gathers context information from the internal and external environment, which is preprocessed by the component *Context Control Output Preprocessing* and used to generate symptoms, which can also influence the adaptation of the target system (cf. label (C)). The *Context Analyzer* analyzes the symptoms, considering the system states' past, current, and future context. Thus, it supports the *Context Adaptation Controller* to adapt the monitoring strategy (e.g., in the case of a change of control objectives via the user-level negotiation or evolvement of the adaptive system). A typical example of the adaptation monitoring strategy for the M-FL could be the deployment of new context management instrumentation. For the CO-FL, the output of the context analyzer is also derived as context symptoms to support the decision-making about changing the system objectives at the CO-FL level (cf. label (B)) [51]. The context adaptation controller defines and executes the adaptation plan for the *Context Manager*; these goals are derived from the measured control output and the sensed internal context of the controlled system.

2.3 Generic Communication Architecture Patterns

In addition to architecture design from the static view of system construction, another perspective on designing self-intensive systems is the dynamic view, which focuses on component interactions within the system. For this reason, this dissertation also focuses on generic communication architecture patterns, which can be applied to specify the

concepts of the communication between the components, primarily focusing on the use case of distributed systems.

This dissertation considers the results of previous work as a reference, including a taxonomy of generic communication paradigms for distributed systems [48]. This taxonomy includes remote procedure calls (RPC), message-oriented communication, stream-oriented communication, and data-based communication, as shown in Figure 2.21. Additionally, several generic communication architecture patterns are summarized as examples, relying on their communication paradigms as underlying technologies [56][57]. These communication architecture patterns will be presented in detail in the following sections.

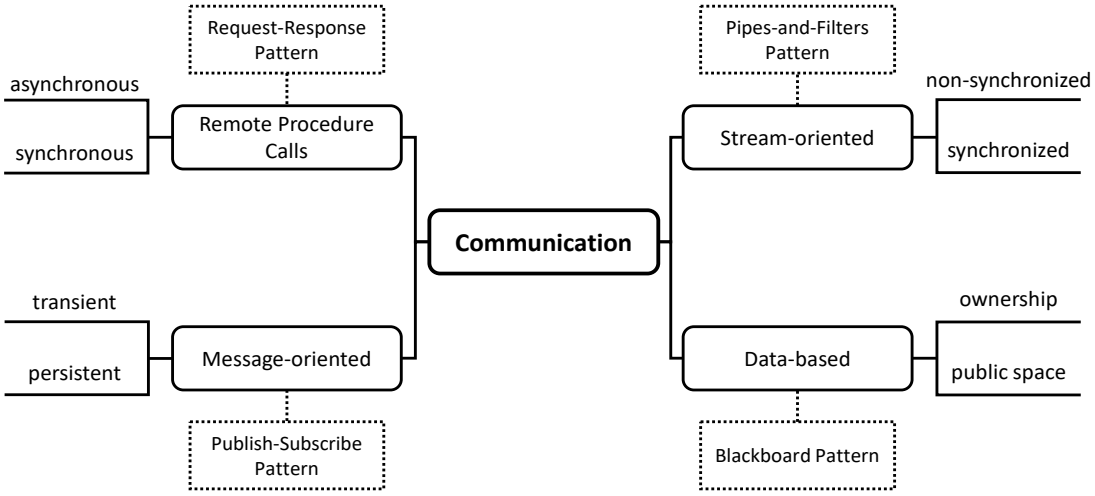


Figure 2.21: Taxonomy of Generic Communication Paradigms in Architecture Pattern [48]

2.3.1 Request-Response Pattern

The first communication architecture pattern is called the request-response pattern, also named the client-server pattern. It is developed based on the communication paradigm of remote procedure calls (RPC). RPCs refer to the process that a program on the local machine (client) calls a procedure located in a different address space, which means that the procedure is located on another machine (server) [57]. Since the client and server machines do not share the same address space, parameter passing by using the stack is unreliable for RPCs. Thus, RPCs are coded like local ones as much as possible so that the developer is not required to handle underlying code for remote interaction.

As the initialized process in an RPC, the client procedure calls a client stub, a piece of code for parameter conversion aiming to build a message, as illustrated in Figure 2.22. The message built by the client stub will be forwarded to the operating system (OS) deployed on the local machine, which communicates with the remote OS deployed on the server machine via the network. Thus, the message can be transmitted from the local machine to the remote machine. Another server stub on the remote machine unpacks the parameters from the message and calls the server procedure [57].

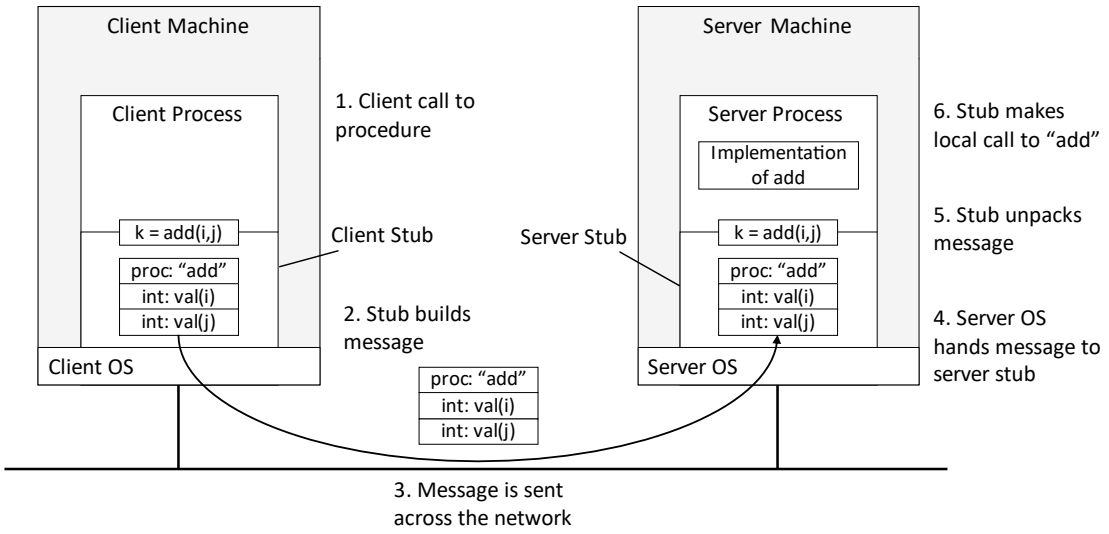


Figure 2.22: Process Flow of a Remote Computation through RPC [56]

After the processing by the server procedure, the result is returned to the client procedure via the process flow in reverse order. It is emphasized that a RPC only supports the mechanism of call-by-value, which means that the client procedure copies the parameter values and sends this copy to the server procedure instead of directly sending the values. In this case, the client procedure aligns the parameter values, which are returned from the side of the server procedure. If necessary, the client procedure should also check consistency between the initially saved parameter values on the local machine and the returned parameter values from the remote machine.

The RPC can still be divided into synchronized and non-synchronized RPC, as illustrated in Figure 2.23. A synchronized RPC means that the client stub blocks itself after sending the message until a reply comes back from the server procedure on the remote machine, and a non-synchronized RPC means that the client stub will not block

itself. In the non-synchronized RPC, the server procedure immediately sends a reply as an acknowledgment back to the client-side to confirm that the request is received. Thus, the client stub can immediately continue working without blocking [58].

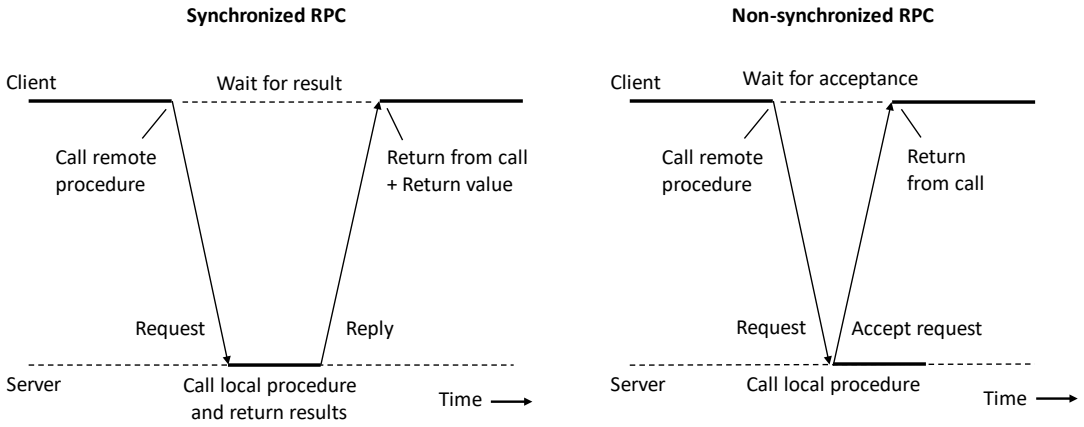


Figure 2.23: Process Flow of Synchronized and Non-Synchronized RPC [56]

In comparison to the synchronized RPC, the non-synchronized RPC increases overall system performance due to the independent parallel computation of the client and server procedures. However, from another perspective, in this case, there is no guarantee of reliability since the client never knows whether the server will process the request.

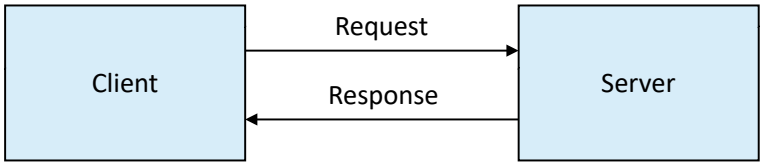


Figure 2.24: Architecture of Request-Response Pattern [59]

Based on RPC as the underlying technology, the request-response pattern is developed, as shown in Figure 2.24. In the request-response pattern, the client initializes an interaction with the server by invoking provided services of the server and subsequently waiting for the requested results [60]. In this case, the client and the server have ports to describe the services they require and provide. One server can be connected to multiple clients through request/reply connectors, defined as a data connector employing a request/reply protocol.

Since the server can be a service provider for several clients, there are significant disadvantages in the request-response pattern. The server can be a performance

bottleneck or a single point with a high risk of failure. Additionally, system design based on the request-response pattern is often complicated since the component interactions must be individually defined. Such an approach makes the component interactions later cost-intensive to change once the system has been built [52] and simultaneously limits the number of active clients. Generally, it can be said that the request-response pattern is not appropriate for applications requiring transferal of data in significant volumes.

2.3.2 Publish-Subscribe Pattern

The second communication architecture pattern is the publish-subscribe pattern, which is based on the underlying technology of message-oriented communication. Message-oriented communication refers to communication between the participating components relying on specified messages via an intermediate-term storage capacity for transmitting messages from sender components to corresponding receiver components.

In message-oriented communication, there is a loose coupling between the sender and receiver, which means that it is not required that the senders and receivers are active during the message transmission. This loose coupling is realized by a change propagation infrastructure that plays the role of the intermediary, as shown in Figure 2.25.

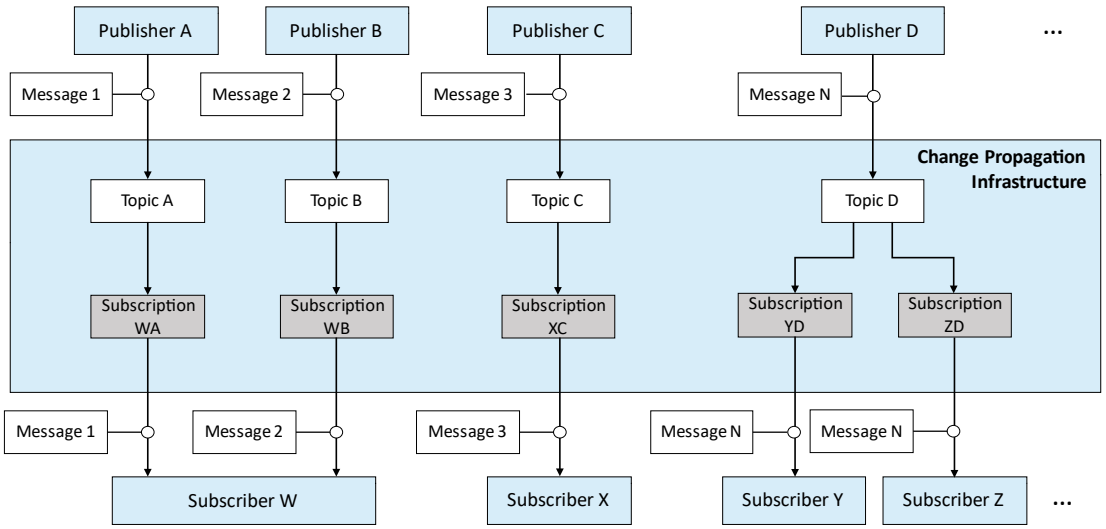


Figure 2.25: Architecture of Publish-Subscribe Pattern [58]

With the help of the change propagation infrastructure, the sender only guarantees that message will be inserted into a logical message channel, which is also called topic (cf. Figure 2.25), without knowing when and whether the message will be read. Instead, the receiver makes decisions about reading the message. In some cases, different applications may have diverse message formats. Thus, the broker [61] can also be integrated into the change propagation infrastructure to handle the conversion between different message formats and temporarily store the messages, making advanced enterprise application integration much more effortless. Indeed, the broker is not always required in the communication once the message formats of the applications are predefined previously and thus are consistent. Generally, it is not worth agreeing with a standard message format in the case of communication between heterogeneous systems [57].

The information exchanged between the sender and receiver components is encapsulated inside events and realized as messages routed and forwarded by the change propagation infrastructure. In this case, sender components (called *publishers*) and receiver components (called *subscribers*) work independently and are not aware of each other's locations and identities. The publishers disseminate events that convey information without concern for which subscribers are interested. The subscribers also do not care about the events coming from which publisher and are only interested in the published events, including the expected information. The communication in the publish-subscribe-middleware is asynchronous, which means that the publishers continue their processing without blocking themselves after event dissemination.

For this purpose, publishers and subscribers must register with the change propagation infrastructure to realize such a communication mechanism. Thus, the infrastructure can acquire the information about event types to be published and received and thus can route events from publishers via network to interested subscribers using registration information.

In such publish-subscribe-middleware, concrete message formats are hidden by the events from the publishers, subscribers, and the change propagation infrastructure. Such an approach makes the modification of the message format very transparent. Nevertheless, from another perspective, the publish-subscribe-middleware also has

disadvantages. For example, the concept of anonymous communication possibly causes unfavorable overhead once the subscribers have limited specifications (e.g., in cases with very few types of events or strictly defined reaction criteria) [61].

Due to its loose coupling between publishers and subscribers, message-oriented communication has provided tremendous advantages for distributed system communication. On top of the message-oriented communication, the publish-subscribe pattern is developed and implemented as so-called publish-subscribe-based middleware (e.g., typical middlewares with a centralized broker like MQTT, CORBA [62], or ROS1[4]). Unlike the middleware with a centralized broker, some other middleware relies on concurrency and networking programming patterns like DDS (data distribution service) products like ROS2[5] instead of a broker.

2.3.3 Pipes-and-Filters Pattern

The third communication architecture pattern focused on in this dissertation is the pipes-and-filters pattern, which relies on the underlying technologies of stream-oriented communication, primarily focusing on the exchange of time-dependent information. Considering the previously introduced two communication paradigms, neither request-response-pattern nor publish-subscribe-pattern focus on the timing of the communication, which, in cases of multimedia, plays a crucial role [56]. Different concepts of stream-oriented communication have been developed relying on the data stream, including a sequence of data units or items, to deal with the timing requirements. The transmission of the data stream can be categorized into three modes: asynchronous transmission, synchronous transmission, and isochronous transmission.

In the asynchronous transmission mode, the data items in a stream are transmitted one after the other without any time constraint. For example, in the case of a file transfer, it is irrelevant when the transfer of each item is completed. Instead of the asynchronous mode, the synchronous transmission mode requires a maximal end-to-end delay for each item in the data stream. As a more strict concept, maximal and minimal end-to-end

[4] ROS1: Robot Operating System, Version 1: http://wiki.ros.org/Documentation (accessed: 4th April 2022).

[5] ROS2: Robot Operating System, Version 2: http://docs.ros.org/en/rolling/ (accessed: 4th April 2022).

delay are required in the isochronous transmission mode, which is particularly meaningful for distributed multimedia systems and thus is included in the focus of this dissertation [56].

In isochronous transmission, the timing requirements are expressed as quality of service (QoS), which focuses on timeliness, volume, and reliability in the continuous data stream (e.g., the maximum delay variance and jitter [63]). The most effective approach in the isochronous transmission is the buffering for reducing jitter to enforce the QoS. Thus, the receiver stores packets in a buffer for a certain maximal period, aiming to keep regularly passing packets (to the application). In addition to buffering, other approaches like error compensation have been developed to deal with the issue of losing packets [64].

In some use cases of stream-oriented communication with multiple data streams (e.g., from different sensors) there is no need to synchronize the data streams. But in other use cases, the synchronization of different streams is required, particularly in the case of the multimedia stream like internet telephone or video conference. The synchronization mechanisms for stream-oriented communication can be understood as approaches on different levels of abstraction. For example, one of the possible approaches is to deploy a procedure in the application to execute read and write operations with consideration for timing and synchronization constraints [56]. Alternatively, a middleware layer with multimedia control offers a group of interfaces to control streams such as video and audio, and devices such as cameras and microphones can be deployed in the application. In this case, each device and stream relies on its high-level interfaces to notify the application about the event of an incoming stream, which is subsequently used to synchronize streams by writing handlers [65]. In addition to middleware, deployment of a synchronization specification containing required information on the receiver side has also been applied [56].

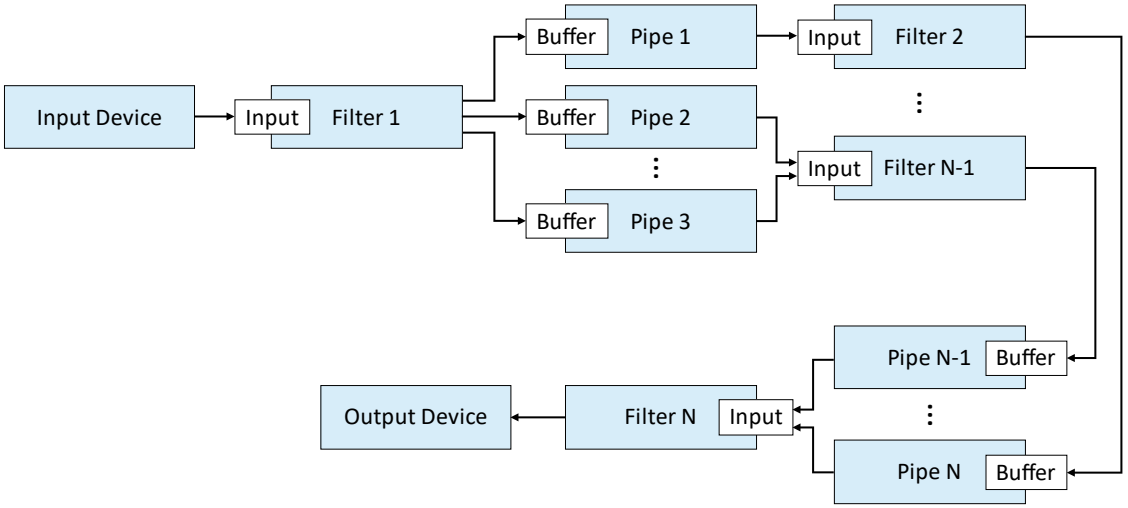


Figure 2.26: Architecture of Pipes-and-Filters Pattern [61]

Based on the underlying technologies of stream-oriented communication, the pipes-and-filters pattern is developed; this architecture is shown in Figure 2.26. As shown in Figure 2.26, the input device provides the input data streams, which are processed stepwise and forwarded as output data streams into the output device. In this case, each processing step is completed by independent filter components with loose coupling. The filter components consume and then deliver the data after corresponding individual transformation processes. Once the filter component includes activity with a long execution duration, it can also integrate multiple instances for parallel computation. Some instances can already initialize the processing of a new data stream, even when the previous data stream is not yet completed.

The pipes are deployed between the filter components, which work as connectors for the stream by consuming the data at the input port and subsequently forwarding the data to one or multiple output ports without any transformation [61]. As the intermediary for exchange and coordination of the data, the pipes include policies for buffering to guarantee a regular data rate, as discussed in the introduction to stream-based communication.

Since there is no cycle in the pipes-and-filters pattern, it is not appropriate for interactive systems, in which user feedback is essential [60]. Due to loose coupling, each filter works as an individual thread or process, which possibly causes overhead in the case of a system with a significant number of filters. The pipes-and-filters pattern does not

have high system reliability in the case of a long-term computation since there is no approach for the realization of checkpoints or restoring functionality in the pattern. Thus, the whole system will fail if any filter has malfunctioned [61].

2.3.4 Shared-Repository Pattern

The fourth communication pattern refers to the shared-repository pattern, relying on the underlying paradigm of data-based communication. The communication between components is based on the exchange of structured data [48][57]. There are multiple application components in the shared-repository pattern, all of which have access to a shared data repository, as shown in Figure 2.27. In this case, the application components do not know each other, and a shared data repository communicates with them. The communication's control flow is triggered and coordinated by the availability, quality, and state of the saved data in the repository [61][66].

The application components can work directly on the data saved in the shared repository. In the case of a data change in the repository, the shared repository notifies application components about the data change to react to it immediately. The notification mechanism is realized by observer arrangement. In this case, the shared repository plays the role of the subject, and the application components are the observers. For example, once a component has created new data and inserted the created data into the shared repository, the other components will receive a notification from the repository and thus can also access the newly inserted data [61].

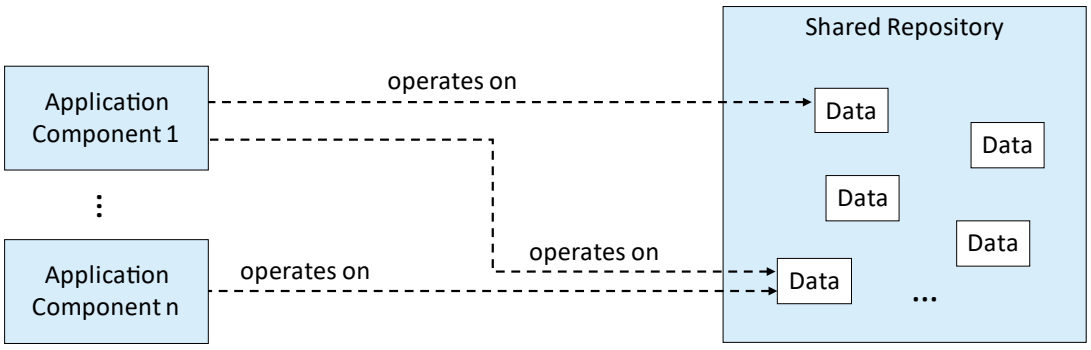


Figure 2.27: Architecture of Shared-Repository Pattern [61][66]

Based on the above, it can be understood that the control flow in the shared-repository pattern is data-driven without participating in the ordinary business process, which

makes the shared repository a performance and scalability bottleneck and a single point with a high risk of failure [60]. The saved data can be encapsulated inside managed objects in the shared repository. The managed objects can be implemented as domain objects that hide the details of concrete data structures and offer operations for their access and modification [61]. Nevertheless, the application components as data producers and consumers may still be tightly coupled through their knowledge of the data structure [60].

Generally, the notification of the shared repository in the case of data change can be designed to perform on different levels relying on the utilization of managed objects. The notification on the repository level makes the implementation easy, which could cause an overhead of notification and data transfer once most application components are not interested in the data change. From another perspective, once the notification is performed on the managed object level, unnecessary notifications and data transfer can be avoided, which leads to higher system complexity.

2.3.5 Blackboard Pattern

The final communication architecture pattern is the blackboard pattern, which includes a blackboard as a shared data repository for structured data exchange [61]. Due to this feature, some researchers categorize the blackboard pattern as a variant of the shared-repository pattern [60]. The only difference is that the control flow in the shared-repository pattern is driven by the data change in the shared repository. Instead, the control flow in the blackboard pattern is driven by a control component independent of the blackboard, which works as a shared repository. Other researchers have also summarized this idea in their work [61].

Generally, the blackboard pattern is designed to construct systems dealing with tasks based on uncertain, hypothetical, or incomplete knowledge and data. It is emphasized that there is no guarantee of a valid result, which means that the pattern is not appropriate for the systems with expected predictability of result quality or time constraints like the worst-case execution time. The core idea behind the pattern is to decompose the overall task into smaller, self-contained subtasks for which deterministic solution algorithms are known [61]. In this case, the subtasks are assigned to

independent knowledge sources, which can be coordinated and activated by the control component based on heuristic computation with an arbitrary order to gradually improve an intermediate solution hypothesis on the blackboard, as shown in Figure 2.28.

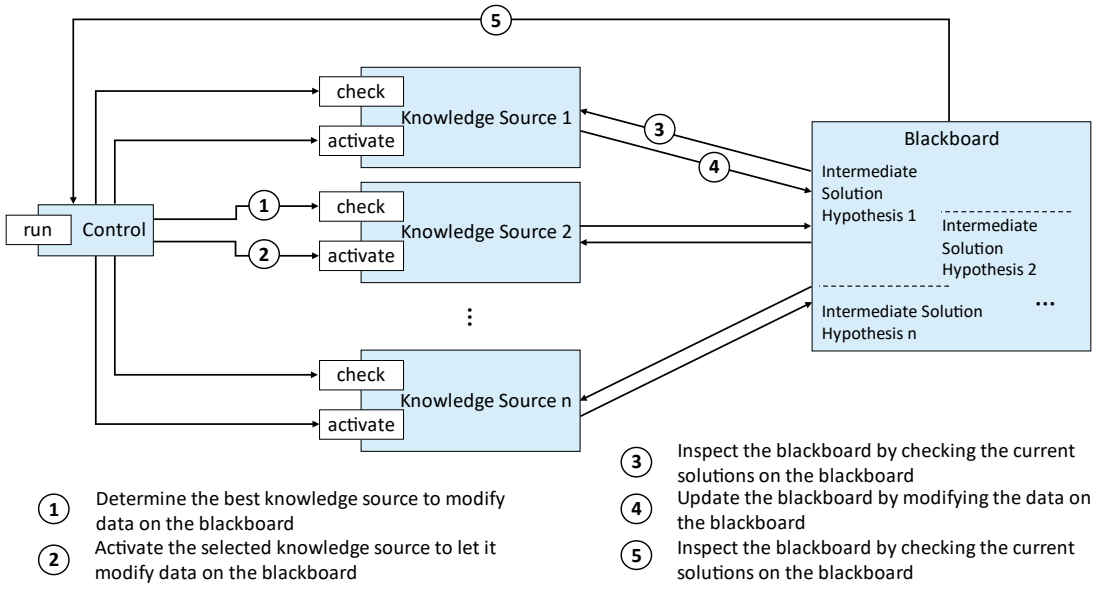


Figure 2.28: Architecture of Blackboard Pattern [61]

The blackboard pattern is appropriate when systems have boundary conditions like diverse input data sources physically distributed in environments. However, from another perspective, this pattern also has disadvantages (e.g., expensive implementation, few supports of parallelism, and testing difficulty due to the application of the heuristic approach).

2.4 Applied AI-Based Technologies in This Dissertation

In this dissertation, the main contribution is a generic architecture style, which can be used as a template to design automatic control systems in the future. An example architecture of a so-called artificial cognitive cruise control (ACCC) system is derived by following the architecture style. In addition, the ACCC is implemented as a prototype of future ACC variants to evaluate the plausibility of the proposed architecture style. For this implementation, different technical approaches, particularly applied artificial intelligence algorithms, will be described in detail in the following sections.

2.4.1 Q-Learning

Artificial intelligence and its subfield of machine learning are becoming increasingly attractive and have been applied to diverse applications. Generally, machine learning approaches can be classified into different broad categories (e.g., supervised learning and unsupervised learning), which are differentiated by whether training data as labels are available while training a machine learning model such as a neural network. Unlike supervised and unsupervised learning, another learning approach named reinforcement learning, which learns the best policy based on the environmental reward by taking actions with a set of trial-and-error runs [67], is applied within the application example of this dissertation. Figure 2.29 illustrates a general process flow of reinforcement learning.

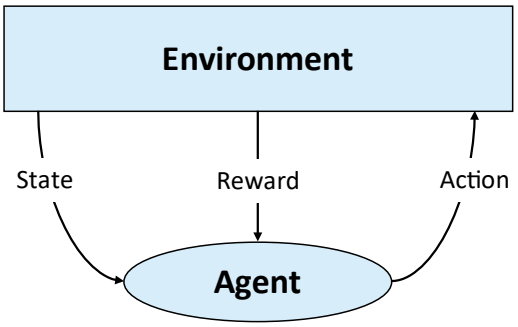


Figure 2.29: General Process Flow of Reinforcement Learning [67]

Based on model availability, reinforcement learning can still be categorized into model-based and model-free approaches. One of the most important breakthroughs in model-free reinforcement learning was Q-learning [68], which is a kind of off-policy TD (temporal difference) control algorithm [69]. In its simplest form, one-step Q-learning is defined in the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.17}$$

In the equation, *Q* represents the learned action-value function. The action-value function is dependent on the variables *S* and *A*, respectively representing the state and action of the agent at each time point (subscript *t*). The variable *γ* represents a learning rate during each step in the learning process by adapting the Q-value. Based on the equation, it can be seen that the learned action-value function can directly approximate the optimal one, which is independent of the policy being followed [68].

$Intialize\ Q(s,a), \forall s \in S, a \in A(s), arbitrarily, and\ Q(terminal-state,\cdot) = 0$
$Repeat\ (for\ each\ episode)$:
    $Initialize\ S$
    $Repeat\ (for\ each\ step\ of\ episode)$:
        $Choose\ A\ from\ S\ using\ policy\ derived\ from\ Q\ (e.g., \epsilon-greedy)$
        $Take\ action\ A, observe\ R, S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
        $S \leftarrow S'$
    $until\ S\ is\ terminal$

Figure 2.30: Pseudo Code of Q-Learning — An Off-policy TD Control Algorithm [69]

Such an approach enormously simplifies the analysis of the algorithm and enables an early convergence, in which the mentioned parameter of learning rate also plays an important role. The detailed process flow of the Q-learning algorithm is illustrated in Figure 2.30.

2.4.2 Kernel Density Estimator

In machine learning, the inferential statistical method is a significant important subfield. It can be divided into parametric, semiparametric, and nonparametric methods. The parametric and semiparametric methods work based on the assumption that the data is drawn from one or a mixture of probability distributions of known form. The nonparametric methods are applied if there is no such assumption about the input density, and the data speaks for itself [67].

In this dissertation, the kernel density estimator (KDE), one of the most well-known approaches in nonparametric methods, is used within the application example. Depending on variable numbers, KDE can still be categorized as monistic and multivariate approaches. The multivariate approach is out of the focus of this dissertation. The most fundamental element of KDE is a so-called Gaussian kernel, which is a probability density function based on Gaussian distribution, as illustrated in the first Equation (2.17). *K(u)* represents the probability density, in which *u* represents the deviation between each sample (*x'*) and the central data point (*x*) within the Gaussian distribution. The central data point here means the data point with the highest density.

$$Gaussian\ Kernel: K(u) = \frac{1}{\sqrt{2\pi}} exp\left[-\frac{u^2}{2}\right] \tag{2.18}$$

The Gaussian kernel works based on the assumption that the world is smooth and all functions inside change slowly [67]. Thus, in the case of a parameter has acquired a historical value of x_i ($i \in [1\ n], n \in \mathbb{R}$), during the value prediction, all neighbor values will receive corresponding probability densities, in which the Gaussian kernel is used as a smooth weight function, as shown in Figure 2.31 (cf. individual green curve). In cases where the parameter has acquired several historical values ($x_1, x_2, \ldots, x_m, \ldots, x_n$), as shown in Figure 2.31, multiple green curves with the historical values as their central data points can be found to visualize the probability density distributions.

$$Kernal\ Estimator\ (Parzen\ Windows): \hat{p}(x) = \frac{1}{Nh}\sum_{t=1}^{N} K(\frac{x - x'}{h}) \tag{2.19}$$

All density distributions can be merged using the kernel estimator, as illustrated in Equation (2.19). In this case, the kernel with subscript *h* is called the scaled kernel ($K_h(u) = \frac{1}{h}K\left(\frac{u}{h}\right), u = x - x'$), in which *h* is an application parameter of bandwidth. Selection of an optimized bandwidth requires consideration of a tradeoff between the variance and bias of the overall estimator [67].

An overall density distribution (purple curve in Figure 2.31) can be found with an appropriate bandwidth. With the help of the purple curve, the next value prediction of the parameter *x* will become more precise since all historical values will be considered. However, from another perspective, it is emphasized here that temporal dependencies between the historical values are not specially considered in the approach of KDE, which means that the appearance sequence of the historical value does not influence the final prediction result.

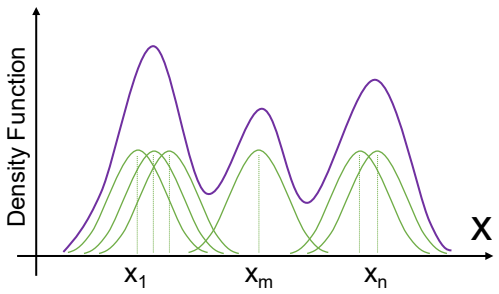


Figure 2.31: Sample Visualization of Probability Density in Kernel Density Estimator [70]

2.5 Summary

Chapter 2 has introduced the theoretical basics of control theory and the architecture design of self-intensive systems. For the control theory, the research work of Trächtler and Gausemeier [11] is taken as the reference taxonomy of current control systems in this dissertation, which was presented in this chapter. In addition, this chapter provided a short overview, including different concepts of software architecture design. An introduction to generic communication architecture patterns and applied AI-based technologies in this dissertation are also included in this chapter.

Since the main contribution of this dissertation deals with a generic architecture style for designing automatic control systems, a case study focusing on the architectural evolution of different automatic control systems is included in this dissertation, which will be covered in the following chapter. Additionally, this dissertation uses a vehicle's ACC as an example in the case study to better illustrate the architectural evolution. Finally, an artificial cognitive cruise control (ACCC) to improve the current ACC variants on the market is implemented based on the instantiated architecture by following the generic architecture style.

3 Case Study: Architecture Evolution of Automatic Control within the Example of Adaptive Cruise Control

This chapter introduces a case study focusing on the architectural evolution of automatic control systems based on application examples of a vehicle's ACC. Different automatic control concepts applied in current ACCs are analyzed and categorized into four levels: basic control, naive adaptive control, controlled-plant-dependent adaptive control, and physical-system-dependent adaptive control, which are initially inspired by Trächtler and Gausemeier's taxonomy and furtherly developed considering the perspective of increasing system adaptability and autonomy (cf. Chapter 2) [11][71][72].

In this chapter, the well-known MAPE-K model described in Section 2.2.7 is taken as a reference to be roughly compared with the previously mentioned four levels of control concepts. Technical limitations of ACCs applying these control concepts are also discussed in this chapter to determine existing issues with current control concepts. To address these issues, challenges for the architecture design of future automatic control systems are discussed. Finally, a future control concept named artificial cognitive control is defined in the summary of the illustrated architectural evolution at the end of this chapter.

3.1 Basic Control in ACC

While designing a control system, it is always challenging to take all possible operating situations and corresponding appropriate system behaviors at run time into account, particularly for systems working in an uncertain surrounding environment with high nondeterminism. Faced with this challenge, different automatic control concepts with diverse technologies have been developed. The most fundamental concept of automatic control is named basic control in this dissertation, referring to control systems based on a controller with a static parametrization at run time, as shown in Figure 3.1.

Basic control consists of a technical system and a physical system. A controller (e.g., a PID controller) is implemented in the *Controller* component of the technical system,

relying on the static parametrization saved in a so-called *Parametrization Memory* component (cf. Figure 3.1). The controller takes a control error represented by the deviation between the set value of the reference variable and the controlled variable's current value, which as an input is provided by another component *Measurement Unit,* via a feedback loop. Another input to the controller comes from an optional *Analyzer* component, which delivers the value of the predicted variable by analyzing future feedback from the physical system within the following time cycle. Based on the inputs of control error and predicted variable, the controller makes a decision and sends a control command to the component *Final Control Unit*. Thus, the final control unit can manipulate the hardware actuators (*A*) to guarantee the user's preferred set value.

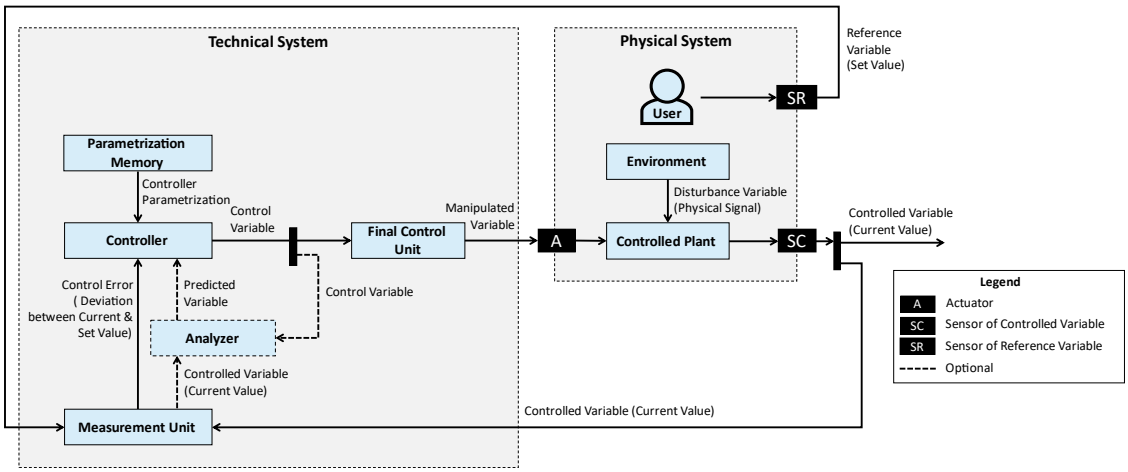


Figure 3.1: Basic Control Applied in ACC

In addition to the actuators in Figure 3.1, different hardware sensors are deployed as interfaces between the technical and physical systems. For example, the current value of the controlled variable is collected by the component *Sensor of Controlled Variable* (*SC*), and the user-preferred set value of the reference variable is collected by the component *Sensor of Reference Variable* (*SR*). In this case, the technical system's observation scope for the physical system is determined based on the data access of the sensor components.

The controller parametrization in basic control is defined as a vital system configuration, considering system operation and corresponding environmental disturbances within a limited workspace. The limited workspace is determined by manual assumptions based

on the human engineers' domain knowledge, usually described in the form of differential equations. The defined parametrization is saved in the parametrization memory component at design time. Such a control concept has been widely applied to ACC systems for a long time.

The first generation of ACC was only utilized for luxury vehicles by automobile manufacturers and their suppliers to enhance driving comfort and convenience in the 1990s. It has been over twenty years since the first ACC-equipped vehicles were available and over ten years since the ISO standard for vehicle systems was produced [73][74]. The ACC relieves the driver from routine physical tasks in driving by maintaining a steady headway from the last preceding vehicle or a constant cruise velocity in the case of no vehicle in front [74]. As the system user, the driver manually sets the headway and constant cruise velocity in advance in the ACC. The driver-preferred headway and cruise velocity are then taken as the set values (cf. Figure 3.1) of the reference variable in corresponding use cases of the control system.

The controller in ACC relies on static parametrization to decide the vehicle's acceleration or deceleration to fulfill the driver's preferences. The operation of the entire control system relies on a feedback control loop, as shown in Figure 3.1. In the control loop, the physical system refers to a summary of physical processes within a limited observation scope (e.g., in the case of ACC with basic control, referring to the directly dependent processes influencing the controlled variable within the vehicle's physical components). The measurement unit is used to forward the current and set value of the cruise velocity or headway to the controller. To distinguish from later ACCs with advanced features, an ACC solely based on basic control with static parametrized controller and feedback control loop at run time is referred to as classical ACC in this dissertation.

As discussed earlier, the controller component in basic control is parametrized at design time, considering a certain workspace limited by manual assumption and aiming to guarantee system performance at run time within the workspace. From the view of the ACC application, this feature ensures that classical ACC based on basic control has strong robustness against disturbances coming from the driving environment, like the wind influences on the vehicle's driving dynamics. From another point of view, this feature unfortunately also leads to disadvantages for classical ACC, especially in the

cases of its physical system (cf. Figure 3.1) with highly dynamic factors or the driver as a user with dynamically changeable preferences during the ACC's operation.

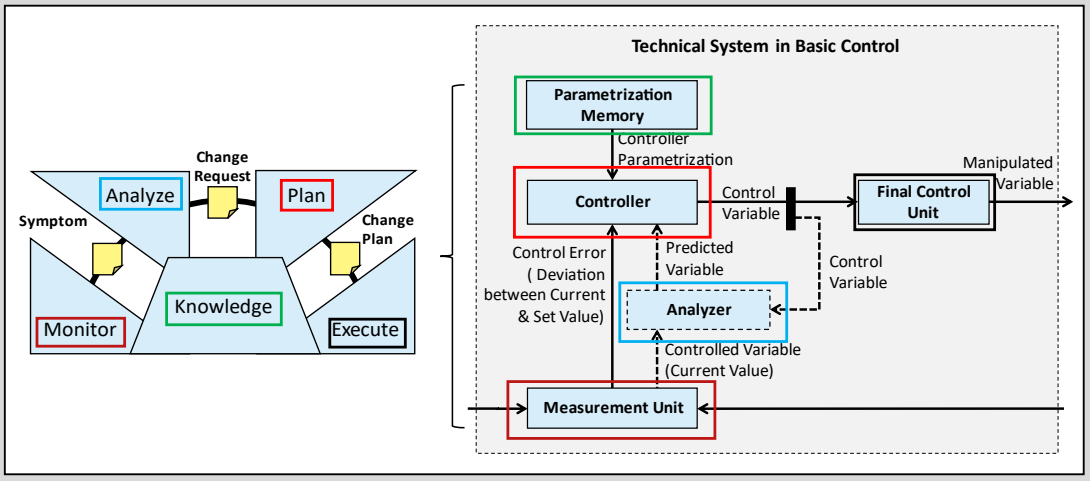


Figure 3.2: Architectural Comparison of Basic Control and MAPE-K

Figure 3.2 illustrates component dependency between the architecture of basic control and MAPE-K (cf. Section 2.2.7). Relying on the technology of variable monitoring, the measurement unit in the basic control is responsible for the sensory data collection and preprocessing, which corresponds to the monitor function of the MAPE-K model. As an optional component in basic control, the analyzer component takes the current value of the controlled variable as input to decide whether it is necessary to provide a time-series prediction for the future context of the technical system. Thus, the functionality of the analyzer roughly corresponds to the analyze function in the MAPE-K, with the task of symptom evaluation and decision-making about change request, possibly considering the future context of the controlled system. Moreover, the controller here takes the similar responsibility of plan function in the autonomic manager, which is very simplified, only considering the current time point without any predictive time horizon. Finally, the final control unit takes a similar responsibility as the execute function in the MAPE-K, which performs the determined strategy by the plan function.

For example, a vehicle controlled by an activated classical ACC moves on a mountain road with many curves. In this case, the ACC may determine critical control strategies due to the controller's static parametrization (e.g., aggressive acceleration or deceleration of the vehicle). Such critical strategies may make the driver uncomfortable and thus violate the requirement of driving comfort. Furthermore, if the road surface is uneven or slippery, aggressive driving activity may even possibly lead to an accident risk (due to unsafe slip) in the worst case. Facing this issue, classical ACC needs to adjust its configuration, mainly referring to the controller parametrization. Thus, the

vehicle's acceleration and deceleration strategy could be adjusted during system operation if necessary. Unfortunately, as discussed earlier, a change of controller parametrization at run time is beyond the ability of a classical ACC based on basic control.

3.2 Naive Adaptive Control in ACC

As presented in the critical scenario in the previous section, classical ACC has limitations due to the missing adaptation capability of the controller parametrization. Thus, classical ACC has been developed by integrating naive adaptive control to overcome this limitation. Naive adaptive control in this dissertation refers to the control systems with a run-time adaptation ability of controller parametrization once the user selects a preferred operating mode by manual intervention. Figure 3.3 shows the architecture of naive adaptive control.

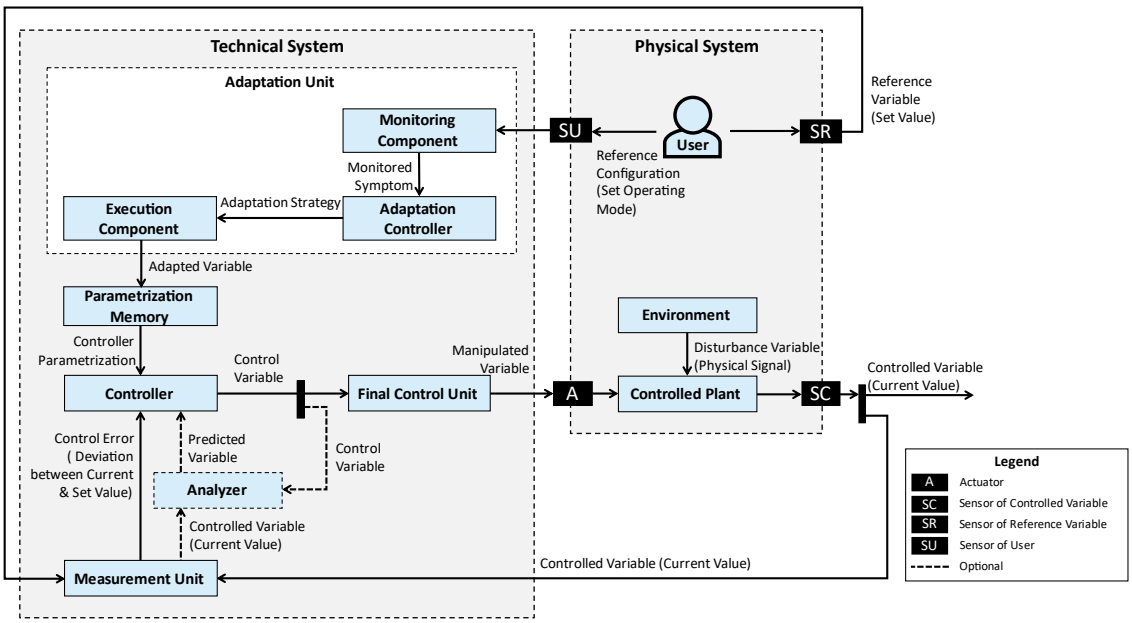


Figure 3.3: Naive Adaptive Control Applied in ACC

An additional subsystem *Adaptation Unit* is integrated into the technical system to realize the expected adaptability. A *Monitoring Component* is deployed in the adaptation unit to catch up with user changes to system configuration, particularly referring to the system's operating modes represented by different controller parametrizations and relying on the data collected by the component *Sensor of User* (*SU* in Figure 3.3). An *Adaptation*

Controller in the adaptation unit determines an adaptation strategy. With the help of an included *Execution Component*, the adaptation strategy is split up into individual adaptation activities. A new value of the adapted variable is forwarded into the parametrization memory component as the outcome of the execution component to update the current value. Thus, the originally saved controller parametrization can be updated at run time in the case of a user's request.

As presented before, the component SU represents additional data access to catch up with the user's request to change the operating mode, which is not included in basic control. In this sense, the technical system's observation scope for the physical system in naive adaptive control is increased compared to basic control.

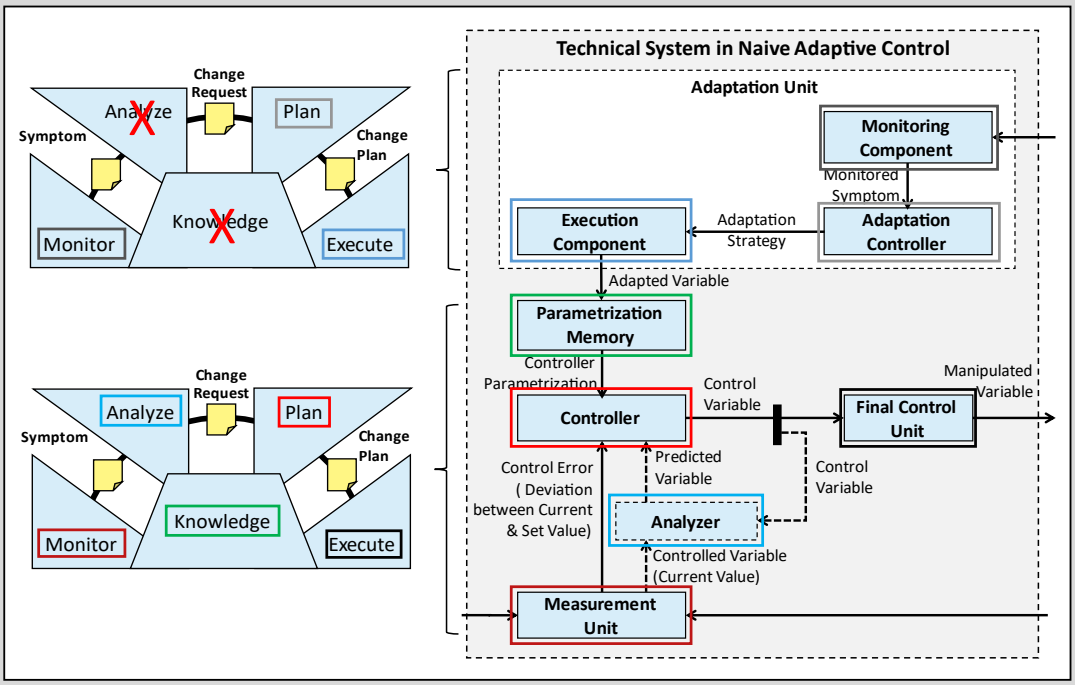


Figure 3.4: Architectural Comparison of Naive Adaptive Control and MAPE-K

The newly integrated subsystem adaptation unit in the naive adaptive control comprises a monitoring component, an adaptation controller, and an execution component. Compared to the MAPE-K reference model, it is indicated that the monitoring component can be roughly mapped as the monitor function of the autonomic manager since both are responsible for data collection and preprocessing to generate the symptom as the same output. The adaptation controller and the execution component can then be roughly mapped as the autonomic manager's plan and execute function. There are two main differences

in the architectural comparison. The first difference is that the MAPE-K model includes the analyze function, which decides whether changes are necessary. However, the adaptation unit has no such decision-making mechanism, since the user makes decisions externally. The second difference is that the adaptation unit does not include a knowledge source function with an updatable knowledge base, which is a significant feature of the MAPE-K. Since the adaptation unit is deployed on a secondary feedback loop instead of on the primary loop, on which the mentioned controller component in the previous section is deployed, another independent MAPE-K structure is added in Figure 3.4. In this case, the primary control loop, especially including the controller component, can be interpreted as the managed system of the adaptation unit.

With the application of naive adaptive control, advanced ACC variants enable the driver to adjust the control strategy by manual selection of provided operating modes like “Eco,” “Comfort,” “Efficient,” and “Sport,” which are respectively represented by different parametrizations of the controller [75]–[78]. For example, the operating mode “Eco” will make the ACC realize a consumption-optimized driving strategy once the driver recognizes that the vehicle does not have much remaining fuel or energy and thus selects the mode. The operating mode “Comfort” delivers a very conservative accelerating and decelerating strategy for the vehicle to maximize the driving comfort and thus also reduce the risk of unsafe slip. Thus, the issues presented in the previous critical scenarios regarding the uneven and wet road surface (cf. Section 3.1) can also be avoided.

The deployment of naive adaptive control has significantly improved the ACC’s adaptability and flexibility. It enables the driver to change the controller parametrization by manually selecting operating mode at run time, which is realized by reloading predefined set values of different application parameters. Unfortunately, such a concept still has its limitations, especially due to the limited number of operating modes, which may not be beneficial to fulfill the driver’s preferences.

By way of example, the driver has selected the operating mode “Comfort”. Nevertheless, the driver also wishes to simultaneously have a more energy-efficient driving strategy later, with an expectation of using the operating mode “Eco” since there is not much fuel remaining. The importance of the operating modes “Comfort” and “Eco” for the driver will still change with reducing remaining fuel or energy during the trip. Thus, selecting a single individual operating mode like “Comfort” or “Eco” may no longer completely fulfill

the driver's diversified requirements. Another critical case may also happen when the vehicle moves on a mountain road with many curves, making the original set value of reference variable like high cruise velocity unreliable due to the high risk of unsafe slip. Frequent human intervention for changing operating mode or repeated adaptation of the set cruise velocity is required in these two critical scenarios. However, such a requirement overstrains the driver's reaction capability and patience while driving.

3.3 Controlled-Plant-Dependent Adaptive Control in ACC

The integration of naive adaptive control enables the ACC to dynamically change the controller parametrization at run time by manually selecting a predefined operating mode according to the driver's request. Such an approach further increases the system adaptability compared to the concept of basic control, which still has limitations in certain critical cases, as discussed at the end of Section 3.2. To overcome these limitations, some advanced ACCs have used a control concept with further improvement by integrating an additional *Interpreting Component* into the adaptation unit. This architecture is shown in Figure 3.5.

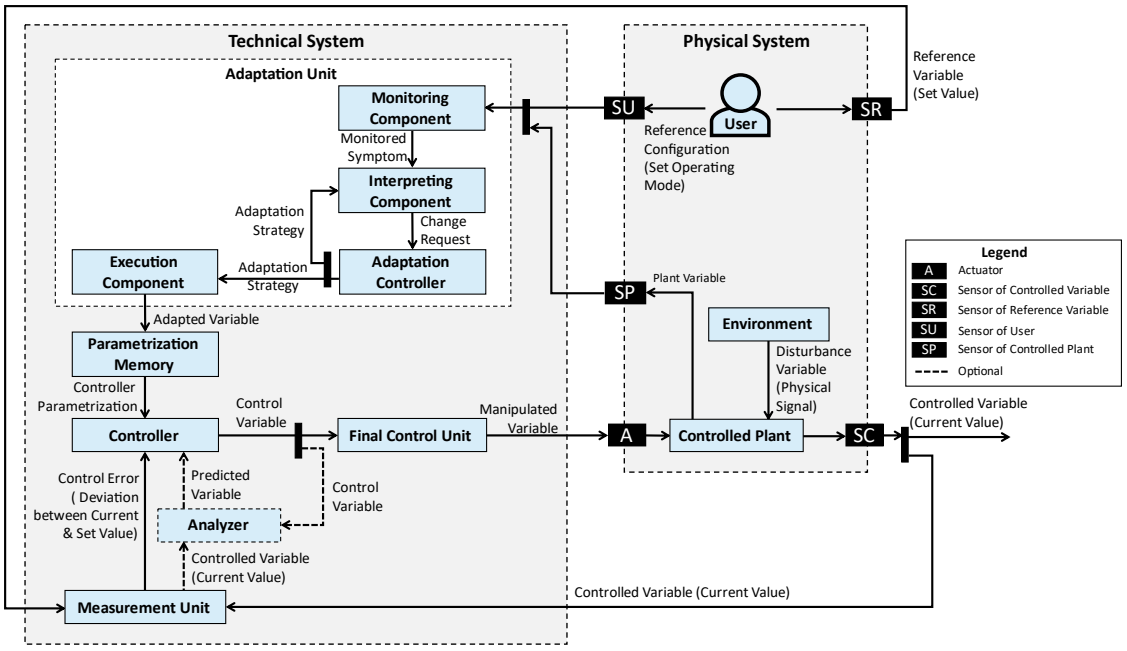


Figure 3.5: Controlled-Plant-Dependent Adaptive Control Applied in ACC

The integrated interpreting component has access to all data forwarded by the monitoring component in such a control concept. As previously mentioned regarding naive adaptive control, the monitoring component still observes and catches up with the driver's requests for changes to the operating mode collected by the component *Sensor of User* (*SU* in Figure 3.5) and preprocesses (e.g., aggregates and filters) the data collected by different sensors. Additionally, the component *Sensor of Controlled Variables* (*SC* in Figure 3.5) delivers current values of controlled variables that represent the state of the impressionable surrounding environment of the controlled plant. In this dissertation, such an impressionable environment is called *Dependent Environment* since it can directly be influenced by the controller's activity (e.g., by maintaining the set cruise velocity and headway in the case of ACC).

Relying on the component *Sensor of Controlled Plant* (*SP* in Figure 3.5), values of plant variables representing the state of the controlled plant, such as the vehicle's engine and gear speed, could also be delivered to the monitoring component. In this case, it is emphasized that the state of the controlled plant also covers the state of its dependent environment, which the component SC can monitor. That means the component SP has a larger monitoring horizon (i.e., more sensory data access) than the component SC. For this reason, the technical system's observation scope for the physical system is increased further compared to naive adaptive control.

Depending on the sensory data forwarded by the monitoring component, the interpreting component estimates the physical system's state, and the result is used to decide whether it is necessary to request the processing of the adaptation controller for planning the control strategy. In the case of a change request, the adaptation controller determines a new adaptation strategy to update the previous one, for example by including a group of set configurations over time such as the controller parametrization for the future. The adaptation strategy is then forwarded to the execution component, which splits the strategy into individual activities consisting of individual values of the adapted variables. Subsequently, the execution component forwards the activities to the parametrization memory component. In this case, both the human user and the adaptation unit can change the system configuration. Such a concept is called controlled-plant-dependent adaptive control in this dissertation.

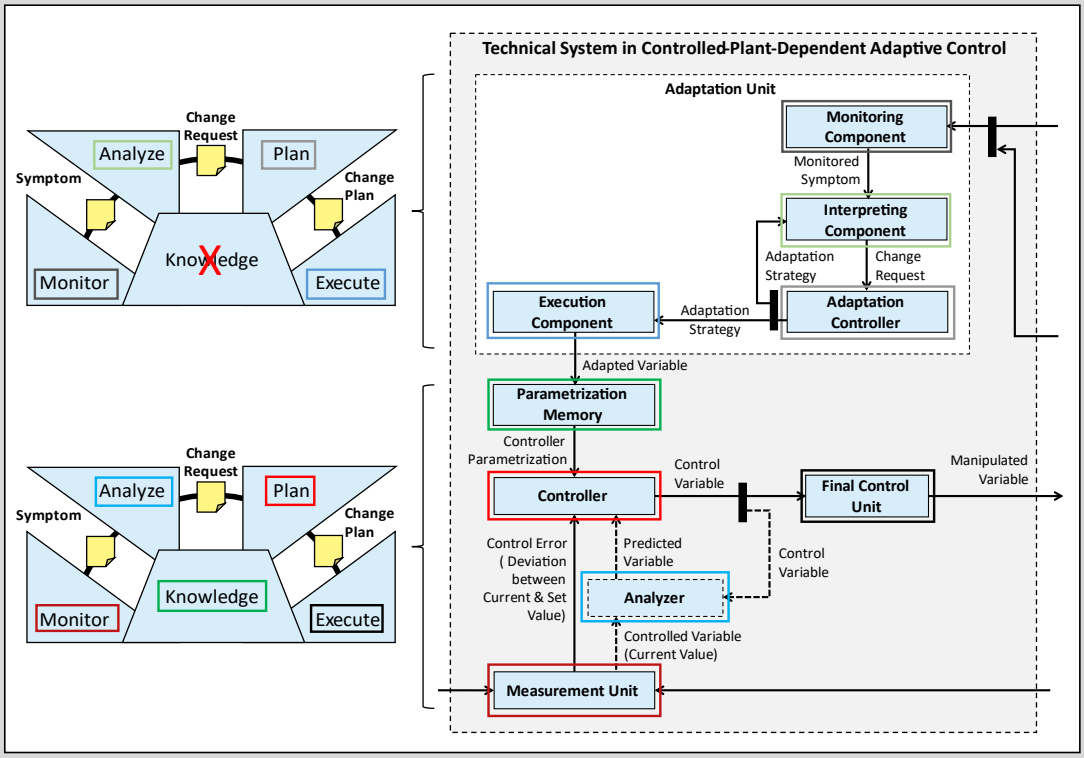


Figure 3.6: Architectural Comparison of Controlled-Plant-Dependent Adaptive Control and MAPE-K

Roughly compared to the MAPE-K reference model, the adaptation unit in controlled-plant-dependent adaptive control is upgraded with a new interpreting component. The interpreting component makes decisions to request the planning of a new adaptation strategy and thus roughly corresponds to the analyze function of the MAPE-K model. Thus, the adaptation unit deployed on a secondary feedback loop instead of directly on the primary control loop in naive adaptive control can be roughly mapped as a summary of monitor, analyze, plan, and execute functions in the autonomic manager. In this case, the primary feedback control loop, including the controller component, can be interpreted as the managed system of the adaptation unit.

Using the ACC based on controlled-plant-dependent adaptive control, both critical scenarios mentioned in the previous section can be eliminated. For example, the driver is not required to repeatedly change the operating mode while driving, like in the case of ACC with naive adaptive control. Instead, such ACC enables the system to automatically select the optimal controller parametrization for the driver based on its state estimation, which is realized by an additional operating mode “Automatic”.

In the mode “Automatic,” it can also be understood that the ACC switches between the operating modes like “Eco,” “Comfort,” and “Sport” without any human involvement. For example, the ACC has detected that the vehicle does not have much remaining fuel or energy with the state estimation based on the variable monitoring. Thus, the system automatically adapts the controller parametrization to avoid an aggressive control strategy, which may cause high fuel or energy consumption. In this case, the vehicle is able to arrive at the planned destination as much as possible. If the driver is not satisfied with the ACC’s automatic decision, they can still directly change to their favorable mode.

In another case, if a vehicle controlled by the activated ACC moves on a mountain road and the driver’s originally preferred driving strategy is too aggressive for the vehicle to move through the curves safely, by relying on the sensory data about the plant variables, ACC with controlled-plant-dependent adaptive control can also perceive the manual steering activity of the driver and thus can correspondingly adapt the controller parametrization to guarantee driving safety.

Relying on the concept of controlled-plant-dependent adaptive control, the ACC becomes able to determine and adapt the optimal parametrization of the controller with consideration of the state of the controlled plant, which covers its dependent environment. Thus, the technical system’s observation scope for the physical system is increased compared to naive adaptive control. However, such a concept still has adaptability limitations due to missing information about the particular environment, which is out of the sphere of the controller’s influence. Due to this reason, such an environment is called *Independent Environment* for the ACCs within this dissertation. A typical example of such an independent environment would be physical disturbances from the driving environment like ambient temperature and wind speed.

Without context information about the independent environment, the determined controller parametrization by the ACC may be extremely critical for the vehicle’s driving in the worst case. For example, the ACC takes over the longitudinal control of the vehicle moving on a mountain road. With high humidity and a low ambient temperature near 0°C, the road surface could be slippery due to ice. An aggressive driving strategy may lead to a high accident risk if the ACC does not know the ambient temperature. Another critical example would be when the ACC controls the vehicle on a route with a temporary

speed limit due to reconstruction work or an accident. In this case, an ACC with a high cruise velocity originally set by the driver would ignore the speed limit without information about the independent environment and thus lead to a high accident risk.

3.4 Physical-System-Dependent Adaptive Control in ACC

As presented in the previous section, the ACC may make inappropriate decisions with a negative control strategy due to the lack of information about the independent environment, which is not directly influenced by the controller's activity. Thus, controlled-plant-dependent adaptive control has been upgraded with additional access to independent environmental information. In this dissertation, such a concept is called physical-system-dependent adaptive control.

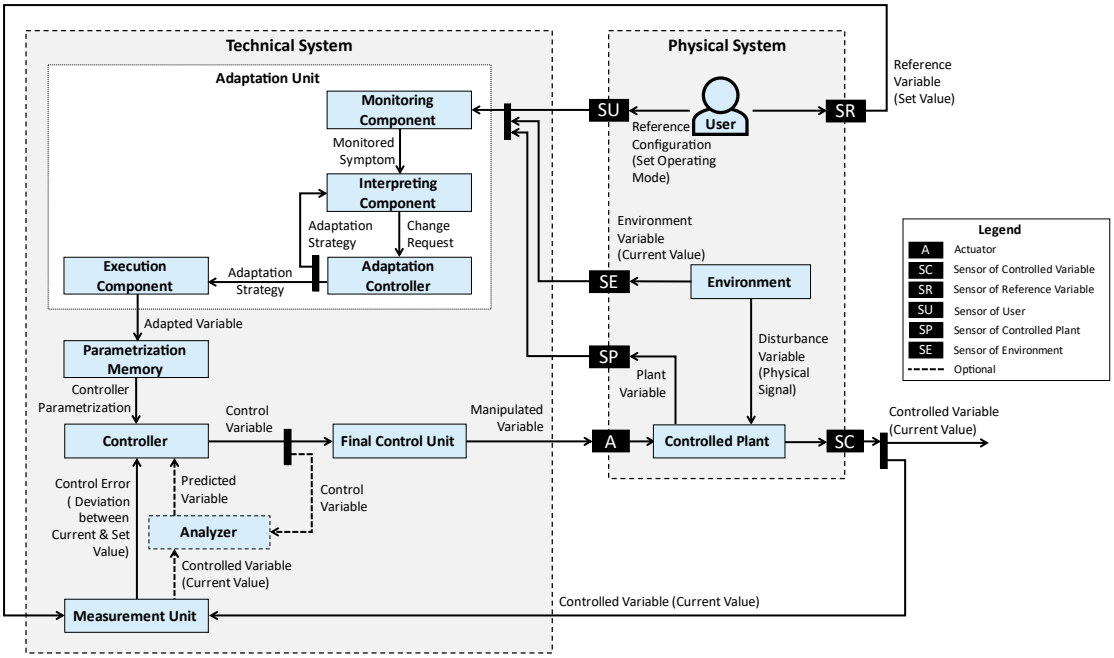


Figure 3.7: Physical-System-Dependent Adaptive Control Applied in ACC

As shown in Figure 3.7, the control system based on physical-system-dependent adaptive control can make decisions considering the state of the vehicle's physical components and their dependent (including controlled variables like cruise velocity and headway) and independent environment. Compared to the architecture of controlled-plant-dependent adaptive control, an additional component, *Sensor of Environment Variable* (*SE* in Figure 3.7), is added to acquire the independent environmental

information, covering states of different environmental variables. Thus, the observation scope of the technical system for the physical system increases again.

ACC with physical-system-dependent adaptive control may use the vehicle's onboard sensors to acquire information from the independent surrounding environment like ambient temperature. As known, information about the future driving environment like the route profile and speed limit is previously saved in the navigation map database. In this sense, such information can be understood as inputs from an onboard virtual sensor on the vehicle and be considered during the determination process of adaptation activity [73][79].

For example, a well-known advanced ACC following the concept of physical-system-dependent adaptive control is called “InnoDrive,” developed by Porsche. In the “InnoDrive,” different operating modes such as “Dynamic,” “Comfort,” and “Dynamic Plus” can be selected by loading different value sets for the weights of criteria like driving comfort, energy-efficiency, and driving dynamics to adjust the style of the driving strategy [80]. Depending on the selected operating mode, the “InnoDrive” uses dynamic programming based on the vehicle's identified state and the state of the dependent and independent environment to repeatedly plan an optimized predictive driving strategy. This predictive driving strategy consists of a location-based trajectory of set cruise velocities for the following route with a deterministic distance horizon, considering the loaded values of weighted factors of criteria [78][80]. Compared to the illustrated architecture in Figure 3.7, the driving strategy planning can be completed, for example, by the adaptation controller in the adaptation unit deployed on the secondary feedback loop of physical-system-dependent adaptive control.

In addition to planning the predictive driving strategy, the “InnoDrive” also includes the functionality of location-based driving behavioral prediction of the preceding traffic, depending on the identified driving style of the driver in the preceding traffic and the route profile. Such a prediction functionality is categorized as a high-level prediction in this dissertation since the high-level environmental information included in the digital map of the navigation system is utilized. Such high-level prediction can be deployed in the interpreting component of the adaptation unit compared to the architecture in Figure 3.7, which is different from the so-called low-level prediction of the component analyzer.

The low-level prediction only focuses on the level of variable data like the controlled variable of headway without any real awareness of the system's surrounding environmental context.

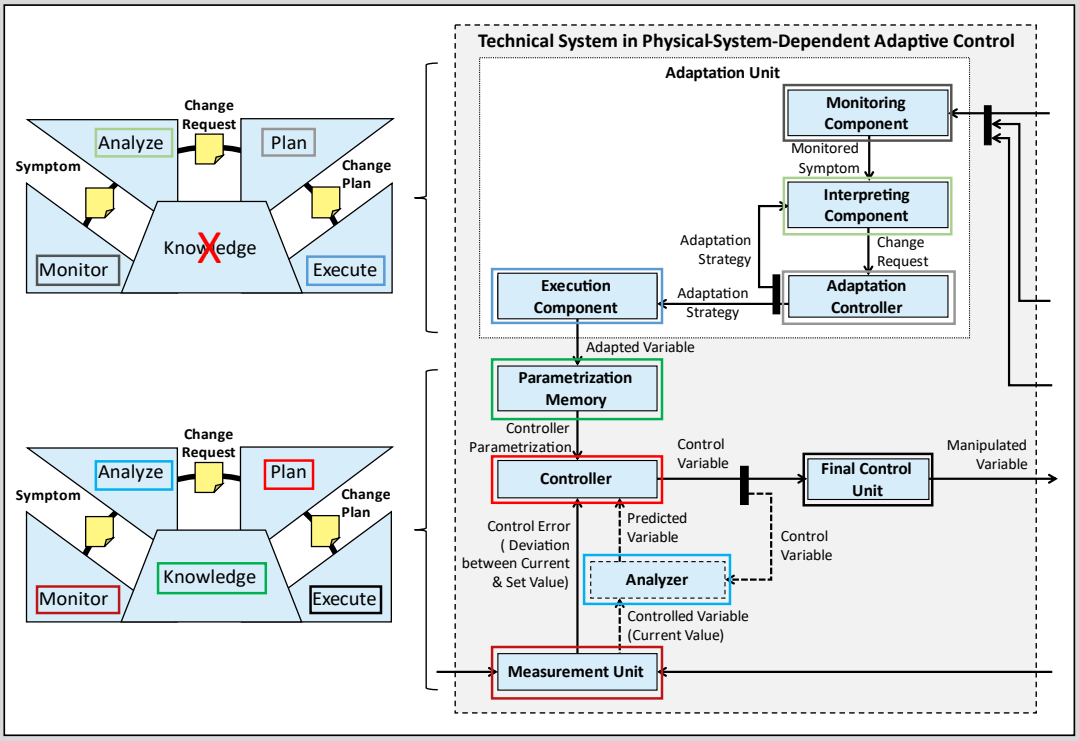


Figure 3.8: Architectural Comparison of Physical-System-Dependent Adaptive Control and MAPE-K

Compared to the MAPE-K, the adaptation unit in physical-system-dependent adaptive control covers the monitor, plan, analyze, and execute functions of the MAPE-K, which works as a secondary control loop for adaptation of the controller deployed on the primary control loop (cf. Section 2.2.7). Nevertheless, a significant remaining difference is that the knowledge source of the MAPE-K is still missing in the physical-system-dependent adaptive control. As is well known, the knowledge source is used to store the high-quality domain knowledge, including the solution topology, policy, and problem determination on higher levels of abstraction. Instead of the knowledge source, the parametrization memory component for storage of values of application parameters on low-level is included. Additionally, the monitor and the execute function in MAPE-K can create new domain knowledge and subsequently update the knowledge into the knowledge source, which is out of the ability of physical-system-dependent adaptive control [72]. Due to the reasons mentioned above, it can be said that the architecture of physical-system-dependent adaptive control still does not completely cover all MAPE-K functions. Instead, only dependencies between the components due to similar functionalities in the architecture and the MAPE-K can roughly be found.

Finally, in the “InnoDrive,” the planned driving behavior of the preceding traffic is taken as a constraint to correlate with the planned predictive driving strategy and then forwarded to the longitudinal control of the vehicle, which is deployed on the primary feedback loop compared to the architecture in Figure 3.7.

With the help of physical-system-dependent adaptive control, the mentioned critical scenario of unsafe slip due to the slippery road surface with ice will be eliminated since the ACC has access to information about the ambient temperature. Thus, it can adjust the control strategy more conservatively by adapting the controller parametrization. However, in some cases, onboard sensors are also insufficient for the ACC to acquire required information, for example, in the mentioned scenario regarding the change of speed limit due to the reconstruction site on the following route.

To work against such a challenge, ACC with physical-system-dependent adaptive control also has a communication ability with external resources to require the support of their information accesses. Thus, it can take the temporary speed limit change into account while planning the optimized driving strategy. With the help of physical-system-dependent adaptive control, the adaptation unit in the ACC completes its adaptation considering the acquired integrated context information about the elements in the physical system (e.g., the controlled plant, the dependent environment, and the independent environment).

3.5 Functional Vision of Future ACCs

The evolution of ACC from the first generation to the following advanced variants with further improvements has been presented in previous sections of this chapter. Different control concepts behind these ACCs have been categorized, analyzed, and discussed to derive their corresponding functionalities and technical limitations. Along with the functional supplementation, it is indicated that the system autonomy of current ACCs, particularly due to stronger adaptability realized by the upgrade of the adaptation unit compared to the first generation, has been strongly increased in recent years. In addition, the technical system has acquired increasing context information about the controlled physical system to determine the adaptation activity, as shown in Figure 3.9.

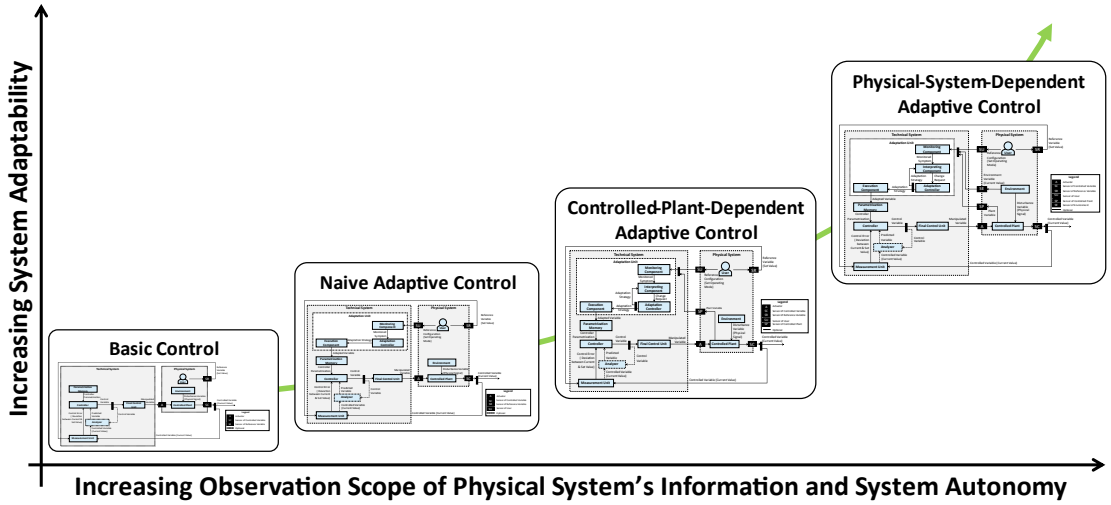


Figure 3.9: Evolution of Control Concepts Applied in Current ACC Variants

Instead of requiring the driver's manual intervention, the latest ACCs (e.g., based on controlled-plant-dependent or physical-system-dependent adaptive control) can already automatically select the optimal operating mode (under the operating mode "Automatic") from given candidate modes based on state estimation and decision-making, relying on the provided input data collected by the sensors from the physical system. Subsequently, considering the selected optimal operating mode, such ACCs can also automatically determine an optimized driving strategy for a certain predictive distance horizon of the following route, including a trajectory of location-oriented set cruise velocities. Although such ACCs have already become much more powerful, there are still large potentials to further improve their performance in the future, which will be discussed in the following sections through two potential future evolution directions.

3.5.1 Personalized ACC by Learning Individual Driver Preferences

ACC is designed to realize semi-automated driving of a vehicle since it only takes over longitudinal control. This means that the driver must still participate in the driving task by taking lateral control with manual steering. Thus, once an activated ACC controls the vehicle, the whole driving task can be interpreted as a process of human-machine collaboration, in which trust and reliance between the human driver and the ACC as typical issues have been investigated by various researchers [81]–[83].

Unfavorable scenarios may happen once the ACC's determined and performed longitudinal driving activity is out of the driver's expectation. For example, the driver may be afraid of possible critical scenarios like accidents due to a lack of reliance once the vehicle enters a curve with a too-high cruise velocity. He may immediately take over the vehicle's longitudinal control from the ACC, thus violating the ACC's original design principle of relieving the driver from routine physical tasks in driving to maximize driving comfort [74]. Thus, how to maximize the reliance and trust between humans and machines becomes an interesting research question. For this purpose, one of the most meaningful solution approaches is to make the ACC "drive" the vehicle as much like a human driver as possible.

Due to their diverse preferences, various drivers drive their vehicles very differently. For example, they may have different styles of acceleration and deceleration, preferred headways to the preceding vehicle, and preferred location-dependent driving velocities along with the route profile, which may even change over time along with context changes of the driving environment such as the weather profile. Thus, it is utopic to deploy the ACCs with one or deterministic numbers of static configurations and serve it as a generalized solution to satisfy all drivers simultaneously. Against such a background, a *Personalized ACC*, designed from completely the opposite direction to the generalized solution by satisfying a single individual driver instead of diverse drivers, becomes a potentially meaningful future solution to enable machine-automated longitudinal driving that is as similar as possible to a human driver.

To realize such a personalized ACC that satisfies a single individual driver, it is impossible to specify an appropriate system configuration at design time due to diverse driving preferences. Instead, the personalized ACC is required to monitor concrete manual driving activities of the observed individual driver with the help of physical variable monitoring, and subsequently, based on the acquired sensory data about the monitored physical variables, to extract and learn the driver's driving preferences on a higher level of abstraction at run time. Thus, the personalized ACC can automatically adapt the individual driver's preferred location-dependent cruise velocity and headway. Additionally, preferred individual acceleration and deceleration style can be automatically adapted so that the driver is not required to switch between different

operating modes. Unfortunately, such a personalized feature is still unavailable in ACCs currently on the market.

3.5.2 Experience-Dependent ACC by Learning Historical Context of Driving Environment

In the previous section, personalized ACC as a potential future variant was discussed. Such ACC would be able to learn the high-level preferences of a single individual driver so that the ACC can “drive” the vehicle as similarly to the individual human driver as possible and thus make the semi-automated driving more reliable for the driver within their expectations. In addition to learning a single individual driver’s preferences, the learning of historical context of driving environment would be another future evolution direction. Such a learning feature must rely on the observed facts by the perception of future ACCs, which capability but has been increasingly strengthened by integrating other sensor accesses like LiDAR and cameras, in addition to the original radar sensor [73][84]–[86].

Considering current ACCs with the control concepts introduced in previous sections, all these ACCs normally determine their control activities only by considering current contextual information of the driving environment (e.g., the perceived input data from the radar sensor). Some advanced ACCs may also consider future context information, either provided by the pre-initialized digital map in the navigation system or possibly by other prediction approaches based on pre-initialized knowledge about the preceding traffic’s driving style and behavior. A typical example of such ACC is the “InnoDrive” by Porsche, as presented in Section 3.4, which repeatedly determines a location-based predictive driving strategy for a certain distance horizon of the following route. If necessary, it also adapts the planned strategy considering the predicted driving behavior of preceding traffic.

In this case, each determination process of the “InnoDrive” is independent without any dependency on previous determination processes, which means that the observed historical context information of the driving environment during the trip is completely ignored. Such an approach is inconsistent with the temporal causality in reality, which

thus possibly leads to critical scenarios since historical context may also influence the current and the future context in the physical world's reality.

For example, a vehicle is being controlled by the ACC and moving on a mountain route with many curves. The ACC has detected a preceding vehicle and adapted its planned driving strategy until the end of the considered predictive horizon represented by a GPS position on the following route. Once the ego-vehicle has reached the GPS position, the ACC again begins to plan a predictive driving strategy for the next following horizon. In this predictive driving strategy, the ACC will “forget” the previously detected preceding vehicle since the ACC, due to a curve, currently cannot perceive the preceding vehicle, even when it could suddenly appear again after the curve. Thus, a high risk of accident is created.

To eliminate such an issue, it would be great if future ACC could “remember” its observed facts that happened during the trip, such as, the “disappearing” preceding traffic by relying on memory ability. In this dissertation, the ACC with such a feature is called *Experience-Dependent ACC*. It refers to ACC that can record the observed facts and, subsequently, extract knowledge on a higher level of abstraction from the observed facts and learn the knowledge as its own experience. Thus, the future ACC can continually strengthen its experience about the driving environment and thus become increasingly more intelligent and adaptive to the previously mentioned critical scenarios. Unfortunately, the features included in the described experience-dependent ACC are still not available in current ACCs on the market, which could be crucial to realize in the future.

3.6 Opening Issues of Current Control Concepts for Future ACCs in the Functional Vision

The personalized and experience-dependent ACCs were presented in previous sections as functional visions of future ACCs. However, such functionalities have not yet been covered by current published serial ACC variants on the market. Thus, this dissertation tries to identify the reasons for this phenomenon by investigating existing issues in the current concepts of automatic control behind current ACCs at the architectural level. More details about this investigation will be presented in this section.

3.6.1 Missing Knowledge Acquisition and Adaptation

Considering the presented functionalities of personalized and experience-dependent ACC, it is indicated that both variants as visions require that the future ACC system be able to deal with high-level domain knowledge. Typical examples of such high-level knowledge could be the driving preferences of individual drivers and the system's experienced environmental context like the previously "seen" but currently "invisible" preceding car on the mountain road with many curves, as previously introduced examples in Section 3.5.2. Most automatic control systems work fundamentally at a very low level of abstraction with concrete data. Thus, an appropriate representation of the domain knowledge and corresponding abstraction mechanism from the low level of data up to the higher semantic level becomes significantly important for the control systems in the future.

3.6.1.1 Current Domain Knowledge Modeling in Control Systems

Considering concepts of automatic control systems presented in Section 3.1–3.4, the adaptation unit, as an essential subsystem in the technical system (especially relying on its included interpreting component and adaptation controller), analyzes and determines an optimized adaptation strategy, for example in the case of ACC for adapting the controller parametrization. For this purpose, a physical system model, including the required domain knowledge, needs to be deployed in the adaptation unit to enable a faithful adaptation. Thus, it is aware, for example, what kind of an influence on the final control performance the adapted controller parametrization has.

The interaction between technical and physical systems in automatic control strongly relies on physics. For this reason, physical formulas based on algebraic functions with the physical variables are chosen as a kind of domain knowledge representation to describe the behavioral relationship between input and output physical variables. This approach guarantees great generalization potential and is categorized as physical modeling in this dissertation. However, considering this from another perspective, such a physical modeling approach requires explicit knowledge about the domain's physical behaviors at design time. Thus, the human engineers' knowledge quality and completeness strongly influence the system's performance.

Generally, it is quite difficult for domains consisting of physical processes with high complexity, like in the automobile branch, to precisely model processes solely by following explicit physical formulas. Thus, data-driven modeling is an alternative approach to physical modeling, aiming to fulfill such a challenge. Data-driven modeling tries to describe relationships between related variables directly based on available real data points and corresponding human mathematical approximation for data interpolation within the covered value range of the data points (e.g., by using the characteristic diagram as a simplification with assumed deterministic behavioral representation). An explicit understanding of the physical processes is not required in data-driven modeling, which makes the modeling much easier. However, from another perspective, such a modeling approach has a limited generalization potential due to its properties of approximated interpolation between the data points. In addition, a deviation of the modeled behaviors to the ground truth behaviors (in reality) is inevitable, particularly in the data's extrapolation area where the value range of the measurement data is not covered.

Both physical and data-driven modeling formulate domain knowledge by following the so-called closed-world assumption, which assumes a quasi-static or at least predictable world between sensing and acting [87] and assumes that the models contain all required knowledge about the physical system. Along with the expected higher system complexity and flexibility, the complexity of the physical system to be considered in the system design also increases, which strongly challenges the knowledge reserve of development engineers.

Looking at the presented personalized and experience-dependent ACC, such a closed-world assumption is utopic in future system design. The development engineers cannot know a single driver's driving preferences or the individual system's experienced driving environmental context in advance. Additionally, the model behavior (due to the closed-world assumption after the system development at design time) remains static. Such a feature makes the model unable to update at run time. Thus, different time-dependent behavioral changes of the modeled physical system (e.g., due to aging of the vehicle or season-dependent vehicle modification like tire changes) would be ignored. Against

such a background, the conventional modeling approaches for the physical system based on the closed-world assumption reach their limit and become slowly unfavorable.

3.6.1.2 Essential Knowledge Acquisition and Adaptation as Vision

To overcome the modeling limitations of the closed-world assumption, an approach for knowledge acquisition and adaptation by following the open-world assumption, which enables the system to update its available domain knowledge, becomes increasingly important. From the viewpoint of software engineering, knowledge acquisition and adaptation can be interpreted as a kind of configuration adaptation activity by the control system itself. This expected property of self-adaptation is exactly the strength of MAPE-K, which relies on its monitor, execute function, and knowledge source that are responsible for knowledge creation, updating, and storage respectively [46].

As discussed earlier, current control systems already include monitoring and the execution components, which means that an additional component for knowledge storage must still be integrated into the adaptation unit. With the help of this component for knowledge storage, it is emphasized that the acquired and adapted knowledge is no longer limited to the concrete level of sensory data, which could also exist on higher levels of abstraction. Thus, the other components like the interpreting component and adaptation controller must also be upgraded to utilize higher-level knowledge. Compared to the MAPE-K, it can be deduced that future control systems with the properties of knowledge acquisition and adaptation will acquire properties of the real MAPE-K, covering all monitor, analyze, plan, execute, and knowledge source functions.

In the architectures of current control systems, the primary control loop (including the controller) is responsible for the real-time interaction with the physical world (e.g., the vehicle’s longitudinal control in the case of ACC). In the worst case, it would quickly become critical for the control task even within milliseconds if driving safety is violated. For this reason, the primary control loop usually works within the millisecond range with strictly limited timing constraints. Thus, its observation scope of information is also limited, only focusing on determining the control activity for the current time point or a certain limited predictive time horizon.

Independently of the primary control loop, the adaptation unit is deployed on a parallel secondary control loop with another cycle time. Without the requirements of real-time interaction with the physical world, the secondary control loop is not as time-sensitive as the primary loop. With such understanding in mind, the secondary loop for knowledge acquisition and adaptation can be designed to work with a much longer cycle time and lower time resolution, possibly considering a larger observation scope of information.

On a lower component level, knowledge acquisition and adaptation means that the physical system's model must be retrained repeatedly at run time. In this dissertation, such a property of repeatedly retraining the model is called self-learning, which is unfortunately still beyond the ability of current control systems. Aiming to realize the self-learning property of the physical system's model, artificial intelligence (AI) approaches, especially from the subfield of machine learning (ML), came into the investigation focus of researchers due to their strong data-driven learning capability for the model. In these approaches, the self-learning process is driven by an appropriate training algorithm (e.g., in the case of a neural network model).

Along with the further development of artificial intelligence, researchers have already tried to apply different machine learning approaches in the control systems, such as neural networks [88][89]. As a common understanding in the control theory, such control concepts with artificial intelligence are roughly categorized into the subfield of intelligent control [34][90]. Still, they have yet to be deployed in the current ACCs on the market.

3.6.2 Limited System Scalability against Fixed Boundary Conditions

In addition to knowledge acquisition and adaptation, the presented personalized and experience-dependent ACCs also constitute other present issues for the system concept. Both variants of future ACC should be able to deal with high-level domain knowledge. Since high-level knowledge depends on diversified low-level sensory data, the future ACC system requires increased sensor access to gather more contextual information about the entire physical system for the adaptation unit. However, the system's boundary conditions, especially the available hardware infrastructures like the engine control unit's (ECUs') computation capability, are also fixed from another perspective.

Along with the integration of increased sensor access, it can be understood that the secondary control loop in the future personalized and experience-dependent ACC, where the adaptation unit is deployed, will have much more sensory data to be processed. Additionally, the expected processing capability with high-level domain knowledge also requires that the secondary control loop include appropriate mechanisms relying on heuristic or linguistic methods, which means that the worst-case computation time would become longer and nondeterministic. Thus, it would become very difficult for development engineers to design the system scheduling.

As presented earlier, automatic control systems like the ACC on a vehicle, which require real-time processing capability, are designed as pure embedded systems. In this case, such systems react to changes in their surrounding environments, and their corresponding components can be denoted as independent “active objects” or processes with a particular running cycle time [20]. For this reason, the synchronized method calls for communication with external domains is rather used somewhat rarely.

Although in such a case, it does not mean that there is no interdependency between different control loops. The lower primary control loop where the controller is deployed still relies on the inputs provided by the upper secondary control loop, where the adaptation unit is deployed (cf. architectures in Section 3.1–3.4). For this reason, an excessive long worst-case computation time of the secondary control loop strongly influences the scheduling of the primary control loop. It may limit the lower bound of the primary control loop’s permitted cycle time, which thus negatively influences the real-time interaction with a minimum required frequency between the primary control loop and the physical world (e.g., in the case of the vehicle driving).

Additionally, the excessive long computation time may also limit the connectivity of the primary control loop with the outside world. Thus, the system cannot acquire essential external sensory data with the required high time resolution due to the limited lower bound of the cycle time. Further, the control performance of the overall control system may also be negatively limited once there are no sufficient sensory data inputs. Such a case only describes a critical situation between two concurrent control loops in the architecture. If the system includes more than two control loops, interdependencies between the control loops would be much more complicated. The constraints of system

connectivity for the lowest loop, which come from the multiple upper loops, would also become much more critical.

As a pure embedded system, a control system like the ACC is deployed on the vehicle's ECU with limited computational resources. One potential solution is to eliminate the excessive long computation time barrier by integrating more powerful hardware computation units like a high-end graphics processing unit (GPU) and field-programmable gate array (FPGA). Thus, more-complicated computation processes on the upper control loops based on the heuristic and linguistic methods could be completed within an expected cycle time.

However, from another perspective, the system complexity and flexibility of the ACC are also continually increasing, although computing capability (following Moore's law) has massively improved in recent years. In this case, a conflict between the fixed system boundary, particularly the limited onboard computing resource, and expected stronger computing and connectivity slowly becomes a significant bottleneck in further development. Against such a background, the external support of offboard computing resources like cloud computing could be an alternative approach, which researchers have already investigated, but which still has not been deployed in ACCs on the market [91]–[94].

Generally, from the viewpoint of software engineering, the issues mentioned above can be interpreted as a conflict between the expected high system scalability of the future control system and its fixed boundary conditions. The hardware solution mentioned earlier of integrating more powerful computing devices, aims to improve the boundary conditions directly. Another potential solution would be to improve the software system's architecture, aiming to reduce the interdependencies between the loops by loose coupling instead of focusing only on the hardware environment. Unfortunately, such a solution increases system scalability but may also constitute further challenges for the system's architecture design. For this reason, this dissertation focuses on a higher meta-level of system architecture design and tries to identify relevant challenges, which will be covered in the next section.

3.7 Challenges for Architecture Design of Future Control Systems

3.7.1 Current Design of Hierarchical Control System Architecture

The introduced architectures of current automatic control systems (e.g., in the case of a vehicle's ACC variants) clearly indicate that the whole system consists of several concurrent control loops. In this case, each control loop can be interpreted as an individual layer. Thus, considering the viewpoint of software engineering, current automatic control systems are built based on the well-known hierarchical layered architecture pattern [61], which is not actually a completely new topic for the field of control theory.

To deal with increasing system complexity, the field of control theory has proposed approaches of hybrid hierarchical architecture for advanced control systems with sophisticated world models since the 1990s (e.g., for autonomous robots). In these approaches, the field focuses more on controlling technical processes than on configuration control[6] [95][96]. By following the principle of increasing intelligence with decreasing precision, the top-level control task is hierarchically decomposed into a group of distinct subtasks on the next lower level, relying on the assumption that the dynamics of the world decrease with the level of abstraction [2][87]. Thus, a successive delegation of duties by determining and forwarding the reference control strategy exists from the upper to lower levels.

Such hybrid hierarchical architecture of advanced control systems normally consists of three parts with different mechanisms for deliberative computation, reactive plan execution, and reactive feedback control, from the top layer to the lowest, respectively [97]–[100]. The whole system construction is similar to the three levels (skill-, rule-, and knowledge-based) of the cognitive model of Rasmussen for explaining human interaction behaviors [101][102]. Some typical examples introduced in this dissertation are the LAAS architecture [43] and the hybrid control architectures of Yavuz and Bradshaw [44] (cf. Section 2.2).

[6] Configuration control refers to the processes of configuration (e.g., during initial control application set up) and reconfiguration (e.g., when a control application is changed) [95], in which adaptation of component configuration, like controller parametrization or knowledge acquisition and adaptation is included.

The reactive feedback control at the bottom of the architecture computes based on non-symbolic algorithmic methods and thus is designed as a reactive layer in the system architecture. The system part with deliberative computation possibly includes one or several upper layers at the top of the whole architecture based on highly symbolic computation, focusing on physical system behaviors at different levels of abstraction. A sequencing middle layer involving a reactive planner is deployed to realize seamless communication between the bottom and upper layers. It selects and executes appropriate tactics, including a group of pre-written ordered sets of actions. Based on the execution of the actions, appropriate behaviors to accomplish the subtasks are either de- and activated or terminated [103].

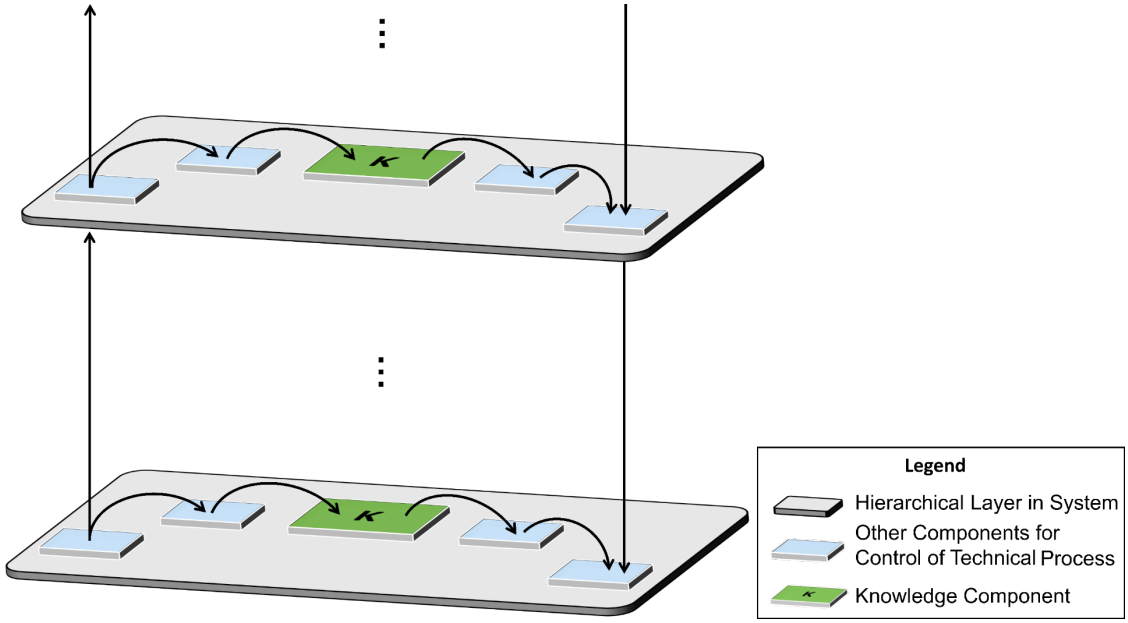


Figure 3.10: Knowledge Coupling on Different Layers within Current Hierarchical System Architecture Design [18][39][40]

Along with the hierarchical layers in the system architecture, the world model is also separated as concurrent parts to model the physical system behaviors on different levels of abstraction (e.g., deployed in hierarchical knowledge bases of Meystel's hierarchical nested architecture) [18]. In this dissertation, such an approach to knowledge decoupling significantly reduces the complexity of the world model on each layer by eliminating the need for a sophisticated "monster" world model with incredible high complexity. Thus, a long processing time and the risk of violating corresponding time

constraints for the fulfillment of the system's required real-time capability are effectively avoided, as shown in Figure 3.10.

3.7.2 Limitations of Knowledge Decoupling Approach in Current Design

As presented earlier, the sophisticated "monster" world model is avoided through knowledge decoupling by division into multiple world models, which are correspondingly deployed on different layers in the hybrid hierarchical architecture and describe physical system behaviors at different levels of abstraction. Thus, relying on parallel computations of the concurrent layers, the timing performance of the whole control system can be increased. However, from another perspective, such an approach still has strong limitations for expected future features.

Considering the hybrid hierarchical architecture, the world models on different layers describe the physical system behaviors on different levels of abstraction. Due to required levels of abstraction, they may rely on different knowledge representations, varying from modeling with linguistic methods like formal languages to the previously mentioned explicit physical modeling or implicit data-driven modeling such as a neural network. In this case, the expected feature of knowledge acquisition and adaptation constitutes challenges for system architecture.

For example, it is known that knowledge acquisition and adaptation extract new domain knowledge based on newly observed facts and subsequently integrate the new knowledge into the world models through learning processes. In this case, the learning process can be realized by adapting the data field in the knowledge base of the knowledge component on a certain layer. Since the world models are deployed and work independently on different concurrent layers, as shown in Figure 3.10, their learning processes are independent.

Nevertheless, a knock-on effect of the newly learned knowledge on a certain layer to the validity of domain knowledge included in the world models on neighbor layers (upper and lower) may exist. The layer-independent learning processes become idealized since the domain knowledge may still have a strong working dependency across different layers. In such a case, vertical adaptation of all knowledge components through all layers is required, leading to an unfavorably high computation effort for the control

system. Thus, a strong challenge to the limited computation resources of control systems deployed in an embedded running environment is raised (e.g., for the vehicle's ECU in the case of an ACC system). To work against such a challenge, how to enable an efficiently interactive self-learning process between neighbor layers becomes a meaningful research topic for system architecture design.

In addition to the challenge of unfavorable required vertical adaptation through all layers, another challenge relates to increased computation time. Adapting the world model for learning the knowledge included in the newly observed facts may further increase the model complexity. Higher model complexity may require a larger knowledge base in the knowledge component and a longer accompanying inference process relying on the knowledge base. Thus, the longer inference process limits the permitted processing time of other components on the loop and may negatively influence computation performance of the whole closed control loop since the overall computation time of the loop, due to interaction with the physical world is always limited. Against such a background, enabling knowledge acquisition and adaptation without negative influences on the computation of the closed control loop in the system architecture design becomes an interesting research topic.

3.7.3 A Vision of Architecture Design for Future Control Systems

Considering the challenges discussed in Section 3.7.2, especially for integrating the knowledge acquisition and adaptation, the most serious pain point lies in the deployment of the knowledge component, including its corresponding knowledge base and world model on the closed control loop of each layer. Thus, the deployment concept of the knowledge components needs to be rethought to fulfill the previously mentioned challenges, considering the influences of knowledge acquisition and adaptation.

Aiming to fulfill the challenges introduced in the previous section, one of the best approaches in designing future control system architecture is to remove the knowledge component from the closed control loop. Thus, the knowledge component has a loose coupling with the closed control loop, and acquisition and adaptation regarding the knowledge component can perform independently. In this case, the knowledge component is deployed as an interface between every two layers. The original closed

control loop, consisting of components for the technical process control, is still deployed on each layer. Thus, the basic control cycle, referring to information flow on the closed control loop for the technical process control, is separated from the so-called knowledge cycle. The knowledge cycle here refers to the information flow for the knowledge acquisition and adaptation in this dissertation, covering sensory data acquisition, extraction of new domain knowledge based on the acquired data, integration, and deployment of the new domain knowledge.

As shown in Figure 3.11, each two neighbor layers have a shared knowledge base, which means that each knowledge component may contain domain knowledge on two different levels of abstraction. As interfaces between each two neighbor layers, the knowledge components may also allow direct exchanges of data or domain knowledge mutually, relying on underlying communication paradigms of distributed systems. Such a design fulfills the previously discussed challenge of knock-on effect due to newly learned knowledge on one layer for domain knowledge validity on corresponding neighbor layers.

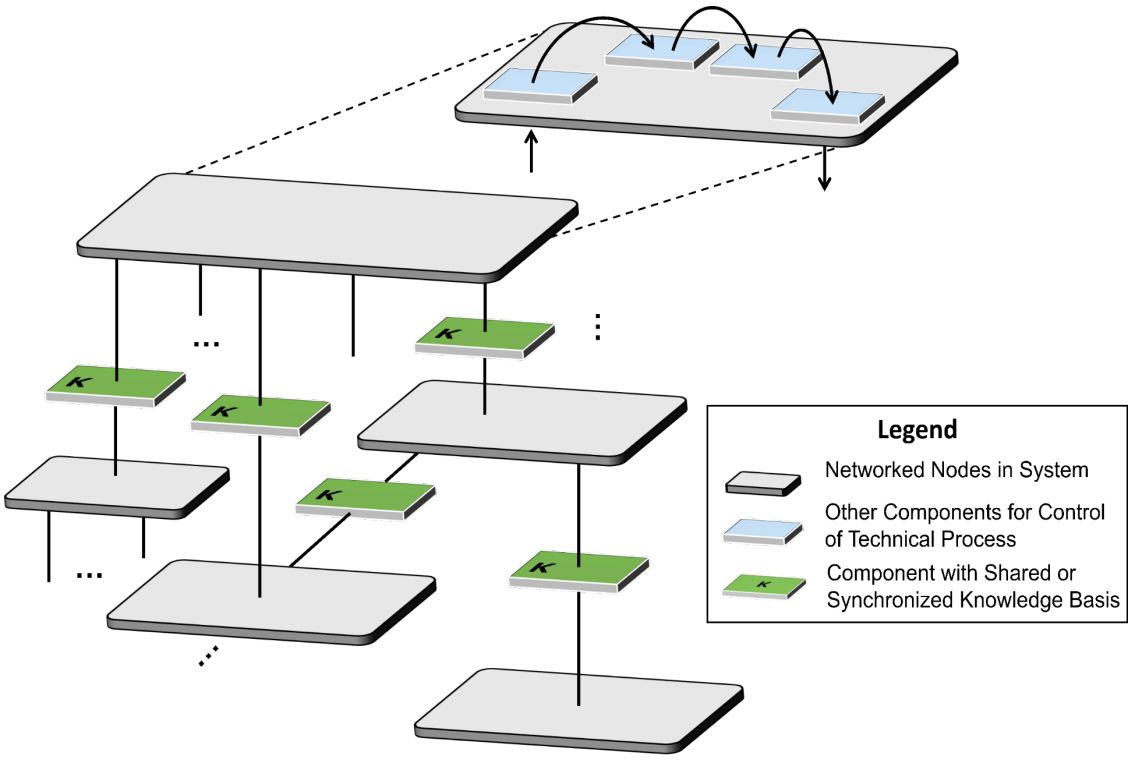


Figure 3.11: Preliminary Idea of Multidimensional Networked Architecture for Future Automatic Control

Following the concept illustrated in Figure 3.11, it is indicated that the hierarchical layers (grey layers), including closed control loops consisting of a group of components for technical process control, are networked through corresponding knowledge components (green components) in the overall unified system architecture. Since a loose coupling between the layers relying on the isolation of knowledge components has been realized, such system architecture can be interpreted with a networked topology. Each previous layer can be generalized as a node in the network, and the knowledge components play the role of interfaces between the nodes. In this sense, the hierarchical layered topology is only considered an instance of the networked system architecture in this dissertation. For this reason, the previous terminology of “layers” is replaced by “nodes” in the following paragraphs.

This concept of system architecture design can be further generalized. The group of components for control processing on the closed control loop can be interpreted more abstractly as a software module. In this dissertation, each software module is defined as a functionally closed unit consisting of single or multiple building blocks in a software system, solely completing a certain functionality from the viewpoint of software engineering [15]. Thus, seamless integration of different functions within an overall software architecture can also be realized by the proposed preliminary ideas in Figure 3.11. The system decomposition following the illustrated preliminary idea of architecture design in Figure 3.11 can consist horizontally of different components and consist vertically of different modules realizing corresponding functionalities. From this point of view, the proposed concept and its preliminary ideas can be illustrated as a kind of multidimensional networked architecture.

However, many detailed questions must still be answered to design such a sophisticated system architecture with the proposed networked construction. For example, the knowledge component in the current architecture design is applied to complete an inference process. Thus, the control system can make decisions for control activities by relying on the observed facts and the domain knowledge included in the world model of the knowledge component. Since the knowledge component does not directly participate in the closed control loop in the proposed new architecture design concept, its original inference task should be taken over by other appropriate components in the

control loop to complete the technical process control. Thus, the functionalities of the components staying on the closed control loop must be reconsidered during the system design.

Additionally, some typical questions for the design of hierarchical system architecture may also need to be reconsidered. For example, a tradeoff decision between precision and level of abstraction of the domain knowledge included in different knowledge components must be made. Along with a higher level of abstraction, the precision of the domain knowledge decreases. The decreased precision may limit the performance of the planning control activity, depending on the context of expected control tasks of different nodes.

Further, a similar architecture design challenge exists for determining the observation scope of information on different nodes (e.g., the time horizon of the control strategy). A longer time horizon due to included extensive information guarantees that the planned control activity has a greater performance potential for the control system, which but due to a higher risk of uncertainties in the future leads to a risk of performance violation.

The distribution of sensor access on different nodes must also be a focus during the architecture design of hierarchical control systems. More access to sensory data helps the nodes acquire more information about context but strongly challenges computation capacity limits. Against such a background, identifying the global optimum for the design of the overall architecture with consideration for different influencing factors together still represents a challenge.

3.8 Summary: Future Automatic Control—Artificial Cognitive Control

In previous sections, current control concepts applied in ACC examples were introduced, which are used to derive functional visions for the future. The existing issues with current control concepts for realizing functional visions were discussed from the perspectives: (1) missing knowledge acquisition and adaptation, and (2) the limited scalability against fixed boundary conditions. Limitations of current approaches for architecture design of hierarchical control systems were also discussed, aiming to investigate the root causes

of these issues. Finally, a vision of future architecture design with some preliminary ideas was given, in which some detailed initial questions were also discussed.

In this dissertation, a new category of automatic control named *Artificial Cognitive Control* is proposed to better understand future control systems following from preliminary ideas. The term artificial cognitive control is originally inspired by the human cognition process, referring to the mental action or process of acquiring knowledge and understanding through thought, experience, and the senses [104].

The term “cognitive control” is not completely new in research. In the fields of psychology and neuropsychology, diverse definitions for this term have been published from different perspectives over a long period [105]–[107]. One of these definitions, proposed by Feldman and Friston [108], is concluded from the probabilistic view of the environment. These researchers defined the cognitive control of the human as using the brain to continually optimize the probabilistic representation of the environment, relying on attention, which refers to a perception system for extraction of available information out of noisy sensory measurements. From the viewpoint of information theory, cognitive control thus can be understood as an activity aiming to minimize the information entropy since entropy is a measurement of the uncertainty of a variable [109][110].

Based on the perspective of information theory, another definition of cognitive control from the engineering perspective was proposed by Haykin et al. [111]. In this dissertation, this definition is taken as a reference. Haykin et al. [111] define cognitive control as adapting the directed flow of information from the perceptual part of the system to its executive part to reduce the information gap. Thus, reducing the information gap is equivalent to reducing the properly defined risk functional for the task at hand, with the reduction having a probability of close to one. Here, the information gap is generally understood as the lack of contextual information and domain knowledge about the physical system, which is also directly relevant to information entropy.

Following this understanding, the expected integration of knowledge acquisition and adaptation in future automatic control systems with stronger system scalability and connectivity can thus be understood as approaches to reduce entropy by supplementation and augmentation of domain knowledge and context data or information about the physical system. Such a feature clearly shows that the term

"cognitive control" is appropriate for the definition of a future control system. In this dissertation, the following functional requirements are defined to provide a better understanding of artificial cognitive control:

- Artificial cognitive control shall be able to acquire low-level context data and high-level context information about the physical system (including user, controlled plant, and environment), relying on system connectivity to communicate with available distributed (onboard and offboard) perception resources[7] (e.g., sensors or perception systems with appropriate interpretation mechanisms).
- Artificial cognitive control shall be able to make decisions to determine control strategy by analysis of acquired context data and information and by manipulating actuators to complete the required tasks for the control of the technical process.
- Artificial cognitive control shall be able to automatically analyze and identify the current situation based on acquired context data or information. Further, it shall be able to utilize its available domain knowledge to adapt its configuration (i.e., changing system configuration like component connections or component configuration like values of application parameters to guarantee the performance of the technical process control).
- Artificial cognitive control shall be able to acquire[8] and automatically adapt its domain knowledge about the physical system. Thus, it can continually improve the performance of technical process control.

In Section 3.6, it was already stated that artificial cognitive control systems would acquire the properties of a real MAPE-K by integrating knowledge acquisition and adaptation, which will then roughly cover all MAPE-K functions: monitor, analyze, plan, execute, and

[7] The perception resources can be data providers like pure sensors, but they also be data service providers that strongly rely on the interpretation mechanisms deployed on the sensors (e.g., object-recognition algorithms).

[8] The domain knowledge could be either self-created based on acquired context information or directly obtained from external domains.

knowledge source. Thus, it can be concluded that the automatic control system is increasingly evolving as a self-adaptive system with the properties of autonomic computing (cf. Section 2.2.7).

The software engineering field has proposed holistic approaches for designing self-adaptive systems with sophisticated architectures. Against such a background, this dissertation takes the field of control theory and software engineering together, aiming to investigate how software engineering approaches can benefit the architecture design of future sophisticated automatic control systems. Thus, the architecture design challenges mentioned above (cf. Section 3.7) can be overcome. Finally, a generic architecture style for the architecture design of automatic control systems, covering current control concepts and particularly artificial cognitive control, is proposed from the viewpoint of software engineering as the main contribution of this dissertation. A more comprehensive introduction to the generic architecture style will be provided in Chapter 4. Based on this generic architecture style, an example architecture for artificial cognitive control will be further derived as an instance in more detail in Chapter 4.

4 A Generic Architecture Style for Designing Automatic Control Systems

In Chapter 3, current concepts of automatic control were introduced based on a case study with application examples of ACC. Issues with current concepts due to two perspectives, including missing knowledge acquisition and adaptation, which is interpreted as a kind of self-adaptation activity in this dissertation, and the limited system scalability against fixed boundary conditions, were discussed. Based on the issues regarding both perspectives above, architecture design challenges for future control systems with sophisticated architectures were also discussed. A vision with preliminary ideas for future automatic control systems' architecture design was introduced to address the challenges. Finally, artificial cognitive control by following the preliminary ideas, aiming to eliminate the issues facing current concepts, is briefly defined as the concept of future automatic control in the next generation at the end of Chapter 3.

As discussed at the end of Chapter 3, the artificial cognitive control system is evolving toward a self-adaptive system that the software engineering field has focused on for a long time. Thus, the architecture design of automatic control systems can benefit from relying on established holistic approaches for designing sophisticated self-adaptive systems. For this reason, this dissertation takes the fields of control theory and software engineering together and tries to understand automatic control systems from the viewpoint of software engineering. Finally, a generic architecture style considering software engineering is proposed as the main contribution in this dissertation, which is also included in this chapter.

4.1 Control Theory Meets Software Engineering

Examining the research work of recent years reveals that it is not a completely new idea to bring control theory and software engineering together. Researchers have investigated how software engineering approaches and control theory concepts could mutually benefit. For example, some researchers studied on the development process level in a dissertation on how to design self-adaptive software systems with formal

guarantees on desired properties and behaviors, which are available in the design of conventional feedback control systems [112].

Another dissertation focused on the control-theoretical software adaptation [113]. It found that linear models are mostly used to represent the behavior of the software. However, the behavior is considered highly nonlinear, far from real-world applications. Furthermore, the dissertation argued that classic controller guarantees are poorly exploited when engineering control-based solutions to guarantee the adaptation goals. Thus, appropriately linking control-theoretic concepts to guarantee software quality is still a challenging research topic. The same statement is also included in another paper [71], in which the feedback control loops were interpreted as the MAPE-K loop for autonomic computing. This research noted that the mapping from high-level adaptation objectives in terms of QoS[9] or SLO[10] and abstract models towards lower-level effective actions on the managed system is still missing. Additionally, a more complicated system involving multiple control loops, which may have intertwined interferences due to composition and coordination, is also a difficult and hardly tackled question.

In addition to the theoretical investigation of the system architecture, control theory approaches have also been applied in software engineering of self-adaptive systems within diverse applications. For example, the closed feedback control loop was applied in the database system to realize self-tuning memory management with dynamic resource allocations [114]. Another application uses a model-based control-theoretic solution combined with the MAPE-K control loop for the resilience management of cloud computing resource services [115][116]. Unlike other use cases of control theory in MAPE-K, some researchers used the MAPE-K control loop to optimize the classical controller based on fuzzy logic. Additionally, they integrated the property of online learning to modify the fuzzy rules at runtime, which is also evaluated in an application of cloud architecture adaptation [116].

The research works mentioned above have shown significant benefits by combining software engineering approaches, especially for the self-adaptive systems (e.g., by

[9] Quality of service.

[10] Service level objectives.

following MAPE-K for autonomic computing with control theory). Most of the related works mentioned focus on the benefits of a self-adaptive system by utilizing the principle of control theory. A paper focused on applying MAPE-K to improve the classical feedback controller but only considered the cloud architecture adaptation without strict timing requirements and constraints [116]. As is well known, this is not the same use case as for artificial cognitive control systems like future ACC, which are required to interact with the physical world in real-time. In this case, it is important not only to consider the self-adaptation capability of the system but also concurrently to consider the time-computation-dependency due to the requirement of real-time interaction with the physical world. Such a requirement constitutes challenges for the design of system architecture, especially regarding the deployment of time-consuming computing processes (e.g., knowledge acquisition and adaptation).

This dissertation aims to find a holistic concept for the architecture design of artificial cognitive control systems to address the aforementioned requirements, especially considering the architecture challenges introduced in Section 3.7. For this purpose, fundamental disciplines for designing software architecture, including relevant components and their relationships and interfaces, will be defined and summarized together as a concept of a generic architecture style for automatic control systems. The next section introduces these fundamental ideas about this generic architecture style.

4.2 Fundamental Design of Generic Architecture Style

Section 4.1 presents a brief introduction to related works on conceptual integration of control systems in control theory and self-adaptive systems in software engineering. As introduced, the design of the control system in control theory and the design of the self-adaptive system in software engineering benefit from each other mutually, relying on the design disciplines and concepts from both fields.

At the end of Section 4.1, it is noted that this dissertation aims to find a holistic concept for architecture design of sophisticated control systems, especially for the artificial cognitive control that is defined in this dissertation as a next-generation automatic control concept. For this purpose, this dissertation investigates the architecture design of control systems relying on software engineering approaches. Against such a

background, a generic architecture style with logical components from the viewpoint of software engineering is proposed in this dissertation, which can be applied as a template to design the architecture of different control systems. Subsequently, the architecture pattern is used to instantiate architectures of the introduced current control systems, aiming to deeply understand and validate the control systems from another perspective, thus contributing to the development of the mentioned networked architecture of artificial cognitive control (cf. Section 3.7–3.8).

4.2.1 Preliminaries of the Design of the Generic Architecture Style

Current control systems are designed as pure embedded systems to guarantee real-time interaction with the physical world. In such systems, each component is designed to work as a single independent and active process with a particular cycle time [48]. However, such a design concept strongly challenges artificial cognitive control, which includes multiple world models distributed in different nodes of the networked architecture (cf. Section 3.7.3). As a potential instance derived from the networked architecture, each node, including a single world model, can be instantiated as a hierarchical deliberative upper layer, combined with the lowest layer with the real-time control loop (also as a node), including the controller, to form a system architecture instantiated with multiple hierarchies. As discussed earlier, knowledge acquisition and adaptation properties require a nondeterministic processing time. Thus, there is a high risk of violating the required real-time capability of the embedded control system based on deterministic scheduling. From another perspective, the complexity of distributed world models is increasing due to increasing sensor access. The accompanying increasing data volume to be processed therefore further increases the risk of timing-constraint violation, particularly in the case of upper layers due to their larger observation scope for the physical system.

Against such a background, the deliberative part of the hierarchical architecture must be designed with an event-triggered computation mechanism like an information system instead of the previous design as a pure time-triggered embedded system relying on its strong information processing capability and mechanism without timing constraint. Thus, knowledge acquisition and adaptation with a long processing time would become

uncritical for the basic control task of the control system since the reactive part, including the feedback control loop with the controller, stays independently on the lowest layer.

In this case, the hierarchical architecture leads to a hybrid system design consisting of an information system and an embedded system. This trend makes the whole automatic control system evolve as a cyber-physical system, including heterogeneous computation mechanisms according to the formal definition in previous related work of Rehfeldt [20]. For this reason, this dissertation puts the focus on the architecture pattern, which is not limited to a self-adaptive system designed as a pure information system, but also considers hybrid system construction with heterogeneous computation mechanisms. The focused systems with such properties are called self-adaptive cyber-physical systems in this dissertation.

In Chapter 3, control systems within different categories were introduced. The MAPE-K for designing the self-adaptive system with the property of autonomic computing was used as a reference architecture to roughly identify corresponding functional similarities between components in the architectures from the viewpoint of control theory and software engineering, respectively. Thus, a basic understanding of different control systems from a software engineering perspective was derived.

The MAPE-K architecture is a well-known reference architecture for the self-adaptive system, particularly with consideration for the property of autonomic computing, and it has already been utilized in different applications (cf. Section 2.2.7). However, in this dissertation, it is not directly taken as the architecture solution for several reasons.

The first reason lies with the fundamental idea of MAPE-K, which is designed for self-adaptive IT systems like databases and servers. As noted earlier, such information systems focus on the sequence of events instead of physical execution time. That means there is no guarantee in this case that all activities included in the functions of MAPE-K like monitor, analyze, plan, and execution can be completed under deterministic time constraints. However, this is very important for control systems designed as cyber-physical systems requiring real-time interaction with the physical world. For example, in the case of ACC, the vehicle’s longitudinal control could even completely fail due to the violation of the timing constraints in the worst case, which is definitively not permitted to happen in reality. Thus, it can be said that the MAPE-K works

well for the “cyber” part; however, unfortunately, it is not exactly appropriate for the “physical” part of automatic control, particularly in artificial cognitive control.

Another reason relates to the differences about sensors and actuators (also called effectors in MAPE-K). The sensor and actuator in MAPE-K are deployed as interfaces in a software component touchpoint, which exposes the state and management operations to a resource in the system [46]. The sensor and actuator, categorized as manageability interfaces in MAPE-K, deliver the autonomic manager a standard interface rather than the diverse interface mechanisms associated with various managed resource types. In this case, the sensor and actuator are responsible for requesting and sending messages without additional functions (details cf. Section 2.2.7).

However, in the case of an automatic control system, the sensor and actuator are not only interfaces for information exchange between the technical and physical systems. Additionally, they also include hardware mechanisms and corresponding basis software components to manipulate the hardware infrastructure, which thus enables the sensor and actuator to influence the controlled plant and the environment directly. For example, in some advanced ACCs, LiDAR is utilized instead of the radar sensor to track multiple preceding obstacles [86]. In such ACCs, the software component for control of LiDAR also relies on the sophisticated numeric-algorithmic approach, possibly including a configuration with application parameters (e.g., the rotation speed of the LiDAR). The configuration may need to situationally adapt to guarantee the lowest acceptable sensing performance if the original sensing reliability is limited due to the environment, such as under extreme weather conditions like fog or heavy rain.

Another significant difference is that the sensor and actuator or effector in the MAPE-K model are linked together. Such a concept guarantees that a configuration change caused by the effector can be reflected as a notification through the sensor interface. However, this reflection is not included in the case of the automatic control system since the sensor and actuator are independent components linked via the controller and the measurement unit (cf. Figure 3.1). Due to these differences, the reference model of MAPE-K cannot be directly taken as an architecture solution for the design of future control systems.

The final reason lies with the architecture concept regarding system decomposition. In the MAPE-K, autonomic managers can also constitute a layered architecture, which includes a group of several touchpoint autonomic managers (on the lower layer) and an orchestrating autonomic manager (on the higher layer). Each touchpoint autonomic manager only concerns its managed resources, which may be only a part of the whole system. Instead, the orchestrating autonomic manager is deployed to coordinate the touchpoint autonomic managers to guarantee system-wide autonomic computing behavior (e.g., in the case of workload management) [46]. Thus, the task of self-management is horizontally decomposed into concurrent subtasks, which are respectively taken over by each touchpoint autonomic manager. However, in the case of an automatic control system, the hybrid hierarchical architecture still requires a hierarchical decomposition of the control task on different levels of abstraction, in addition to the horizontal decomposition. Such decomposition has not yet been exactly specified in the MAPE-K reference architecture.

For these reasons, the MAPE-K reference model is not directly taken as a solution for designing the architecture of automatic control. Instead, a completely new architecture concept more appropriate for the design of sophisticated architectures for future control systems is proposed in this dissertation. Details of this new architecture concept will be introduced in the following sections.

4.2.2 Fundamental Component Structure within Generic Architecture Style

Since the MAPE-K reference architecture is not exactly appropriate for the design of automatic control, another architecture pattern consisting of a fundamental component structure of “SICAP-K” is proposed in this dissertation, as shown in Figure 4.1. The fundamental component structure includes a technical system and a physical system, respectively marked in yellow and green, which has the same system construction as the architecture of current control systems applied in the ACCs, as illustrated in Chapter 3.

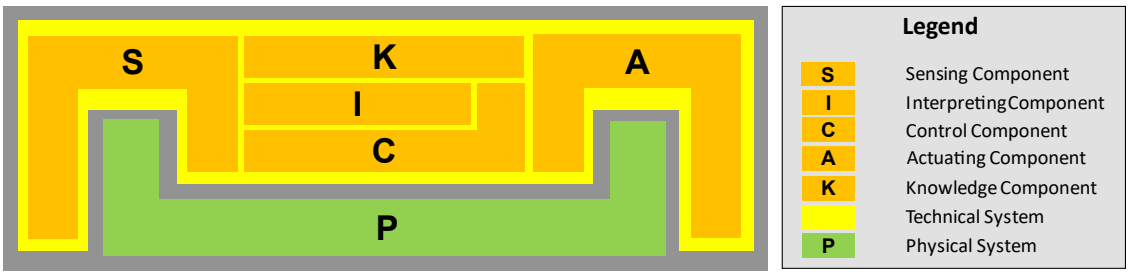


Figure 4.1: Fundamental Component Structure in Generic Architecture Style

The physical system refers to a summary of the user, controlled plant, and surrounding environment. From the viewpoint of software engineering, the technical system architecture here looks slightly different from the technical systems in the architectures of current control systems based on the block (plugging) diagram from the control theory viewpoint. The architectures in the control theory focus strongly on the technical process's control flow with corresponding physical variables on a very concrete level. The architecture derived from the software engineering viewpoint focuses instead on static system construction, with components, their relationships, and logical functionalities on a meta-level. Thus, such architecture is also called logical or functional architecture in software engineering. Since the control system needs to interact with the physical world to complete its control task, hardware sensors and actuators, including pieces of basis driver software deployed on them, work as interfaces between the technical system and the physical system. Since the sensors and actuators are not a part of the software system to be designed, they are hidden in the logical architecture of Figure 4.1.

In the “SICAP-K” structure, the component “P” refers to the *Physical System*, including the user, the controlled plant, and the environment. The component “S” refers to a *Sensing Component* responsible for the data preprocessing (e.g., collecting, aggregating, and filtering the raw data collected by the hardware sensors from the physical system). The output of the sensing component is the structured data, which is called a “symptom” in this dissertation, similar to the case of monitor function in the reference architecture of MAPE-K. The symptom is forwarded by the sensing component into three other components, “C”, “I”, and “K”, respectively referring to *Control Component*, an *Interpreting Component*, and a *Knowledge Component*.

Based on the symptom, which describes the current context of the physical system on a concrete level of sensory data, the control component “C” determines a control strategy. The determined strategy can be transient like the case of a conventional feedback controller, similar to a reflect system without consideration of future time horizons. Alternatively, the determined control strategy can also be predictive (considering a future time horizon), in which case the interpreting component “I” is required. Subsequently, the control strategy is forwarded to an *Actuating Component* “A”, which decomposes the strategy into individual activities to manipulate the actuators.

Unlike the sensing component describing the physical system's current context on a concrete data level, the interpreting component identifies the current control problem using an inference engine. It evaluates the current context on a higher level of abstraction by checking correlations of observed facts included in the symptom and the constraints and properties predefined in the problem catalog. Optionally, the observed facts included in the symptom are also used to predict future context, and thus the future context can also be evaluated. Once the problem is identified, the interpreting component sends a change of request to the control component. Thus, the control component will update the control strategy for realizing adaptation. If the problem cannot be identified, the interpreting component acquires the knowledge support (e.g., from a knowledge component “K”).

The knowledge component “K” is responsible for the storage of the control system’s domain knowledge. Additionally, knowledge exchange with the outside world, like acquiring and sharing to and from external knowledge sources, also happens inside the knowledge component. The processing in other “SICA” components relies on the domain knowledge included in the knowledge component. The knowledge component initializes the domain knowledge for the other components in the first system operation. The other components also adapt the knowledge included in the knowledge component from the opposite side. More detailed processes of knowledge-relevant system behaviors will be explained in Section 4.2.5.

Generally, it can be understood that a fundamental feedback control loop based on the component cycle of “SCAP” is constituted. The interpreting component “I” is also involved in the fundamental structure, playing the role of an optional component

participating in the feedback control loop. Thus, the control system can understand and analyze the context information and knowledge on a higher level of abstraction. With the support of the knowledge component “K”, a complete fundamental structure of “SICAP-K” is constituted.

4.2.3 Structural Adaptation Composition in Generic Architecture Style

After introducing the fundamental component structure, which is seen as the fundamental building block for representing a closed control loop, the next question is how to organize different component structures to construct a more sophisticated system architecture, including multiple control loops.

In Section 3.7, it was discussed that the current architecture design of sophisticated control systems proposed by the intelligent control field follows the idea of hierarchical decomposition of the control task into different subtasks, vertically on different levels of abstraction or horizontally on different levels of competence. Different limitations of such architecture design approaches have also been discussed. For example, the system design aims to adapt the control activities on lower levels to complete the original control task on the highest level, which refers to the technical process control in this dissertation. However, configuration control [95] (cf. Section 3.6.2) is neglected, which refers to the adaptation of component configuration (e.g., the controller parametrization or the domain knowledge in the knowledge component).

This dissertation has introduced preliminary ideas of architecture design based on the proposed multidimensional networked topology to overcome the limitations mentioned above, as illustrated in Figure 3.11. As discussed earlier, artificial cognitive control as the future control system in the next generation is continually evolving into a self-adaptive cyber-physical system with the property of so-called autonomic computing. Autonomic computing typically refers to the well-known self-X properties covering self-configuration, self-optimization, self-healing, and self-protection [46]. Thus, following these preliminary ideas, the configuration control and the component adaptation are taken as the focus in developing the generic architecture style in this dissertation.

The fundamental component structure includes a basic feedback control loop, represented by the “SCAP” component cycle. Optionally, the interpreting component “I”

and knowledge component “K” can also participate in the loop. In this case, the primary feedback control loop, including the controller based on linear robust control theory and previously deployed on the lowest layer in the hierarchical architecture design by the field of intelligent control, can be represented by a proposed fundamental component structure of “SICAP-K” and deployed as a node in the networked system architecture.

Once a controller with static parametrization becomes insufficient, due to variations in the behavior of the physical system, the adaptive control technique is required. Adaptive control technologies allow, for example, controller parametrization to be adapted by the adaptation unit, as discussed in Chapter 3. In this case, another adaptation unit as a complete autonomic manager can be deployed on a secondary control loop. It takes the primary feedback control loop, including the controller based on robust control theory as its managed system. The deployment of the secondary control loop means that another fundamental component structure of “SICAP-K” is deployed in the networked architecture, as introduced in Figure 3.11 of Section 3.7.3.

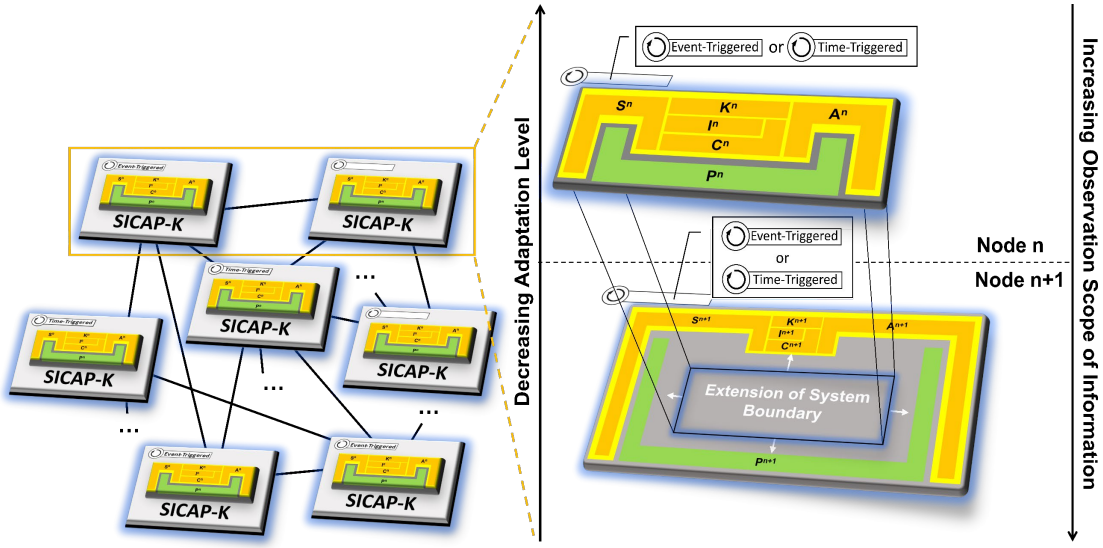


Figure 4.2: Structural Paradigm of Adaptation Composition for Networked Architecture of Future Control System

In this case, these two “SICAP-K” component structures are linked. Each “SICAP-K” structure represents a node, considered the fundamental computing element. Thus, a networked system architecture can be constituted once multiple “SICAP-K” component

structures are linked mutually in any arbitrary form, as shown on the left side of Figure 4.2. As mentioned earlier, the secondary feedback control loop works as the adaptation manager and takes the primary feedback control loop as its managed system. Such a management relationship can also be transferred to the other nodes in the networked architecture.

On the right side of Figure 4.2, an example of two arbitrary nodes from the networked system architecture is illustrated. The superscripts n and n+1 represent the index of different nodes. As shown in Figure 4.2, node n+1 has a larger observation scope of information about the physical system than node n. Thus, the adaptation level decreases from node n+1 to node n, which means that node n+1 is a so-called “adaptation manager” of node n. Each node can be designed with an appropriate triggering mechanism (event-triggered or time-triggered, cf. Section 4.2.4). If necessary, these two nodes can also be instantiated as two layers to constitute a two-layered architecture. Thus, this dissertation interprets such layered architecture as an instance derived from the networked system architecture.

As discussed in Section 3.7, the hybrid hierarchical architecture in intelligent control follows the closed-world assumption. This means that it is assumed that the system has all the needed domain knowledge about the world (on different levels of abstraction) to accomplish the determined control tasks. Unfortunately, such an assumption is utopian for the case of future control systems like artificial cognitive control due to the world’s nondeterministic aspects, which has been demonstrated by diverse negative examples, particularly in the field of autonomous driving safety [117]–[119]. Some other critical scenarios are also discussed in Section 3.6. Instead of the closed-world assumption, this dissertation takes an open-world assumption during the architecture design and aims to avoid the previously mentioned critical scenarios. The open-world assumption holds that the control system does not have all the required domain knowledge about the world. Additionally, it allows self-adaptation activities like knowledge acquisition and adaptation, relying on the knowledge component and interpreting component in the proposed fundamental component structure shown in Figure 4.1.

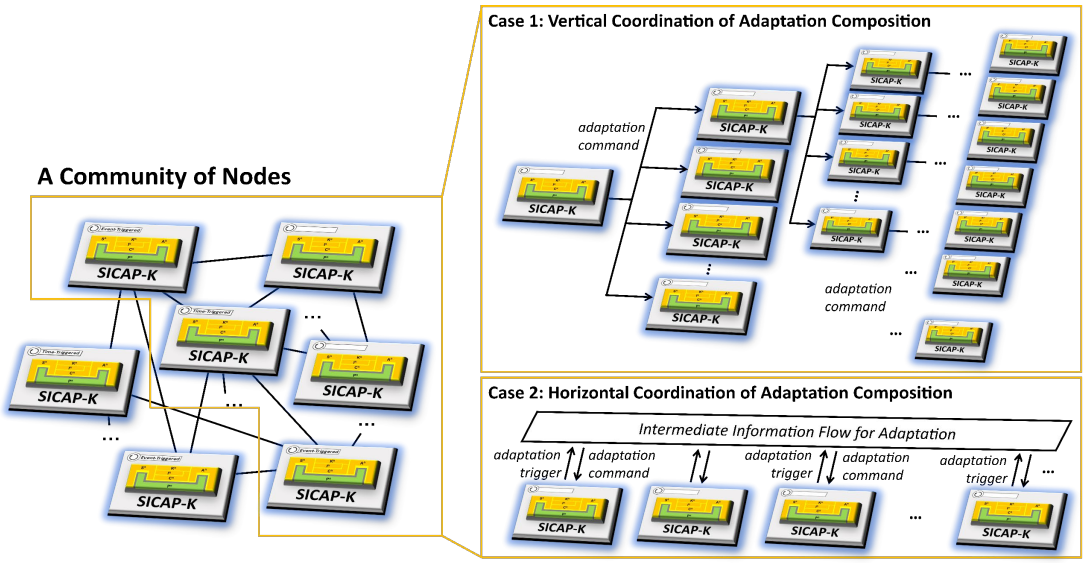


Figure 4.3: Coordination of Nodes' Adaptation Composition in Networked System Architecture

Following the principle of open-world assumption, a compositional system design approach is selected instead of the decomposition approach, which means that the system construction begins from one node to multiple nodes. As presented earlier, each node represents a closed control loop with the "SICAP-K" structure. In the case of extensive sensor access, the processing tasks of the sensory data context analyzing and inferencing become more time-consuming and thus hard to complete within a limited cycle time. Against such a background, it is utopic to expect that only a single adaptation unit with the "SICAP-K" structure can handle all collected sensor data. Thus, more adaptation units must be deployed on additional control loops, which leads to an architecture consisting of multiple nodes connected as a network (cf. left side in Figure 4.3). With this idea in mind, an architecture paradigm of so-called adaptation composition is proposed in this dissertation, which can be applied to interconnect increasing nodes to construct a more sophisticated system architecture continuously. The right side of Figure 4.3 illustrates different cases of coordination of the nodes' adaptation composition in the networked system architecture.

4.2.3.1 Vertical Coordination of Adaptation Composition

If necessary, the networked nodes can be grouped as different communities in the networked architecture. Unlike the nodes as the fundamental building blocks in the

system architecture, this community can be understood as an overarching unit like a software module consisting of single or multiple building blocks. Some nodes are so-called “adaptation managers” and take other corresponding nodes as their managed systems. Thus, the nodes can be classified as “manager nodes” and “managed nodes.”

In most cases, the manager nodes have larger scopes of information observation about the physical system than managed nodes. Thus, adaptation relationships between the nodes can be organized hierarchically with multiple layers (cf. Case 1 in Figure 4.3). A node on a higher layer coordinates the adaptation of multiple nodes on a lower layer. Nodes on the same layer can communicate interactively but do not have any adaptation relationship.

Once a system architecture includes multiple nodes, a significant point for the architecture design is distributing the effort of acquiring and processing sensory data and knowledge basis to different nodes and knowledge components between them. For example, several nodes are instantiated as layers within a hierarchical architecture. Different adaptation units are deployed on different layers, along with increasing observation scopes for the physical system from lower to upper layers, to guarantee the self-adaptability of the system, as is roughly described in the case study of architectural evolution in Chapter 3.

In this case, the adaptation units on lower layers need to handle fewer sensory data, but they must do so with high precision to frequently determine precise and short-term adaptation strategies on a concrete level. The adaptation units on higher layers must handle much more sensory data. Nevertheless, since higher layers do not require such high time resolutions of adaptation activity as lower layers, they only need to focus on determining long-term adaptation strategies on higher abstraction levels with less precision. Thus, considering more context about the physical system becomes reliable for the adaptation units on higher layers.

Along with this idea, the deployment of adaptation units can be extended to arbitrary nodes. Thus, a networked architecture, including multiple nodes with different adaptation levels, is constituted, as shown in Figure 4.2. Each adaptation unit in a certain node is responsible for adapting its neighbor nodes on a lower adaptation level via

corresponding knowledge components. Thus, all adaptation units in different nodes with nested observation scopes work together to determine adaptation strategies on different levels.

With the help of the parallel computations of multiple adaptation units, processing efforts for all collected sensory data can be distributed into different nodes to guarantee timing constraints, such as a limited cycle time, are met. Additionally, a world model with incredibly high complexity is avoided, and instead, distributed world models are deployed in corresponding knowledge components between the nodes. Thus, based on the paradigm of adaptation composition, the self-adaptability of future control systems is particularly considered, which is beneficial for the architecture design of artificial cognitive control with the property of knowledge acquisition and adaptation.

4.2.3.2 Horizontal Coordination of Adaptation Composition

In the last section, the vertical coordination of adaptation composition was introduced. In vertical coordination, the control system is decomposed into different nodes with different levels of abstraction. In such a case, the nodes following the adaptation chain from higher to lower levels of abstraction have strong dependencies mutually. They can be seen as several building blocks in a software module for realizing a certain function, defined as a functionally closed unit in the overall system [15].

For example, in the case of the multi-layered planning function of an autonomous robot, the planning on the highest level of abstraction may focus on the mission from a point A to another point B without focusing on concrete behaviors for realizing the robot’s movement from A to B. However, like the set profile of travel time, its planned mission is taken as a specification during the behavioral planning at a lower level of abstraction. In this case, both nodes for the planning at different levels simultaneously complete the autonomous robot's planning task.

Nevertheless, nodes with adaptation relationships may not be considered within a software module with strong dependencies in some other cases. Instead, their adaptations may be across different software modules with loosely coupled functionalities. Since such decomposition follows in an orthogonal direction compared to vertical coordination, the coordination between the nodes in such a case is called

horizontal coordination in this dissertation, as illustrated in Case 2 (cf. Figure 4.3), for example, in a case where there are several driving modes like “Eco,” “Comfort,” and “Sport” available in a vehicle. The driving mode’s change by the ACC may cause the adaptation of other loose-coupled functions’ configurations such as the vehicle’s steering and gearbox.

In such a case, the community of nodes may include building blocks in multiple software modules from the viewpoint of software engineering. The manager nodes may publish their adaptation requests by sending adaptation triggers as commands via an intermediate information flow. Thus, corresponding managed nodes can subscribe to the adaptation commands and subsequently realize their adaptations. Compared to vertical coordination, no node has a global view of the whole community in the case of horizontal coordination.

4.2.4 Applying Triggering Mechanisms for Nodes with Fundamental Component Structure

Following the paradigm of structural adaptation composition mentioned in the previous section, multiple fundamental component structures can be organically constructed to build a sophisticated system with a unified networked architecture. Each fundamental component structure is interpreted as a node within the architecture.

As introduced in Section 3.7.1, the current design approach of hierarchical control system architectures by the intelligent control field focuses on a system construction with several hierarchical layers. Each layer can be implemented as a subsystem with an appropriate triggering mechanism (either time- or event-triggered), and thus makes the overall system become a hybrid system, including both triggering mechanisms.

This dissertation supports these triggering mechanisms in its proposed generic networked architecture style. Different nodes with their fundamental component structures in networked system architecture can be designed as time-triggered or event-triggered by applying the proposed generic architecture style. Each node is permitted to have its unique triggering mechanism, and multiple nodes can be deployed within a single subsystem, which thus leads to a hybrid subsystem with different computation mechanisms.

Depending on concrete applications, the time-triggered mechanism can be applied for the system parts with critical time-relevant requirements to realize the real-time interaction with the physical world. From another perspective, the event-triggered mechanism is also appropriate for system parts without any time-dependent requirement (e.g., the inferencing process for state identification of the physical system, which is deployed in the control loop without direct connection with the physical world). Since the knowledge components play the roles of interfaces to isolate the nodes with different computation mechanisms, loose coupling is realized, and the computation mechanisms will not negatively influence each other. During system operation, the knowledge components are responsible for fulfilling timing constraints in the case of communication across nodes. Different timing requirements at the system level or corresponding lower levels (e.g., at the node level) can also be seen as a kind of domain knowledge saved in the knowledge components.

4.2.5 Applying Communication Architecture Patterns for the Design of Dynamic System Behaviors

In addition to the static system construction, which was introduced in the previous section from the perspective of structural adaptation composition, another perspective of the generic architecture style focuses on dynamic system behaviors, which more concretely refers to the component interactions via appropriate interfaces within the system architecture. Figure 4.4 illustrates an overview of communication paths between the components via different interfaces, which provides a framework for designing component interactions.

In this dissertation, the component interactions are not limited by using a certain specified communication paradigm to retain the generalization capability of the generic architecture style. Instead, several standard communication architecture patterns are taken as candidate solutions for specifying the roles of components in the communication of defined use cases. Theoretical fundamentals of these communication architecture patterns were introduced in Section 2.3, including the request-response pattern, the publish-subscribe pattern, the pipes-and-filters pattern, the shared-repository pattern, and the blackboard pattern.

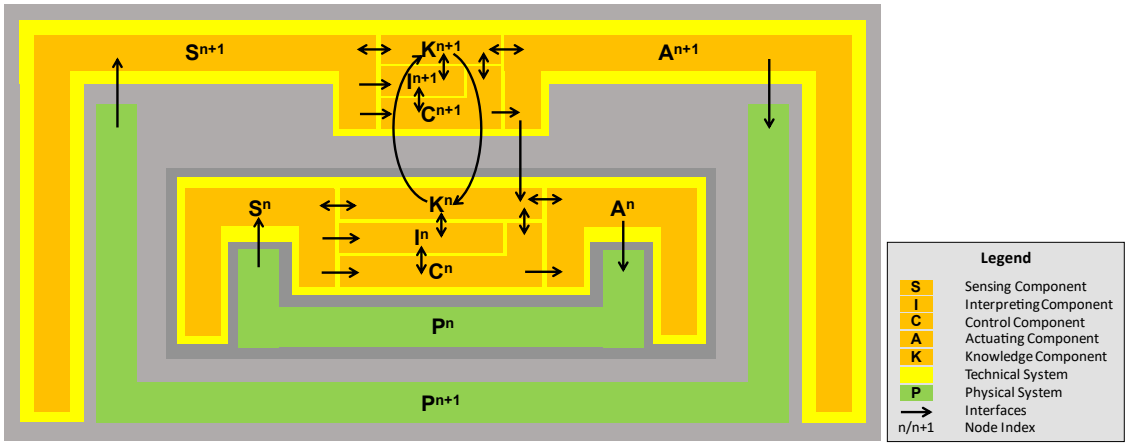


Figure 4.4: Component Interfaces within the Generic Architecture Style

Through selection of appropriate communication paradigms from the candidates of standard communication architecture patterns, the component interactions for a concrete instantiated system architecture can be reliably implemented, depending on the concrete boundary conditions of required system communication. In this dissertation, several use cases (UCs) are defined to understand these interactions via the interfaces better. More details of these UCs will be introduced in the following section in detail. Additionally, some of these standard communication architecture patterns will be taken to apply within a concrete architecture of an artificial cognitive cruise control system, which construction is also instantiated from the generic architecture style and taken as an evaluation example in this dissertation.

4.2.6 Dynamic System Behaviors as Use Cases in Generic Architecture Style

In addition to the static system construction, another view of architecture design lies with the dynamic system behavior, which refers to the interaction of components in the architecture pattern. In this dissertation, four uses cases (UCs) are proposed. To symmetrically describe the component interactions: technical process control in a single node (UC1); knowledge initialization, retrieval, and update in a single node (UC2); adaptation control across multiple nodes (UC3); and knowledge sharing across multiple nodes (UC4). These use cases will be introduced in detail in the following sections.

4.2.6.1 UC1: Technical Process Control in a Single Node

The first use case refers to technical process control relying on a closed control loop in each node of the networked architecture. In Section 4.2.2, the basic closed control loop consisting of “SCAP” components was presented. The interpreting component “I” as an additional component can also participate in the control loop if necessary and thus constitutes a closed loop of “SICAP”.

To provide a better understanding of the process flow, detailed processes within the use case are listed as follows[11]:

1. The sensing component “S*” retrieves the data collected by hardware sensors from the observed physical system. The sensing component further processes all the retrieved sensory data such as aggregation, correlation, or filtering, based on the respective desired requirements. Thus, processed data with an expected structure can constitute a symptom that is a predefined standard description of the current context of the physical system based on the observed facts represented by the sensory data, including the structured sensory data themselves.
2. Optional: Based on the symptom provided by the sensing component “S*,” the interpreting component “I*” evaluates the current of the physical system context by searching for the relevance with its available symptoms as domain knowledge. Once the relevance of one of the available symptoms can be found, the interpreting component relies on an inference engine to identify the current problem. If necessary, the predicted future context of the physical system relying on prediction algorithms can also be considered here. Finally, the interpreting component can determine a change of request based on its policy knowledge and forward it to the control component “C*”. It is emphasized that the interpreting component is an additional component in the use case.
3. The control component “C*” takes the change of request and the symptom from the interpreting and sensing components respectively as inputs to determine an

[11] The superscript “*” is used here to describe the index of arbitrary nodes in the networked architecture of the control system.

adaptation strategy, relying on its domain knowledge included in a decision tree. If the interpreting component is not deployed, the control component reflects on the delivered symptom directly without considering the change of request.

4. The adaptation strategy determined by the control component is forwarded to the actuating component, which splits up the strategy as individual adaptation activity. The actuating component forwards the individual adaptation activity further to the hardware actuators. Thus, the actuators can execute the activity to interact with the physical system.

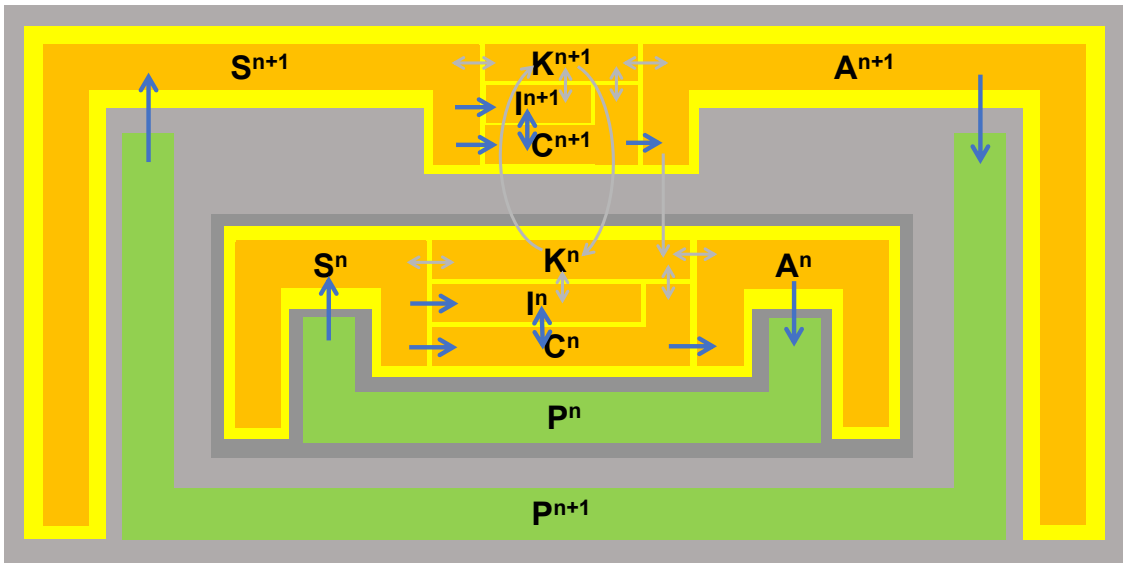


Figure 4.5: Technical Process Control in a Single Node (UC1)

Such a control loop exists in each node of the networked architecture, as shown in Figure 4.5. The proposed generic architecture style mainly focuses on the self-adaptation of the control system across different nodes. However, it also supports concurrent interactions of different nodes directly with the physical system. This is appropriate, for example, for the case of a control system with multiple actuators independently influencing the physical system within different observation scopes.

4.2.6.2 UC2: Knowledge Initialization, Retrieval, and Update in a Single Node

The second use case focuses especially on knowledge flow within the architecture pattern, including knowledge initialization, retrieval, and update, as shown in Figure 4.6. As introduced in the previous use case, the “SICAP” components constitute a closed

control loop to interact with the physical system with different observation scopes within corresponding nodes. To guarantee the basic functionality of the control loop, the components “SICA” must complete the tasks introduced in Section 4.2.6.1, for which they still need domain knowledge provided by the knowledge component “K*”[12].

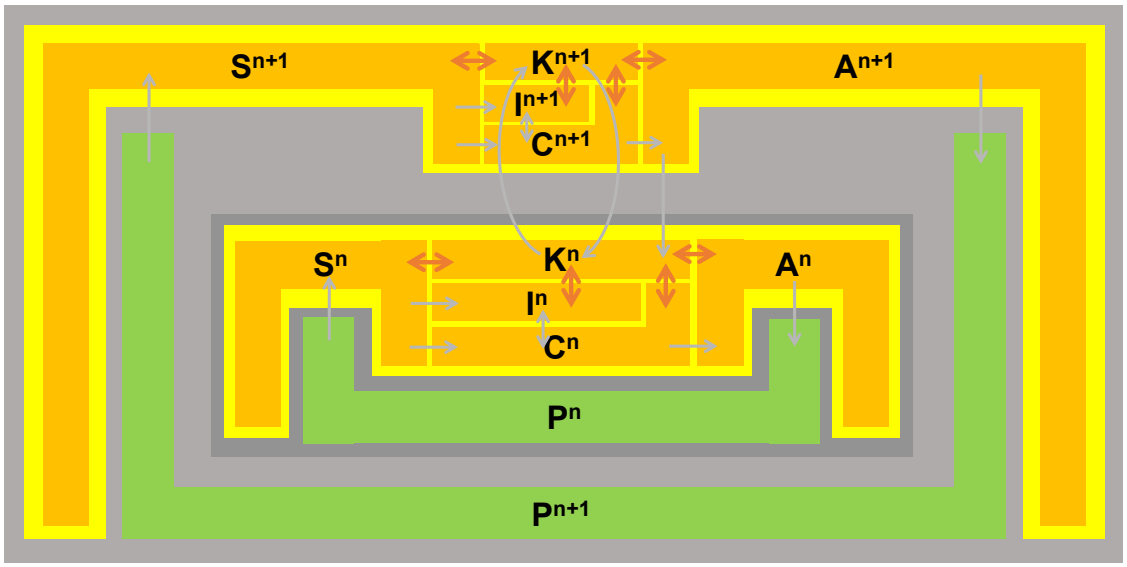


Figure 4.6: Knowledge Initialization, Retrieval, and Adaptation (UC2)

For example, the sensing component “S*” collects and processes sensory data to generate the symptom. In this case, it needs the knowledge about the symptom’s configuration, which is called sensing knowledge in this dissertation, including specifications like the naming format of the symptom, required resolutions of sensory data values, and configurations of data aggregation and filtering. Thus, the symptom can be described systematically, which the inference engine can process in the interpreting component.

Similarly, the interpreting component “I*” also relies on the so-called interpreting knowledge provided by the knowledge component. As presented earlier, the interpretation component's functionality can be briefly categorized as several processes: (1) prediction of future context of the physical system based on the current context, (2) problem identification based on evaluation of both contexts by searching relevance to

[12] The superscript “*” is used here to describe the index of arbitrary node in the networked architecture of the control system.

known symptoms and facts, and (3) reasoning for determination of the change of request. In these processes, the approach for future context prediction, like a data-driven machine learning algorithm, relies on configuration of relevant application parameters. The problem identification needs the domain knowledge about known symptoms and facts of the physical system, which are saved in a systematic format. Additionally, the reasoning process also relies on policy knowledge to determine whether the change of request is necessary.

The same case also exists for the control and the actuating component. The control component must know what kind of control decision is appropriate for which symptom and what kind of consequence the control decision has in order to determine the control strategy. For this purpose, it requires knowledge about the symptom, including the monitored sensory data and the decision tree, which is categorized as the control knowledge in this dissertation. Finally, the actuating component is required to split up the control strategy as a sequence of individual activities, relying on the knowledge about the construction of the control strategy. Additionally, knowledge about time-relevant constraints may also be an important part of knowledge in this case since some control systems also have critical requirements for interaction with physical systems in real time.

The knowledge component serves as a knowledge repository and provider to guarantee that all “SICA”-components can complete their tasks. After the system design, the knowledge component initializes the required domain knowledge for the other components, called knowledge initialization in this dissertation. During the computation processes in the components, they are also able to request support from the knowledge component for the domain knowledge delivery. For example, the interpreting component can request policy knowledge for problem identification if it cannot identify the current problem by itself. In this case, the knowledge component plays the role of a consultant.

From the opposite side, it is also possible that new domain knowledge is created during the control process. For example, the control component determines a control strategy with the expectation of certain control performance. However, due to the time-relevant behavioral change, the physical system has a different reaction than expected. The

control component can also create new domain knowledge and update the newly created knowledge into the knowledge component.

Similarly, the same knowledge update can also be completed by other components in the generic architecture style if necessary. Based on such a mechanism, the mentioned property of knowledge acquisition and adaptation in current control systems is realized in this dissertation by the proposed generic architecture style. Generally, the previously mentioned adaptation within the architecture pattern can be understood as a knowledge- or a data-driven process. Once a component (either the knowledge component or one of the other components) has a change of knowledge, it can disseminate the change to other components. Thus, knowledge within the knowledge component and other components can always be synchronized.

During the design of concrete application of control systems, detailed description formats of the domain knowledge must still be specified. This dissertation does not define any concrete description format, aiming to retain the generalization capability of the proposed generic architecture style. From another perspective, it is also an outlook for this dissertation, which can be further investigated in the future to find out which formats are appropriate for what kind of applications.

4.2.6.3 UC3: Adaptation Control across Multiple Nodes

The third use case in the generic architecture style focuses on the process flow of adaptation control across multiple nodes. In UC3, one node in the networked architecture adapts the managed subsystem deployed in other nodes, following the introduced paradigm of adaptation composition in Section 4.2.3 (cf. Figure 4.7). A single subsystem in a node can adapt to several subsystems in other nodes due to distributed scalability of the generic architecture style, as illustrated in Figure 4.7, focusing on the case with only one additional subsystem in a secondary node.

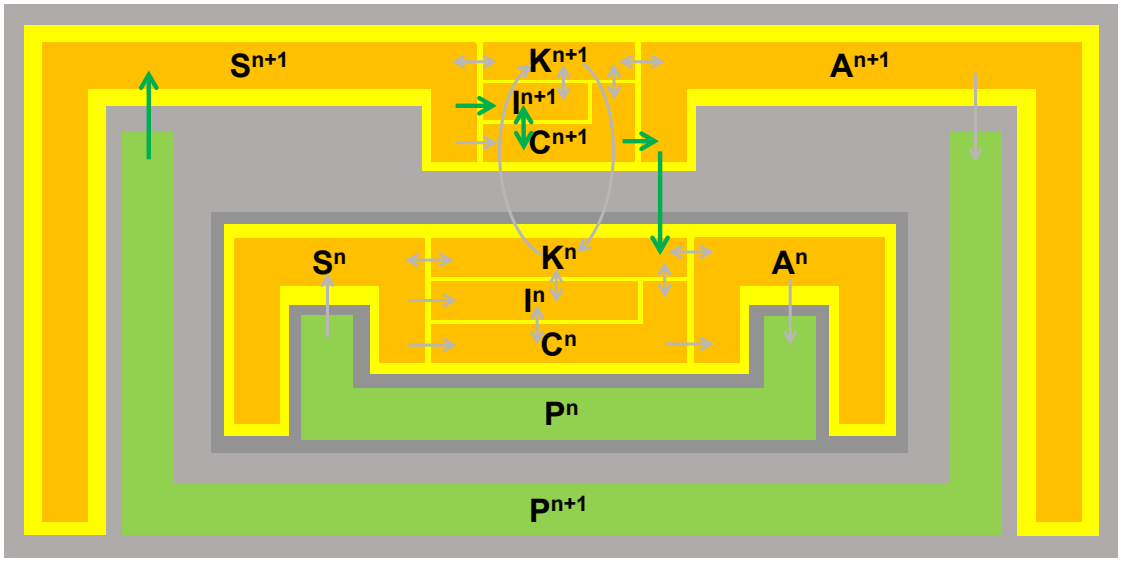


Figure 4.7: Adaptation Control across Multiple Nodes (UC3)

The process flow of adaptation control across multiple nodes is similar to the use case of technical process control in a single node, as introduced in Section 4.2.6.1. There are only two differences between the use cases. The first is that the interpreting component for high-level analysis and reasoning is involved regardless. Thus, the component in the corresponding node can determine the adaptation strategy on a higher level of abstraction. The second difference is that the actuating component does not forward the control activity to the actuators to interact with the physical system. Instead, it forwards individual adaptation activity split from the high-level adaptation strategy to the knowledge component in the node that includes the managed subsystem (so-called "managed" node). In this case, another mechanism to realize the knowledge acquisition and adaptation in the managed node is involved in the architecture pattern.

4.2.6.4 UC4: Knowledge Acquisition and Sharing across Multiple Nodes

In the previous use cases, the knowledge flow focuses on the interaction between the knowledge component and other functional components like "SICA" in the architecture pattern. The final use case, knowledge acquisition and sharing across multiple nodes, focuses on the knowledge flow between knowledge components deployed in different nodes of the networked system architecture, as shown in Figure 4.8.

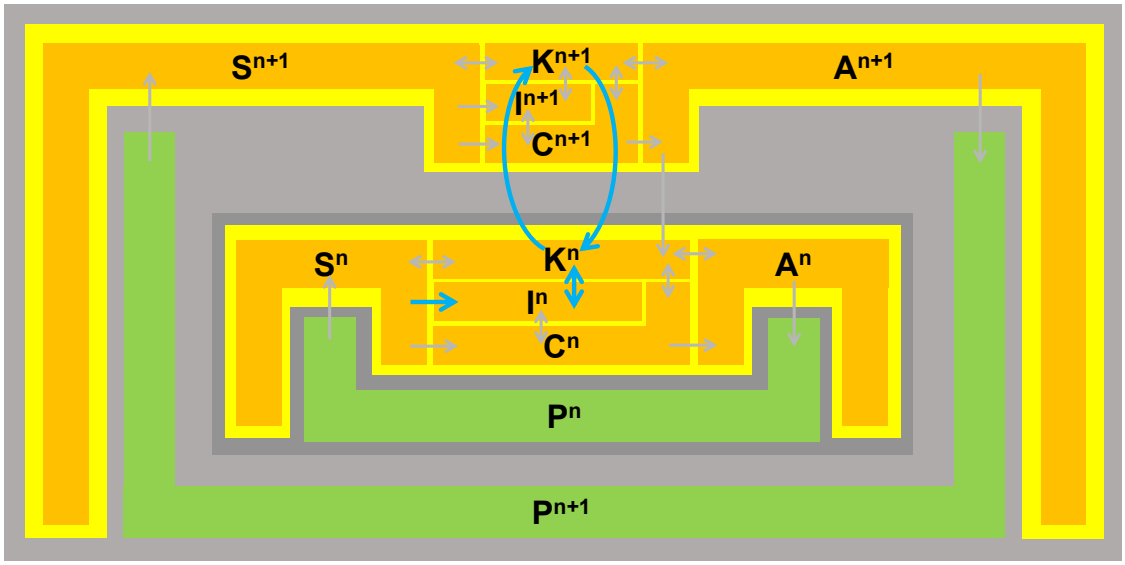


Figure 4.8: Knowledge Acquisition and Sharing across Multiple Nodes (UC4)

In this dissertation, knowledge acquisition and sharing happen when the interpreting component lacks domain knowledge. The other components like “SCA” do not have any requirement for knowledge acquisition and sharing since they rely on the knowledge to complete the tasks, in which a solution can be found in any case. For example, the sensing component follows certain configurations to generate symptoms. The control component relies on its knowledge to identify the best solution of control strategy within the search space of its decision tree.

The standard processes within this use case are summarized as follows:

1. The sensing component “S*” collects the sensory data from the physical system and processes the data to generate the symptom, which is then forwarded to the interpreting component “I*”.
2. The interpreting component “I*” evaluates the physical system's current (and future) context by trying to find relevance with one of its known symptoms (which is saved as domain knowledge) to identify the current problem. Due to the lack of knowledge, the problem cannot be reliably identified. Thus, the interpreting component cannot generate the change of request.
3. The interpreting component requests support from the knowledge component “K*”. Since the knowledge component “K*” has the same domain knowledge as

the interpreting component, it also cannot identify the problem. Thus, it forwards the request to other knowledge components, such as the knowledge component "K*+1" in other neighbor nodes.

4. The knowledge component "K*+1" shares its domain knowledge with the knowledge component "K*" in the neighbor node (e.g., its known symptom). The component "K*" forwards the received domain knowledge to the interpreting component "I*".
5. The interpreting component "I*" accesses the new domain knowledge and tries to determine whether the problem can be identified. If the problem can be identified, the interpreting component "I*" uses its training algorithm to integrate the new domain knowledge into the knowledge base included in the knowledge component "K*". If the problem cannot be identified, the interpreting component "I*" requests the knowledge component "K*" again. Processes 3 and 4 will be repeated with the interaction of other available knowledge components until the solution can be found.

4.3 Instantiation of Generic Architecture Style for Different Control Systems

After introducing the generic architecture style as the main contribution of this dissertation, from the perspectives of the static system construction and dynamic system behavior respectively, it is necessary to investigate the generalization potential of the proposed generic architecture style. For this purpose, the architecture pattern is further instantiated as logical architectures from software engineering for the cases of introduced control concepts with the architectures from the control theory viewpoint (cf. Chapter 3). Thus, the architectures from both viewpoints can be compared to examine whether the proposed generic architecture style is consistent on the architecture level with current control systems.

4.3.1 Basic Control following the Generic Architecture Style

The first instantiation focuses on basic control architecture from the view of control theory, which is shown again as the blue-grey architecture at the bottom of Figure 4.9. There is no adaptation unit as a secondary layer to build a hierarchical system

architecture in basic control. Thus, the architecture pattern is instantiated only with one layer illustrated at the top of Figure 4.9 as a yellow-green architecture from the view of software engineering. By comparing “blue-grey” and “yellow-green” architectures, relationships between the corresponding components included in the architectures can be mapped.

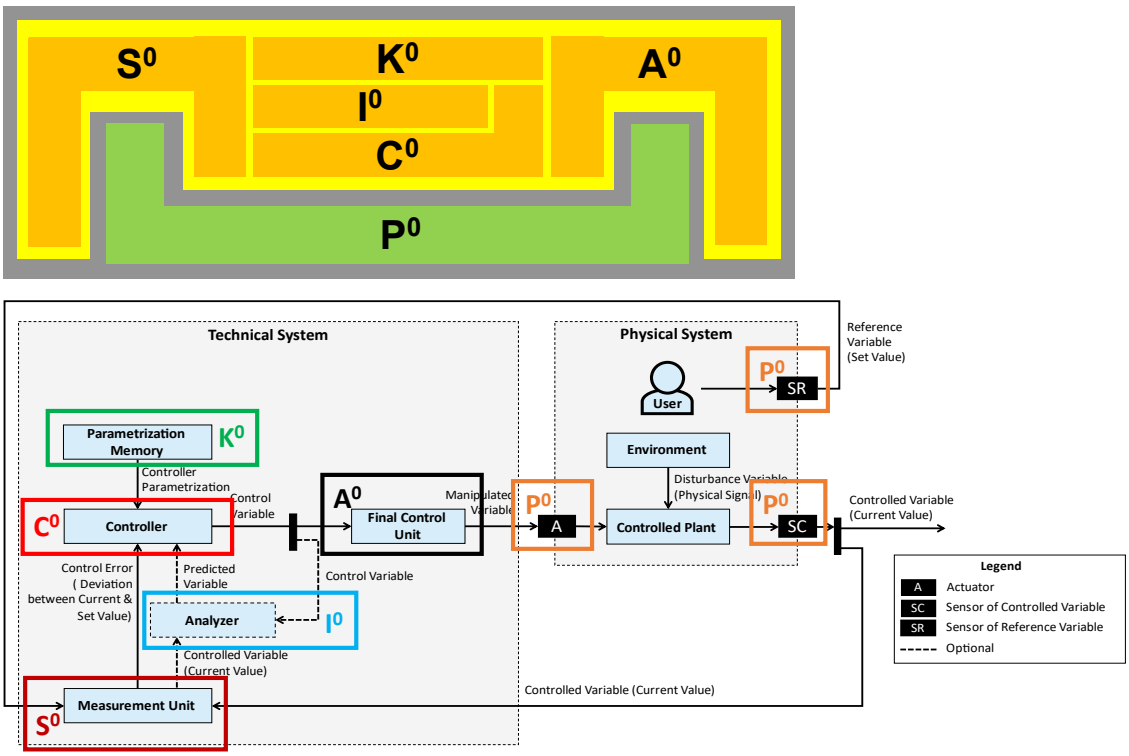


Figure 4.9: Instantiation of Architecture Pattern for Basic Control System

In the following sections, the word “blue-grey architecture” is used to represent the architecture from the control theory viewpoint to reduce description complexity. Similarly, the word “yellow-green architecture” is used to represent the instantiated logical architecture from the view of software engineering derived from the architecture pattern.

Firstly, the physical system and the hardware sensors and actuators (*SC*, *SR*, and *A*) as its provided interfaces for the interaction with the technical system can be mapped precisely as component “P^0” in the yellow-green architecture. The subscript “0” here stands for node zero, which can also be instantiated as a layer if necessary.

Since the measurement unit in the blue-grey architecture is responsible for collecting and preprocessing the sensory data, which is also covered by the sensing component in yellow-green architecture, another mapping relationship can be found. Additionally, it is very clear that the controller and the parametrization memory component respectively correspond to the control and knowledge components ("C^0" and "K^0") in the yellow-green architecture. Since the actuating component ("A^0") converts the determined control strategy into individual control activities, which are then forwarded into the physical system through actuators, it can be mapped by the final control unit in the blue-grey architecture. Finally, an additional component analyzer in the blue-grey architecture is used to predict the future context of the physical system. It can be interpreted as a simplified interpreting component ("I^0") since it also covers the functionality for evaluation of the future context of the physical system ("P^0"). Thus, it can be said that from the view of software engineering the instantiated architecture, which is derived from the generic architecture style, is consistent with the architecture of basic control from the viewpoint of control theory.

4.3.2 Naive Adaptive Control following the Generic Architecture Style

After the architecture comparison for basic control, this dissertation continues its investigation by focusing on naive adaptive control. The architecture of the naive adaptive control is illustrated as blue-grey architecture from the view of control theory again at the bottom of Figure 4.10. Since naive adaptive control includes an adaptation unit deployed on a secondary control loop, an architecture including two nodes is instantiated from the architecture pattern. Since these two nodes can then be instantiated as two hierarchical layers, as presented earlier, a two-layered hierarchical yellow-green architecture is thus constituted. Different layers are represented by superscripts "0" and "1".

In comparison to the blue-grey architecture for basic control, it can be seen that the adaptation unit deployed on the secondary control loop includes several components: a monitoring component, an adaptation controller, and an execution component. The monitoring component is responsible for data collection and preprocessing, which generates a symptom based on the standard data formats. The adaptation controller is

designed to determine an adaptation strategy, which is understood as high-level specifications for the adapted variable to adapt the saved values of application parameters in parametrization memory. The execution component takes the determined adaptation strategy as input to derive individual values of the adapted variable, which are forwarded to the parametrization memory.

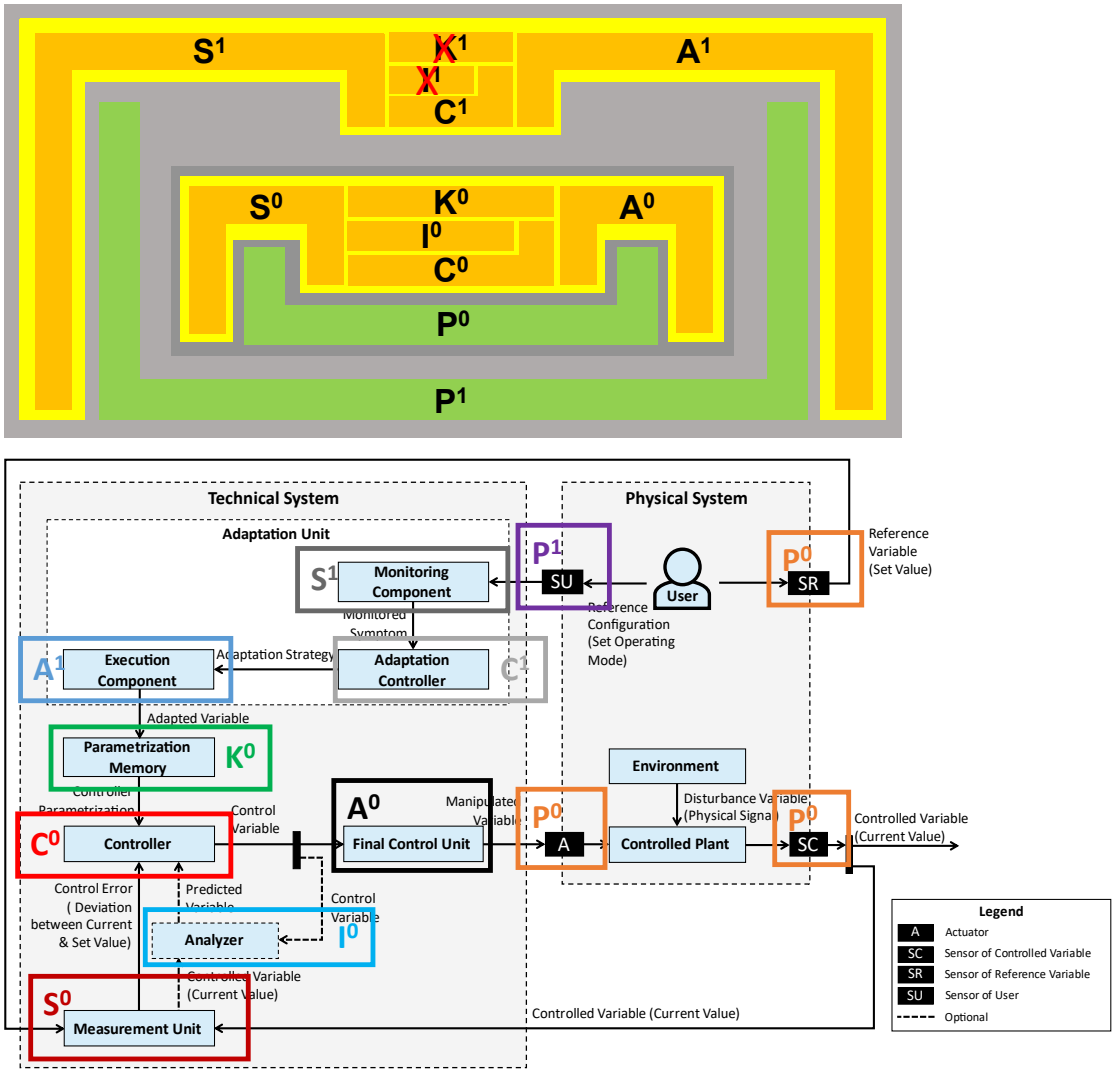


Figure 4.10: Instantiation of Architecture Pattern for Naive Adaptive Control

Compared to the yellow-green architecture, it can be observed that the functionalities of the introduced components in blue-grey architecture correspond to the sensing, control, and actuating components within the generic architecture style. Since the components in blue-grey architecture are deployed on the secondary control loop, the corresponding

components in yellow-green architecture are described as "S^1", "C^1" and "A^1". Superscript "1" means the index of the higher layer in the hierarchical architecture, as shown in Figure 4.10.

4.3.3 Controlled-Plant-Dependent Adaptive Control following the Generic Architecture Style

The architecture of controlled-plant-dependent adaptive control will now be taken as a further investigation object. The blue-grey architecture from the view of control theory, as previously introduced, is shown at the bottom of Figure 4.11. In comparison to the blue-grey architecture of naive adaptive control, an additional interpreting component is deployed on the secondary control loop, which takes the responsibility for evaluating the generated symptom and thus provides a change of request for adaptation activity to the adaptation controller. The functionality of the interpreting component in the blue-grey architecture corresponds exactly to the interpreting component in the yellow-green architecture, which builds a clear mapping relationship. The remaining components stay the same as in the case of naive adaptative control.

In addition to the interpreting component, another supplement compared to naive adaptive control is that controlled-plant-dependent adaptive control has access to the component sensor of controlled plant (*SP*) for providing additional context information to the adaptation unit. In this case, it can be said that the observation scope of the technical system (yellow part of the yellow-green architecture) for the physical system is further increased, which is represented by the integration of the sensor of controlled plant. The knowledge component "K^1" is still missing since there is no knowledge management component similar to the parametrization memory component in the adaptation unit of the blue-grey architecture.

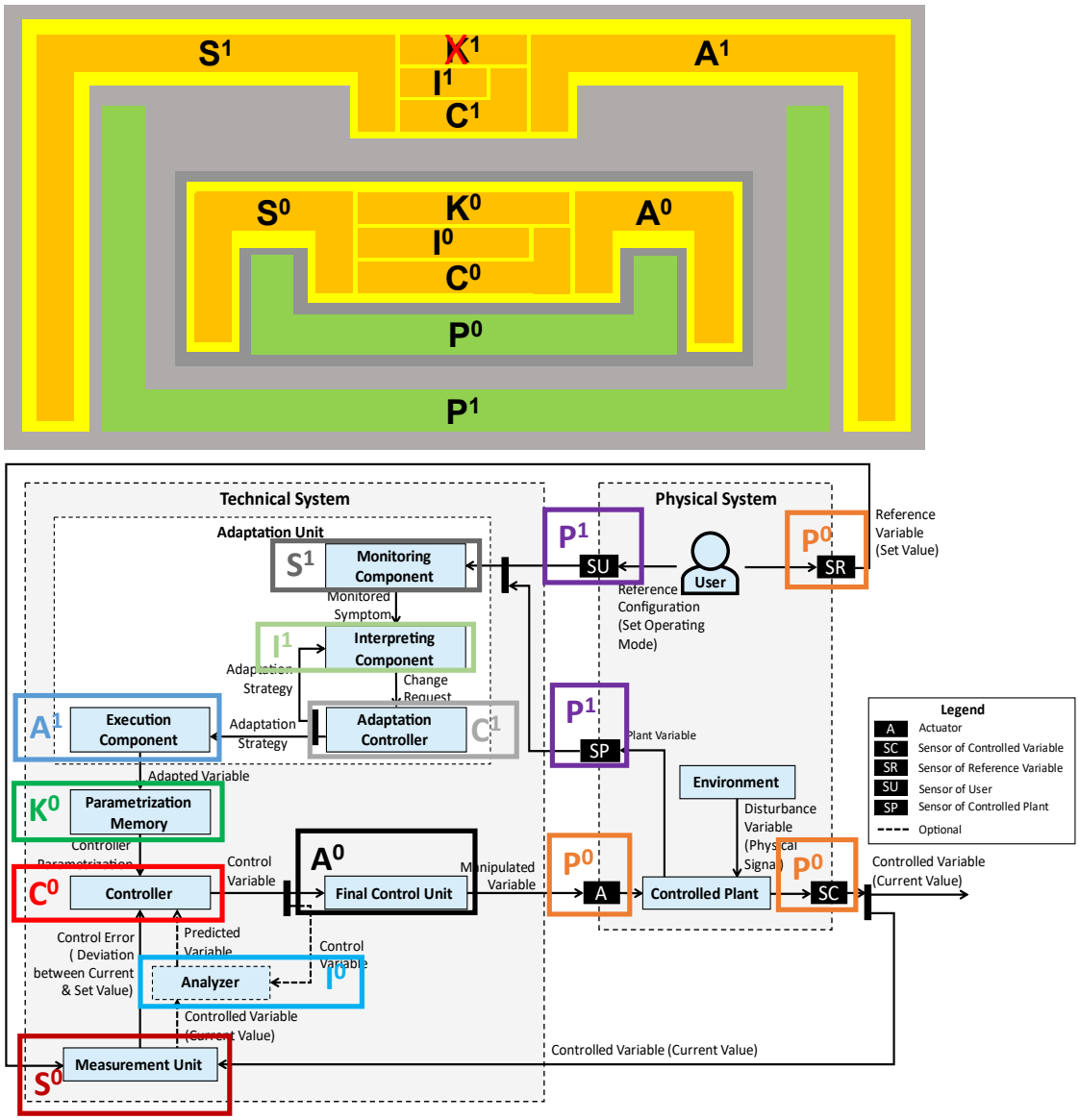


Figure 4.11: Instantiation of Architecture Pattern for Controlled-Plant-Dependent Adaptive Control

4.3.4 Physical-System-Dependent Adaptive Control following the Generic Architecture Style

The next investigated concept of current control systems is called physical-system-dependent adaptive control. The architecture considering control theory is illustrated as the blue-grey architecture in Figure 4.12.

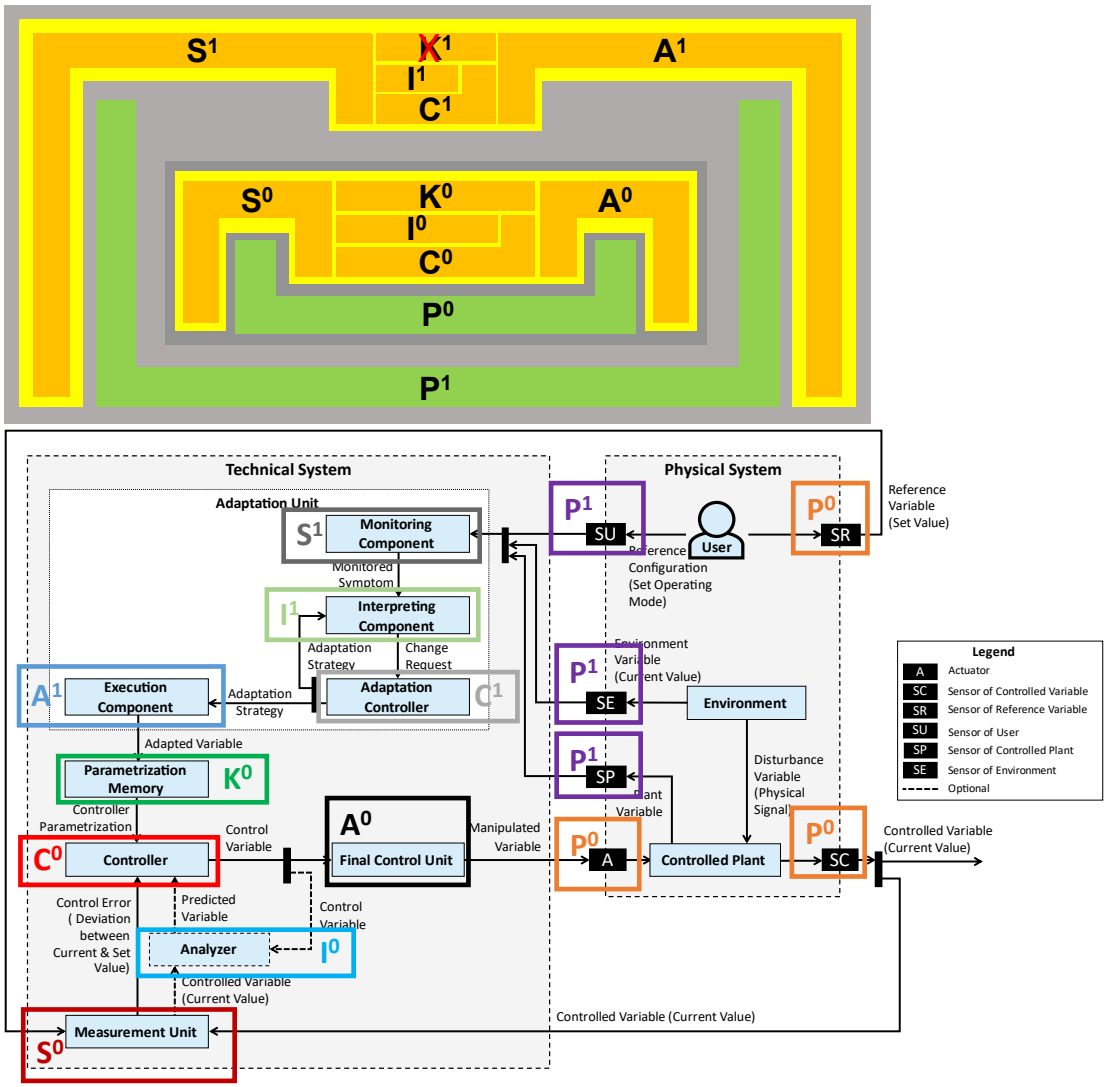


Figure 4.12: Instantiation of Architecture Pattern for Physical-System-Dependent Adaptive Control

In the blue-grey architecture, the component sensor of environment (*SE*) is also connected to the adaptation unit on the secondary loop. Thus, it is also mapped to the physical system "P^1" on the higher layer of the hierarchical architecture. Unfortunately, the knowledge component "K^1" is still missing, similar to the case of controlled-plant-dependent adaptive control.

4.3.5 Artificial Cognitive Control following the Generic Architecture Style

After instantiating the generic architecture style for current control systems, it is also important to check whether the proposed architecture pattern is consistent with artificial cognitive control, defined in this dissertation as the next generation of control systems (cf. Section 3.8).

In Section 3.6, it is clearly stated that current control systems have existing issues from two perspectives: (1) missing property of knowledge acquisition, and (2) adaptation and limited system scalability against fixed boundary conditions. Different challenges for the system architecture design of artificial cognitive control are constituted to eliminate these issues by integrating knowledge acquisition and adaptation and strengthening system scalability and connectivity. For this purpose, the artificial cognitive control system needs to include increasingly complicated knowledge basis as its world model and greater data processing, along with increasing offboard sensor access. However, such a requirement leads to a very high computation effort and long computation duration, thus strongly challenging performance, especially regarding the timing perspective of the control loop.

Against such a background, the design of networked system architecture, including multiple nodes as subsystems, is proposed as a solution. It aims to distribute the computations of control loops into different nodes included in the architecture. Thus, the bottleneck of high computation effort and duration can be overcome, relying on parallel computations of the nodes. Different knowledge components are then deployed as interface components between the nodes to isolate them and thus realize a loose coupling. They can also serve as knowledge repositories to deploy corresponding knowledge bases with limited complexity distributed from the previously highly complicated knowledge basis.

Following the paradigm of structural adaptation composition in the proposed generic architecture style, a networked architecture including multiple nodes can be constituted. Each node can be instantiated as a layer, and the layers can be hierarchically connected. Thus, a two-layered hierarchical yellow-green architecture can be derived as an example to describe an instantiated artificial cognitive control system from the view of software engineering, as shown at the top of Figure 4.13. It should be emphasized that the artificial cognitive control system is not limited to two hierarchical layers. Instead, it

can also be constructed with arbitrary hierarchical layers that can be separately deployed on different distributed domains if necessary.

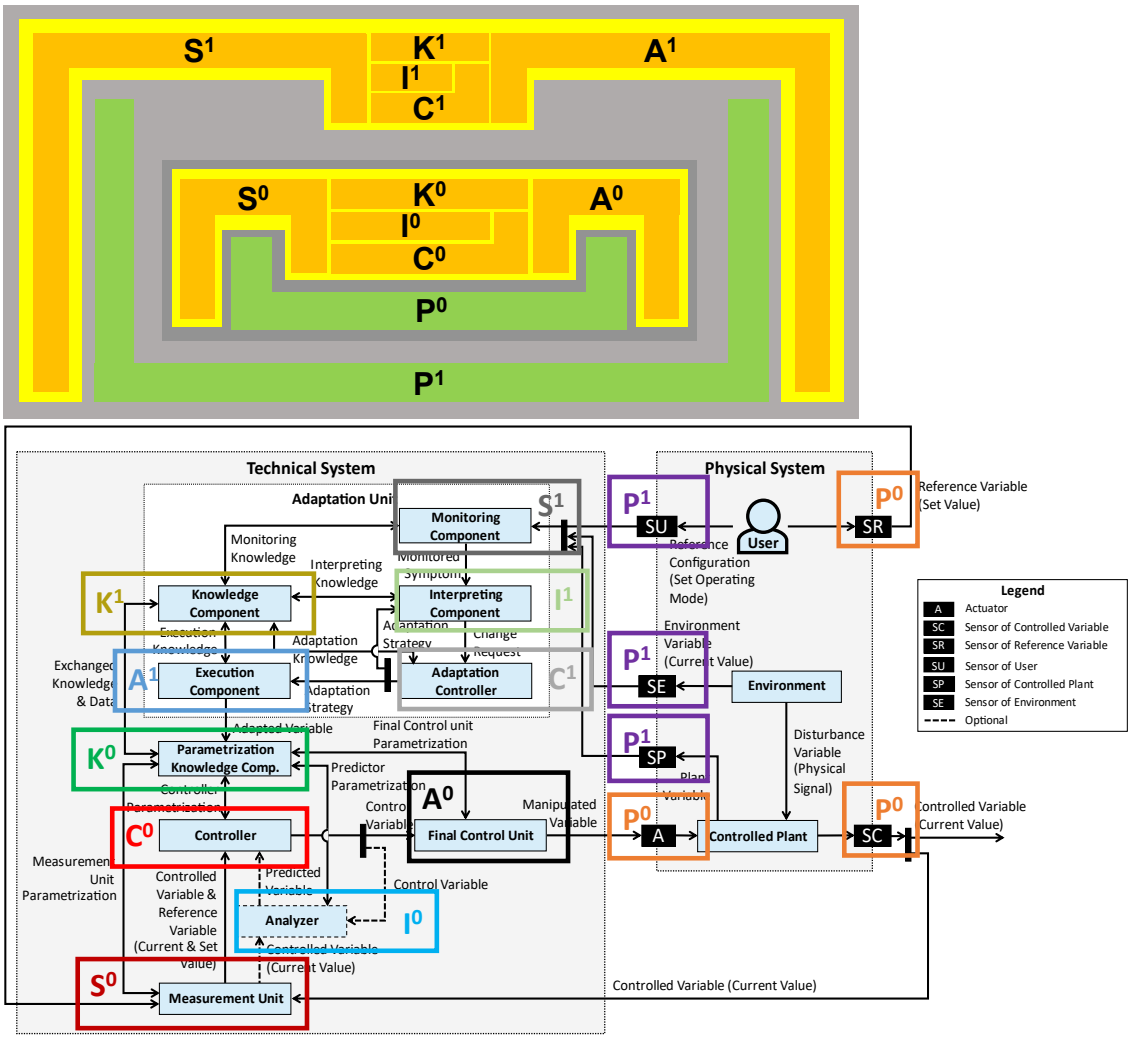


Figure 4.13: Instantiation of Architecture Pattern for Two-layered Artificial Cognitive Control System

Unlike the physical-system-dependent adaptive control, the final missing knowledge component ("K^1") is now included in the higher layer of the yellow-green architecture. Thus, knowledge acquisition and adaptation can be realized in artificial cognitive control, relying on the knowledge component. In the case of a more complicated knowledge basis or more data flow from offboard sensors, higher layers can also be constructed in the system architecture; these layers are not shown in the simplified illustration of Figure 4.13.

The blue-grey architecture is derived from the blue-grey architecture of physical-system-dependent adaptive control. A knowledge component (dark yellow box representing "K^1") is added to the adaptation unit. This knowledge component communicates with the monitoring, interpreting, and execution component, and with the adaptation controller to initialize their required domain knowledge. From the opposite direction, they can also acquire new domain knowledge and update it into the knowledge component to realize the knowledge acquisition and adaptation, as described in use case 2 (UC2) of the dynamic system behaviors (cf. Section 4.2.6.2).

Another difference in the blue-grey architecture of physical-system-dependent adaptive control is that the knowledge component "K^1" on the higher layer is evolved from a pure knowledge repository for storage into a component for knowledge management covering more functionalities. Thus, in artificial cognitive control, the knowledge component "K^1" can communicate with the parametrization knowledge component "K^0" on the lower layer to realize the use case 4 (UC4): knowledge acquisition and sharing across multiple nodes (detailed process flow cf. Section 4.2.6.4).

4.4 Summary

Since automatic control systems become increasingly more complicated, system design becomes an issue with greater challenges, particularly in the case of a sophisticated system architecture with hybrid computation mechanisms, which is exactly what is required in next-generation artificial cognitive control. This dissertation combines control theory and software engineering, aiming to acquire another view of the automatic control system and thus contribute to system design with the help of established software engineering approaches.

With this idea in mind, this chapter first provides a short research overview of related works about combining approaches of control theory and software engineering. Based on the understanding of the related works, the main contribution of this dissertation is introduced in detail, which consists of a generic architecture style for the design of different automatic control systems. The fundamental design of the generic architecture style, including the fundamental component structure and the system construction paradigm of adaptation composition, is introduced. In addition to the view of static

system construction, this chapter also introduces different standard communication architecture patterns and triggering mechanisms that can be taken to define concrete component interactions within several defined use cases of dynamic system behaviors. Finally, to evaluate the generic architecture style, the generic architecture style is used to instantiate logical architectures (from the view of software engineering) as concrete examples for different control systems. Thus, the logical architectures can be compared with the architectures from the view of control theory to check constructional consistency.

5 Artificial Cognitive Cruise Control as Experimental Application of Generic Architecture Style

Addressing the trend that automatic control systems are growing increasingly more complicated, the approaches of hierarchical system architecture and the corresponding challenges of system design were presented in Chapter 3. Aiming to fulfill the challenges due to the more complicated system architecture, the main contribution of this dissertation, a generic architecture style for designing automatic control systems from the view of software engineering, was proposed and introduced in detail within Chapter 4. An overview of related works by combining control theory and software engineering approaches was described.

Subsequently, the fundamental design of the generic architecture style was introduced, including the paradigm of adaptation composition for system construction and fundamental component structure. Different triggering mechanisms for the design of system computation were also introduced. Additionally, a short introduction to different generic communication architecture patterns (cf. Section 2.3), which can be applied to specify the component interactions consisting of dynamic system behaviors, is also included. Several use cases with different component interactions were also presented to describe the dynamic system behaviors.

To validate the proposed generic architecture style, several architectures from the perspective of software engineering were derived as instantiations and compared with architectures of current control systems from the control theory viewpoint, with the aim of investigating architectural consistency. A two-layered example architecture of artificial cognitive control as a generally simplified instance was also included in the investigation.

Further, the generic architecture style is taken again to derive an example architecture of the artificial cognitive control and applied within the vehicle's ACC function as a practical application called artificial cognitive cruise control (ACCC) in this dissertation. This chapter will introduce the detailed architecture of the ACCC and an implemented prototype. Additionally, the technical performance of the implemented prototype will also be evaluated.

5.1 Preliminary Design of Artificial Cognitive Cruise Control

The reviewed architectures of current ACC variants (cf. Section 4.3) indicate that most of these architectures include a two-layered construction. While the lowest layer is responsible for the real-time control to interact with the physical world within milliseconds, the higher layer is responsible for determining a high-level context-based adaptation strategy with a longer cycle time. In this sense, it can be said that current ACC variants with two-layered architectures (cf. Section 4.3.2–4.3.4) are extended gradually based on the classical ACC with a single-layered architecture (cf. Section 4.3.1). As introduced earlier, the architecture design of artificial cognitive control systems in next generation should follow the proposed ideas of adaptation composition in the generic architecture style. With this idea in mind, the layer design in the system architecture of ACCC as an empirical application example of artificial cognitive control system needs to be reconsidered.

As presented earlier, ACC is designed as a driving comfort assistance system to take over the vehicle's longitudinal control during a trip to realize so-called semi-automated driving. In this case, the driver must still participate in the driving task by steering the vehicle. Following the generic architecture style, different nodes have different observation scopes. Since current ACCs have layered architectures, the architecture of ACCC is also instantiated to include a multi-layered topology. Thus, it is obvious that the largest scope could be assigned to the highest layer, which in the case of ACC refers to the complete remaining trip, considering the whole route between the origin where the vehicle is currently located and the desired destination.

In this case, the highest layer would focus more on the so-called global planning of driving strategy. However, global planning cannot foresee every event during the trip in advance due to nondeterministic in reality. Thus, other traffic participants on the route that could be obstacles for the ego-car, such as moving vehicles or pedestrians, will be completely ignored. For this reason, current ACCs' original task of planning driving strategy on a lower level of abstraction such as a set of cruise velocities for the whole route is not worth being included on the highest layer.

Instead, the highest layer would work on a similar level of abstraction to the vehicle's navigation system. The difference is that the navigation system focuses on driving behavior planning like turning left or right, with a set of intermediate points like intersections as nodes on the route. However, the highest layer of the ACCC plans a high-level set travel profile (like the expected travel time and energy consumption) from the origin until each intermediate point on the following route. Thus, specific high-level domain knowledge about the physical system, such as the driver-preferred average travel time or average energy consumption, can be extracted based on the observed facts during the whole trip and learned by the technical system to realize the proposed personalized ACC (cf. Section 3.5.1). Since such planning of the high-level set travel profile is not required to be performed regularly, the highest layer can be designed as event-triggered (cf. Section 4.2.4).

Considering another perspective, the ACCC still needs to interact with the physical world while driving. As a real-time automatic control system, the lowest layer in the architecture of ACCC is thus required to deploy a well-known traditional closed control loop, as previously introduced in the concept of basic control (cf. Section 3.1). Thus, it means that the lowest layer is still required to focus on the determination of concrete low-level control activity like cruise velocity or headway within the range of milliseconds. Since the lowest layer is required to guarantee real-time interaction with the physical world, it can be designed to be time-triggered (cf. Section 4.2.4) to fulfill timing requirements.

Compared with the high-level personalized set travel profile mentioned before, it is indicated that the high-level personalized set travel profile (e.g., the driver's expected average travel time and the energy consumption) is still out of touch with the concrete control activity within the range of milliseconds. This also means that the set travel profile cannot directly be decomposed as individual concrete control activities due to the absence of an appropriate intermediate level of abstraction between them. Thus, an additional middle layer is required. A three-layered architecture for the preliminary design of artificial cognitive cruise control was conceived with this idea in mind, as shown in Figure 5.1.

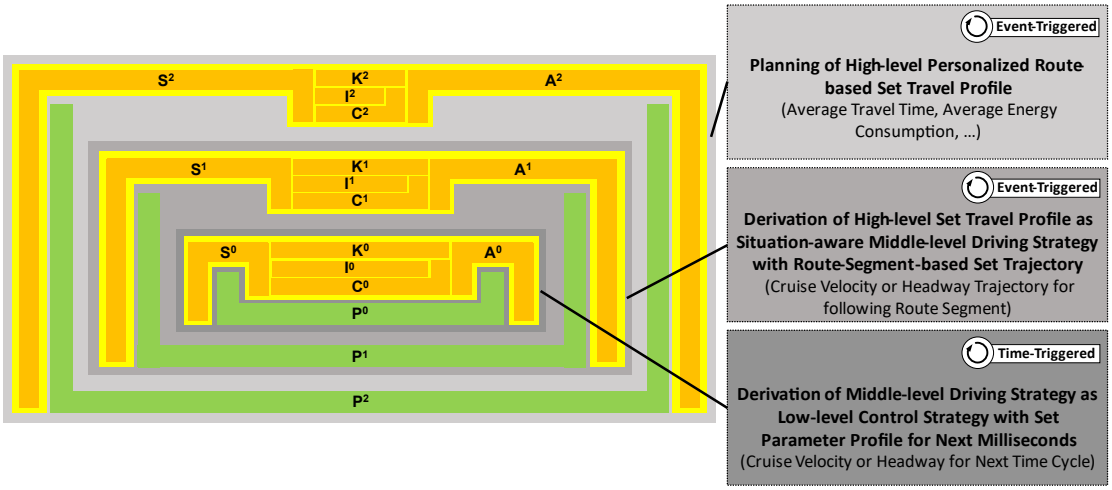


Figure 5.1: Preliminary Design of Three-Layered Architecture of Artificial Cognitive Cruise Control (ACCC)

The highest layer's planned high-level personalized set travel profile would be transferred into the middle layer in the architecture. The middle layer is responsible for deriving the set travel profile as a situation-aware middle-level driving strategy. The middle-level driving strategy includes a set trajectory of cruise velocity or headway, focusing on a certain following route segment with a limited distance horizon instead of focusing on the whole route. Since the middle-level driving strategy is situation-aware, the middle layer needs to be designed as event-triggered. The high-level set profiles of travel time and energy consumption can be understood as specifications for the middle layer to take as constraints during trajectory planning. Subsequently, the lowest layer can take the trajectory coming from the middle layer further as a specification in its determination of low-level control activity with concrete set value of cruise velocity or headway, which is called the set parameter profile in this dissertation.

By applying such a concept, three different hierarchically connected layers as three instantiated networked nodes would have three different observation scopes. The observation scopes vary from the whole route, to a route segment, to a certain limited time horizon, depending on the required cycle time of the closed control loop. Thus, the increasing observation scopes are consistent with the ideas of the proposed generic architecture style. The higher two layers are triggered by events since they work on higher semantic levels and include higher-level domain knowledge like the individual driver's personalized preferences. In constrast, the lowest layer works based on a low

level of variable monitoring and precise actuator manipulation, without any semantic understanding of the physical system regarding the driver or the driving environment.

5.2 Instantiation of Generic Architecture Style for ACCC System Architecture: Static System Construction

After a short introduction in the previous section, a preliminary design for a three-layered architecture of the ACCC was introduced. To further flesh out this preliminary design, the proposed generic architecture style is applied to instantiate an example of logical architecture on a lower component level from the viewpoint of software engineering, illustrated as the yellow-green architecture in Figure 5.2. Subsequently, another example of architecture (blue-grey) from the view of control theory can also be derived, as illustrated at the bottom of Figure 5.2.

In Figure 5.2, it is indicated that the ACCC has a hierarchical architecture (blue-grey) with three layers. On the layers, a route-based adaptation unit, a route-segment-based adaptation unit, and a cycle-time-based control unit are deployed as three subsystems within the technical system. With the help of different sensors, different units in the technical system acquire increasing observation scopes of the physical system from bottom to up, along with the hierarchical layers. The fundamental component structures consisting of “SICAP-K” exist on each layer in the hierarchical architecture, illustrated as color boxes inside the subsystems of the three units in Figure 5.2.

As presented in the previous section, the observation scopes of the physical system by different units on different layers must be predefined. The route-based adaptation unit on the highest layer makes adaptation decisions considering the whole route profile. However, the route-segment-based adaptation unit on the middle layer considers only the following section of the route, which is called the *route segment* in this dissertation. Unlike the route-based and route-segment-based adaptation unit, the cycle-time-based control unit on the lowest layer has an observation scope from the time perspective instead of the spatial perspective of the physical driving environment. It focuses on the determination of control activity for the following cycle time within milliseconds. In the following sections, more details about these units and other subsystems in the architecture of ACCC will be provided.

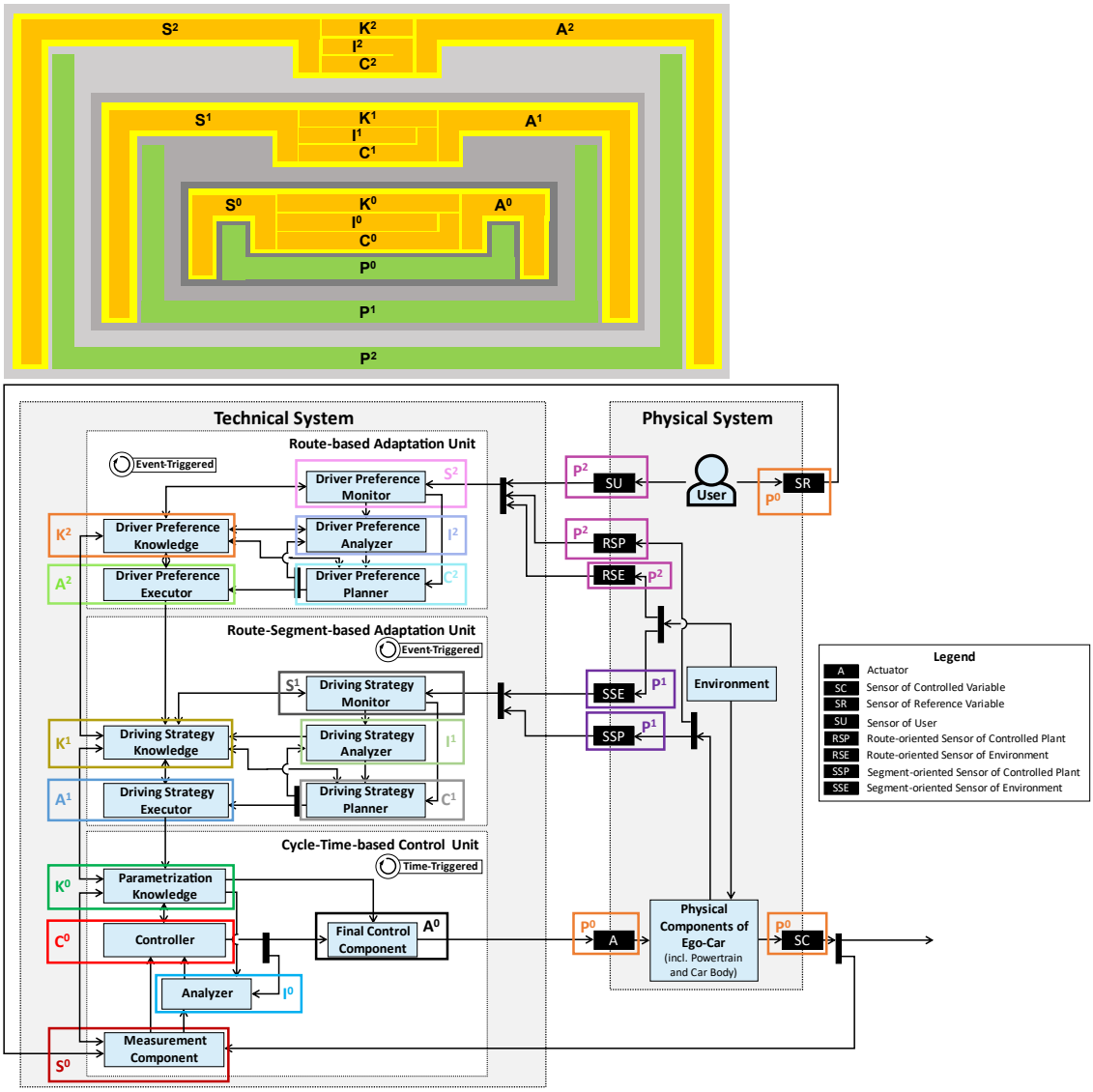


Figure 5.2: Instantiated Architecture of Artificial Cognitive Cruise Control (ACCC) from the Generic Architecture Style

5.2.1 Physical System

As stated in Figure 5.2, similarly to current ACCs (cf. Section 3.1–3.4), the ACCC consists of two major parts: a technical system and a physical system. The physical system consists of the driver, the controlled plant referring to the ego-car's physical components like the car body and powertrain, and the surrounding driving environment (cf. Figure 5.3). As presented earlier, the sensors and actuators play roles as interfaces to enable interaction between the technical and physical system, which are also

illustrated in Figure 5.3 and categorized into three classes ("P^0", "P^1", and "P^2"). The superscripts correspond to three layers with different observation scopes in the hierarchical architecture instantiated from the generic architecture style.

On the highest layer, the component sensor of user (*SU*), the component route-oriented sensor of controlled plant (*RSP*), and the component route-oriented sensor of environment (*RSE*) are responsible for the delivery of sensory data for the route-based adaptation unit. The data coming from the RSE and RSP include context information about the vehicle and its driving environment. Unlike these two sensors, the sensor SU interacts with the driver.

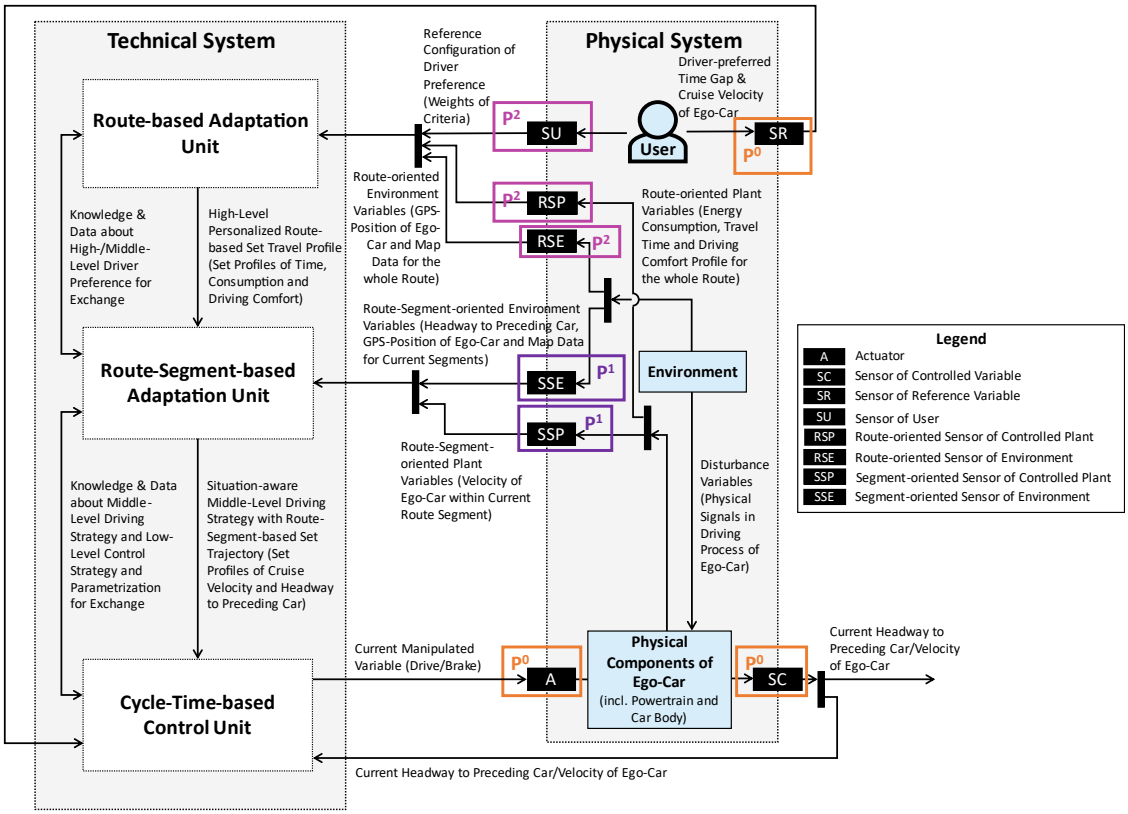


Figure 5.3: Physical System in Artificial Cognitive Cruise Control

This dissertation defines the quality of the set travel profile with a cost function influenced by travel time, the vehicle's fuel/energy consumption, and the driving comfort represented by the vehicle's acceleration/deceleration. In this cost function, different weights of influence factors are required. The driver can manually adjust the weights of

the influence factors through the SU if necessary (e.g., via a human-machine interface (HMI) of the vehicle's entrainment system). In this case, the HMI can also be interpreted as a special actuator and sensor simultaneously. Since HMI only indirectly instead of directly influences the technical process, which refers to the physical processes for the vehicle's driving, it is excluded from the system design of ACCC in this dissertation and thus is not visualized in the system architecture of Figure 5.3.

A significant point that must be emphasized is that all these mentioned sensors are not limited to explicit physical sensors. For example, the SU possibly consists of a group of independent physical sensors, which are all responsible for delivering driver-relevant sensory data since the driver is the system user in this case. In addition to driver-relevant data, the sensors RSP and RSE provide vehicle- and route-relevant data (e.g., the driver's accumulated travel time for the whole route and the GPS position of the ego-car).

Instead of focusing on the whole route, the segment-oriented sensor of environment (*SSE*) and the segment-oriented sensor of controlled plant (*SSP*) on the middle layer of the hierarchical architecture focus on providing the data within the observation scope of individual route segment (e.g., the driver's preferred cruise velocity and headway to the preceding car within the route segment). Along with the car's movement during the trip, the observation scope of the sensors is also moving towards the following route segment like a sliding window, depending on the current GPS position of the ego-car and its located route segment respectively.

On the lowest layer, the component sensor of controlled variable (*SC*) and the component sensor of reference variable (*SR*) provide sensory data about current and the driver-preferred set cruise velocity and headway, respectively, as in the case of current ACCs (cf. Chapter 3). Additionally, the actuators (*A*) receive the current value of manipulated variables to manipulate hardware in the ego-car's engine and brake system. In this case, both sensors and actuators on the lowest layer only focus on concrete activities for the current time point without considering the following route or route segments.

5.2.2 Route-Based Adaptation Unit in the Technical System

As indicated in Figure 5.2, the route-based adaptation unit is designed as a subsystem triggered by trip-relevant events. It works only when trip-related requirements have been fulfilled. For example, it plans the driver's individual personalized global travel profile, including set travel time, set energy consumption or set driving comfort, at the beginning of the trip. Additionally, it updates the personalized travel profile once the planned trip has been changed or strongly violated while driving. For instance, in the case of a traffic jam, the vehicle's navigation system would normally react, and thus this can be seen as a potential trigger for the route-based adaptation unit to update the set travel profile.

Once the ego-car has reached the planned destination and the trip is finished, the route-based adaptation unit stops the data collection. It then triggers the process of learning high-level route-based driver preferences like the driver's preferred travel time or energy consumption as high-level domain knowledge based on the newly observed facts of sensory data during the whole trip.

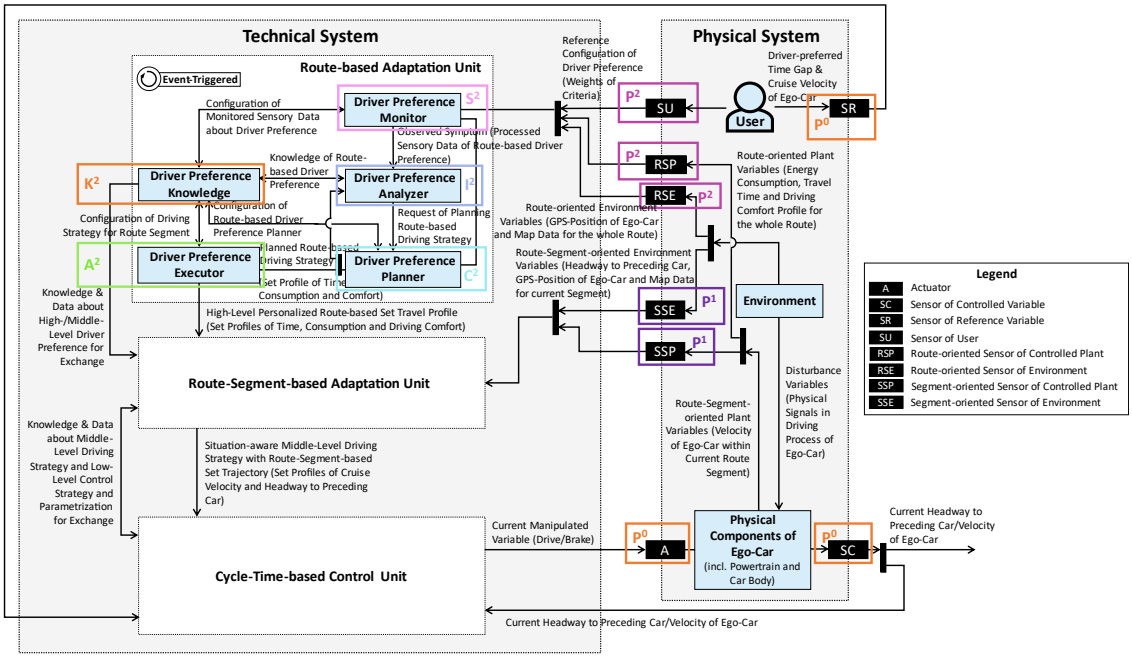


Figure 5.4: Route-Based Adaptation Unit in Artificial Cognitive Cruise Control (Detailed View cf. Appendix A.1)

The route-based adaptation unit (*RAU*) follows the proposed generic architecture style to realize the aforementioned functionalities. It means that the RAU is constructed by following the "SICAP-K" fundamental component structure consisting of five components: *Driver Preference Monitor* ("S^2"), *Driver Preference Analyzer* ("I^2"), *Driver Preference Planner* ("C^2"), *Driver Preference Executor* ("A^2") and *Driver Preference Knowledge* ("K^2"), as illustrated in Figure 5.4. Detailed dynamic behaviors of these components will be presented in Section 5.3 based on different predefined use cases.

The driver preference knowledge component is a centralized knowledge repository to initialize the required domain knowledge for the other four components. It can also support other knowledge components in the other two units by providing its available domain knowledge. More detailed processes about this support of domain knowledge will be presented in Section 5.3.4.

The driver preference monitor generates the symptom defined as well-structured sensory data in this dissertation. As illustrated in Figure 5.4, the included data in the symptom come from different sensors (*SU*, *RSP*, and *RSE*), which were presented in Section 5.2.1. The driver preference monitor preprocesses and also, if necessary, aggregates the collected sensory data. In addition, the collected sensory data may include a slight temporal offset due to the computing frequencies of different sensors. In this case, the driver preference monitor also completes the time synchronization of the input data. Aiming to complete the aforementioned tasks, the driver preference monitor needs related domain knowledge. For example, a data specification is required while preprocessing and aggregating the input sensory data. Thus, the generated symptom conforms to the data structure expected by the other components in the route-based adaptation unit.

The driver preference analyzer is designed to analyze the physical system's current and future context information as an interpreting component. For this purpose, the observed facts (included in the symptom) like the ego-car's current location will be compared with a catalog of previously available symptoms to represent the system's known situations like trip origin and destination. Thus, the driver preference analyzer could know whether the trip has begun and ended and thus whether a request for (re-)planning the driver's personalized set travel profile is essential. In this sense, it can be said that the route-

based adaptation unit is aware of the driving situation and works on an interpreted higher semantic level instead of the low data level.

The driver preference planner aims to find an optimized set travel profile to satisfy the driver's route-based preferences as far as possible if the driver preference analyzer generates the request. The planning strategy is an optimization process. For this purpose, an appropriate optimization algorithm and an accompanying decision tree for describing the solution space of the candidate set travel profiles must be implemented in the driver preference planner. In addition, the evaluation criteria and the interpretation mechanism for identifying qualities of different strategies, like a cost function consisting of a mathematical formula and the weights of the criteria, are required. In the implemented prototype, the weights of factors influencing the route-based preferences (collected by the sensor SU, cf. Section 5.2.1) are taken and considered as application parameters of the optimization algorithm.

The planned optimized set travel profile aims to fulfill the individual driver's route-based driving preferences as much as possible. For this reason, the driver's average profile is taken as a reference to define the evaluation criteria by following rules:

- Travel time shall be as short as possible, with a soft constraint of the driver's average travel time as the maximal acceptable deviation.
- Energy consumption shall be as little as possible, with a soft constraint of the driver's average energy consumption as the maximal acceptable deviation.
- Driving comfort shall be as great as possible, which means that the ego-car's acceleration shall be as gentle as possible with the maximal driver-acceptable acceleration rates (positive and negative) as soft constraints.

A concrete example of the cost function is implemented in this dissertation and will be further introduced in Section 5.5. Considering that the route is divided by significant intermediate route points like intersections as route segments, the implemented cost function includes three terms: (1) the driver-preferred accumulated travel time from the origin until each following route point, (2) accumulated fuel/energy consumption from the origin until each route point, and (3) the vehicle's average acceleration for representing the required driving comfort from the origin until each route point.

The driver preference executor decomposes the accumulated set profiles into a group of partial set profiles for route segments between each two neighbor route points. For this purpose, the driver preference executor requires domain knowledge like the route profile and specification of the route-segment-based partial set profiles provided by the driver preference knowledge component. Thus, the partial strategies can be taken as specifications during the planning strategy on a lower layer.

5.2.3 Route-Segment-Based Adaptation Unit in the Technical System

In addition to the route-based adaptation unit, another subsystem deployed on a lower layer is the route-segment-based adaptation unit. Like the route-based adaptation unit, the route-segment-based adaptation unit is also designed as an event-triggered subsystem. However, it is triggered either by events with dependence upon state change of the route segment or by events about the preceding obstacle's availability of the ego-car.

As presented earlier, the whole route has previously been decomposed as a sequence of route segments. Unlike the route-based adaptation unit, which focuses on the set travel profile for the whole route (until the destination), the route-segment-based adaptation unit takes the set travel profile as a specification to derive a so-called situation-aware middle-level driving strategy on a lower level of abstraction. This middle-level driving strategy includes a location-based set trajectory of cruise velocity (in the case of no preceding car) and headway (in the case of a preceding obstacle) for a limited distance horizon on the route.

For this purpose, each route segment is further divided into a set of subsections through further intermediate route points, which have a standard variable distance between each other, depending on the respective geographical profiles of the route like curvature and altitude. In the implemented ACCC example (cf. Section 5.5), this standard variable distance is simplified as a fixed distance of one meter to reduce the implementation's complexity. Thus, the trajectory of cruise velocity and headway includes a sequence of set values for these two variables that the vehicle should realize once it moves through each route segment's starting point and endpoint and its included intermediate route points.

As an event-triggered subsystem, the route-segment-based adaptation unit has two independent triggering conditions: (1) the state change of the route segment and (2) the state change of the preceding obstacle. The adaptation unit is activated to determine the driving strategy if any condition is fulfilled. Whether the trajectory of cruise velocity or headway will be planned depends on whether the radar sensor detects an obstacle ahead.

The first triggering condition means a state change when the ego-car leaves the previous route segment and enters following one. Thus, the driving strategy determination process repeats during the whole trip as the ego-car enters each segment. The route-segment-based adaptation unit repeatedly has an observation scope dynamically limited by the horizon of the following route segment, which works like a sliding window moving along the route.

The second triggering condition relates to the preceding obstacle's state. Current ACC variants (cf. Section 3.1–3.4) detect this solely by relying on the radar sensor's current state of sensory data. For this reason, the lack of so-called memory ability as a significant limitation of the ACC leads to critical scenarios, particularly in the case of a route with many curves (cf. Section 3.5.2). In the design of ACCC, this memory ability is considered in order to eliminate critical scenarios. The ACCC detects the preceding obstacle's availability based on the current sensory data and the dependency on the previously experienced driving context.

For example, a "virtual" preceding obstacle will be considered during planning the following driving strategy, even once the radar sensor cannot "see" a previously appeared but currently disappeared obstacle due to a curve. When two conditions are fulfilled, such "virtual" preceding obstacles would only be removed from the ACCC's memory. The first condition is that the full sensing range of the radar sensor is overlapped with the following route (from another perspective, this also means a limited lateral curvature of the following route). The second condition is that the obstacle is no longer located within the full sensing range. For this purpose, the detailed map data for each route segment is required by the route-segment-based adaptation unit, as illustrated in Figure 5.5.

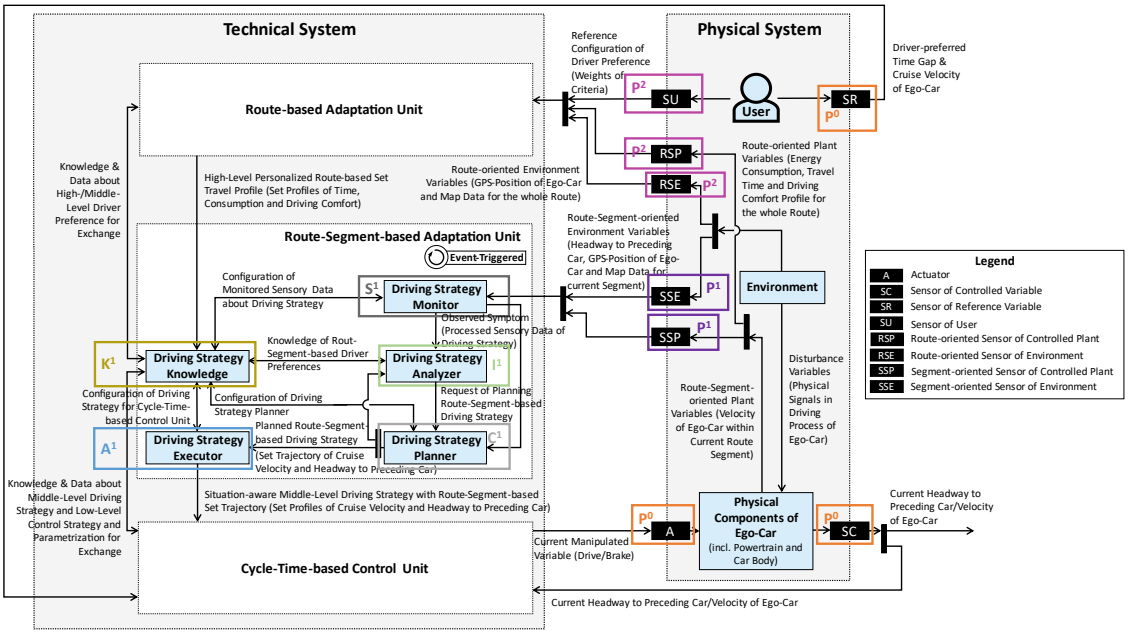


Figure 5.5: Route-Segment-Based Adaptation Unit in Artificial Cognitive Cruise Control (Detailed View cf. Appendix A.2)

The route-segment-based adaptation unit observes and collects the sensory data once the driver is manually driving on the route. The collected sensory data as newly observed facts are used to learn the driver's middle-level driving preferences, namely their preferred cruise velocity and headway within each route segment. Generally, the generic architecture style supports online and offline learning of such preferences. Nevertheless, in the implemented ACCC example provided here, offline learning is chosen to reduce implementation complexity. The learning process will only be activated when the ego-car has completed the trip.

Like the route-based adaptation unit, the route-segment-based adaptation unit is also constructed by following the proposed generic architecture style, which means that it is designed with a fundamental component structure of "SICAP-K" consisting of five components: *Driving Strategy Monitor* ("S^1"), *Driving Strategy Analyzer* ("I^1"), *Driving Strategy Planner* ("C^1"), *Driving Strategy Executor* ("A^1") and *Driving Strategy Knowledge* ("K^1"), as illustrated in Figure 5.5. More details of these components' dynamic behaviors will be introduced in Section 5.3.2 and Section 5.3.4.

The driving strategy knowledge component plays the role of the centralized knowledge repository that initializes the required domain knowledge for the other four components. In addition, it also requests knowledge support from and provides knowledge support to other knowledge components in the ACCC. Mode detailed processes about the dynamic behaviors of the driving strategy knowledge component will be presented in Section 5.3.2 and Section 5.3.4.

The driving strategy monitor is designed to generate the symptom by preprocessing and, if necessary, aggregating the received sensory data. For this purpose, the driving strategy monitor requires knowledge like data specification to preprocess the sensory data. Thus, the generated symptom's structure conforms to the preferred data specification.

The driving strategy analyzer includes the triggering conditions mentioned above to decide whether it is necessary to request the driving strategy planner to plan the middle-level driving strategy. For this purpose, an inference engine and a state machine must be implemented in the driving strategy analyzer. Since the memory ability is considered in the design of the route-segment-based adaptation unit, the driving strategy analyzer also includes functionality for predicting future driving behaviors like the preceding obstacle's velocity trajectory, which is also considered while planning the driving strategy.

The driving strategy planner aims to determine an optimized middle-level driving strategy consisting of the set trajectory of cruise velocity and headway. An optimization algorithm must be implemented in the driving strategy planner to complete the strategy determination. Additionally, a decision tree is required to describe the solution space comprising all candidate cruise velocities and headways. In the ACCC, the decision tree is derived from the previously observed facts about the driver's manual driving behaviors. Thus, the decision tree will be continuously extended and updated along with the knowledge acquisition and adaptation once a novelty is identified due to newly observed facts.

Generally, the optimized middle-level driving strategy used by the route-segment-based adaptation unit, which consists of the location-based set trajectory of cruise velocity and headway, should fulfill the driving preferences of the individual driver as much as possible. For this reason, the driver-preferred average profile of cruise velocity and

headway is taken as a reference to define the evaluation criteria through the following rules within the implementation example:

- In the case of no preceding obstacle, the planned cruise velocity shall be as equal as possible to the driver's average profile.
- In the case of a preceding obstacle, the planned headway shall be as equal as possible to the driver's average profile.
- The planned middle-level driving strategy, including the set trajectory of cruise velocity or headway, shall fulfill the specifications of the planned set travel profile, which is the high-level personalized strategy planned by the route-based adaptation unit.

Following the rules above, a cost function consisting of influencing factors and accompanying weights must also be defined in the driving strategy planner. This dissertation uses the same cost function to identify qualities of the candidate set trajectories in the decision tree (cf. Section 5.2.2), as in the case of the route-based adaptation unit. Three influencing factors regarding the travel time, the required fuel/energy consumption, and the driving comfort represented by the vehicle's acceleration and their weights (customized by the driver, cf. the sensor SU in Section 5.2.1), are considered in the cost function. Thus, the cost function can quantitatively evaluate the quality of candidate middle-level driving strategies. A concrete example of this cost function is implemented in this dissertation, which will be introduced in Section 5.5.

Once the optimized middle-level driving strategy has been defined, it will be forwarded to the component driving strategy executor ("A^1"). The set trajectory planned by the driver strategy planner consists of the set values of cruise velocity and headway for certain intermediate route points in each route segment. The driver strategy executor works similarly to the driver preference executor (cf. Section 5.2.2). It decomposes the location-based set trajectory into different partial sub-trajectories (e.g., the required cruise velocity or headway for each intermediate route point within each route segment). In the decomposition process, domain knowledge like the individual route segment's profile is required. Finally, sub-strategies are forwarded to the cycle-time-based control unit.

5.2.4 Cycle-Time-Based Control Unit in the Technical System

In Figure 5.2, it is indicated that the only time-triggered subsystem in the ACCC is called cycle-time-based control unit. As the name suggests, the cycle-time-based control unit repeatedly works with a deterministic cycle time, which means that each component inside works as an independent and active process with its scheduling, as introduced in Section 4.2.1. Such a computation mechanism aims to realize the real-time interaction with the physical world while driving. Thus, concrete cycle time for each component and the whole subsystem of the cycle-time-based control unit must be specified at design time, considering timing requirements in concrete applications.

As a subsystem deployed on the lowest layer in the ACCC, the cycle-time-based control unit takes over the responsibility of real-time feedback control to interact with the physical world. Such real-time feedback control as the most fundamental functionality in the control system is already included in current ACC variants. As noted earlier, the location-based set trajectory of cruise velocity and headway covering an individual route segment is provided by the route-segment-based adaptation unit. Each route segment is decomposed into different sections. Thus, the set trajectory is decomposed into a sequence of sub-trajectories for the sections in the route segment. Thus, each sub-trajectory includes two set values of cruise velocity and headway and accompanying GPS positions of two route points. With consideration of the distance profiles between the route points, the decomposed location-based sub-trajectories will then be converted into time-dependent sub-trajectories consisting of the set values of cruise velocity and headway. Thus, the control system uses the quantitative set values as dynamically changeable desired values of the reference variable (cf. Section 2.1.1). In this case, the cruise velocity and headway are taken as the ego-car's states expected by the driver once the ego-car moves through corresponding geographical route points within the route segment.

The cycle-time-based control unit relies on variable monitoring and tries to maintain the set values, depending on the current location of the ego-car. Compared to the route-based and route-segment-based adaptation unit, the most significant difference between the cycle-time-based control unit is that it works on the low data level instead of a higher semantic level. For this reason, the control unit does not request access to

map data for environmental perception. Instead, the current location of the ego-car represented by the GPS data only means a group of quantitative values of corresponding variables without any interpreted semantic meaning such as longitude, latitude, or altitude, since the map data is not considered in the cycle-time-based control unit.

Like the route-based and the route-segment-based adaptation unit, the cycle-time-based control unit (CTCU) also has a learning ability to realize knowledge acquisition and adaptation (cf. Section 3.6.1). The other two adaptation units focus on learning the driver's driving preferences on different levels of abstraction. The control unit focuses on the vehicle's internal operating strategy (e.g., to accelerate or decelerate the ego-car as much like the human driver as possible under the same boundary conditions, such as set cruise velocity and set headway). Although the learning ability of CTCU is supported by the ACCC's design, this dissertation removes it from the implementation to reduce complexity.

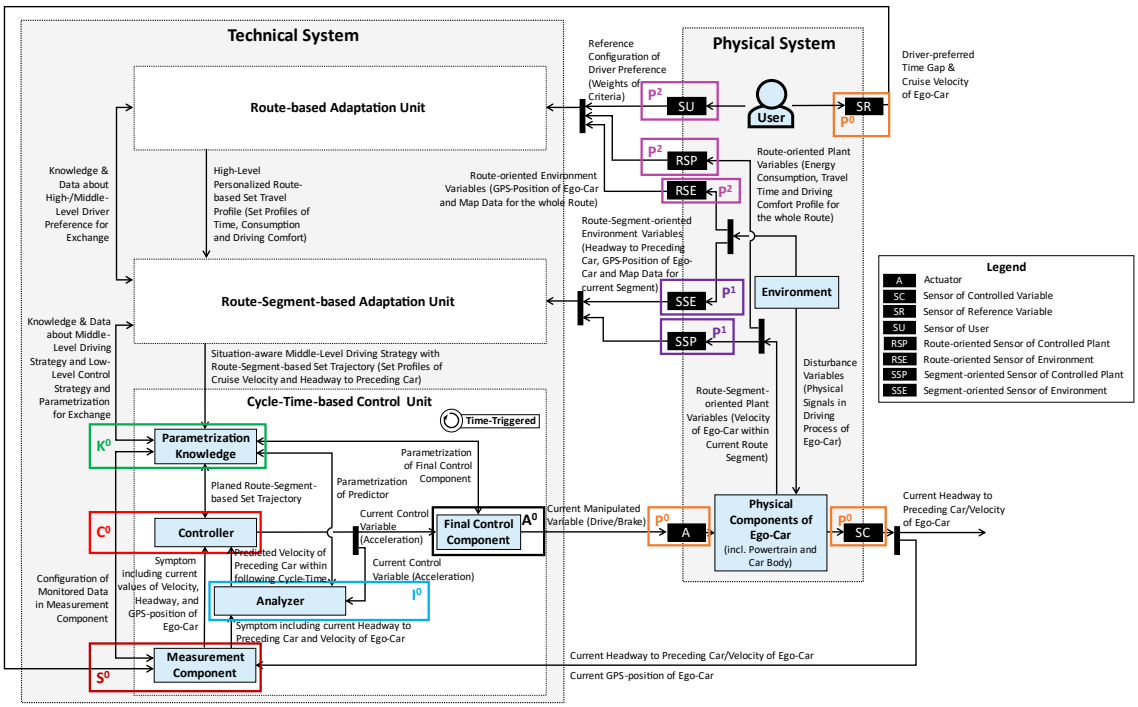


Figure 5.6: Cycle-Time-Based Control Unit in Artificial Cognitive Cruise Control (Detailed View cf. Appendix A.3)

The system construction of the cycle-time-based control unit follows the proposed generic architecture style. This means that the cycle-time-based control unit is designed with the fundamental component structure of “SICAP-K”, including five components: *Measurement Component* (“S^0”), *Analyzer* (“I^0”), *Controller* (“C^0”), *Final Control Component* (“A^0”) and *Parametrization Knowledge* (“K^0”), as illustrated in Figure 5.6.

The measurement component generates the symptom by preprocessing the collected sensory data. In the case of a detected preceding obstacle, the analyzer will forecast its future velocity trajectory. Unlike the prediction in the driving strategy analyzer, the prediction only forecasts the trajectory of the variable’s values for a limited time horizon instead of the horizon of the route segment, without high-level semantic interpretation like environmental perception. The prediction’s time horizon depends on the component controller's working horizon. The predicted velocity trajectory will consider the individual cycle time once the component controller generates the control command only for the next cycle time. Otherwise, a period of multiple following cycles will be considered.

The analyzer’s output variable's values representing the predicted velocity are directly taken as a state identifier of the current control task. The controller component relies on variable monitoring to know whether a preceding obstacle is visible and whether it should maintain the set value of cruise velocity or headway. Different classical control theory approaches for designing the controller, like the PID control and the model predictive control (MPC), can be applied to implement the component controller (cf. Section 2.1).

Lastly, the final control component converts the control variable to the manipulated variable to directly manipulate the actuator. If the controller only focuses on the individual following cycle time (e.g., using PID control instead of MPC), the final control component directly forwards the control command generated by the controller to the actuator to interact with the physical system. Otherwise, it also decomposes the control command into a sequence of sub-commands for individual cycle time and sequentially forwards them to the actuator.

In the ACCC prototype of this dissertation, the PID control was implemented to reduce implementation complexity. This means that the implementation of the analyzer was

also strongly simplified by removing forecast function. Only the detection of preceding obstacle was implemented in the analyzer.

5.3 Instantiation of Generic Architecture Style for ACCC System Architecture: Dynamic Behaviors in Use Cases (UCs)

After introducing the ACCC's static system construction with several subsystems, the predefined four use cases (UCs) from Section 4.2.6 are used again in this section to describe the ACCC's dynamic system behaviors in different situations. Although the proposed generic architecture style allows component interactions for all four of these use cases, it should be emphasized that not all use cases are relevant for all subsystems and their included components in the implemented ACCC prototype. Due to the design of the ACCC, only use cases with relevant subsystems will be included in the following sections.

5.3.1 Dynamic Behaviors of ACCC in UC1

The first use case focuses on the technical process control in a single node of the networked system architecture, as introduced in Section 4.2.6.1. In the ACCC, the networked architecture style is instantiated as a three-layered architecture. Each layer includes a subsystem and represents a single node. Thus, UC1 not only relates to the route-based and the route-segment-based adaptation unit but also the cycle-time-based control unit. As presented earlier, the so-called "SICAP-K" component structure is deployed on each layer of the ACCC. In the ACCC's UC1, the technical process control means that the component structure on each layer directly manipulates actuators to interact with the physical system.

Although the generic architecture style allows such direct actuator manipulation, it is excluded in the design of the route-based and the route-segment-based adaptation unit in ACCC. This point can be identified through the fact that both adaptation units deployed on the upper two layers have no direct connection to the physical system, as shown in Figure 5.2. Nevertheless, the route-based adaptation unit also has other ways to interact with the physical system. For example, the planned high-level strategy can be visualized via human-machine interfaces (HMI) on the vehicle as a suggestion to

request the driver's confirmation, in which case HMI can also be interpreted as a kind of special actuator and sensor simultaneously. Since HMI only indirectly instead of directly influences the technical process, which refers to the physical processes for the vehicle's driving, it is excluded in the system design of ACCC in this dissertation and thus is not visualized in the system architecture of Figure 5.2.

Figure 5.7 provides an overview of the components that participated in UC1 and their interactions. As presented earlier, the route-based and route-segment-based adaptation units do not interact directly with the physical world. Thus, it can be seen in Figure 5.7 that no feedback is forwarded by the final component in the effect chain of the technical process control to the physical system. However, this is not the case with the cycle-time-based control unit.

In the cycle-time-based control unit, UC1 means the control for the ego-car's longitudinal movement in real-time. Generally, the control concept here has no difference compared to concepts of current ACC variants. Firstly, the measurement component is responsible for collecting the raw sensory data and subsequent data preprocessing, aiming to generate a symptom, which only includes current value profiles of relevant variables (e.g., the velocity and headway and the GPS position of the ego-car).

In the case of no preceding vehicle, the analyzer estimates the current context information and forwards it to the controller. Additionally, the symptom generated by the measurement component is also forwarded to the controller component. Thus, the controller component can determine an optimized value of the control variable to accelerate and decelerate the ego-car. In this case, the controller's target is to guarantee that the velocity of the ego-car is as close as possible to a corresponding set value, which is selected from the set trajectory of cruise velocity, depending on the ego-car's current GPS position. For this purpose, the set trajectory should be guaranteed as much as possible during the determination process. Relying on the final control component, the input of the control variable is then transferred into the output of the manipulated variable used to manipulate the hardware actuators directly. For example, in the case of a car with an internal combustion engine (ICE), the control variable could be the acceleration, and the manipulated variable for acceleration could be the throttle position [84][9].

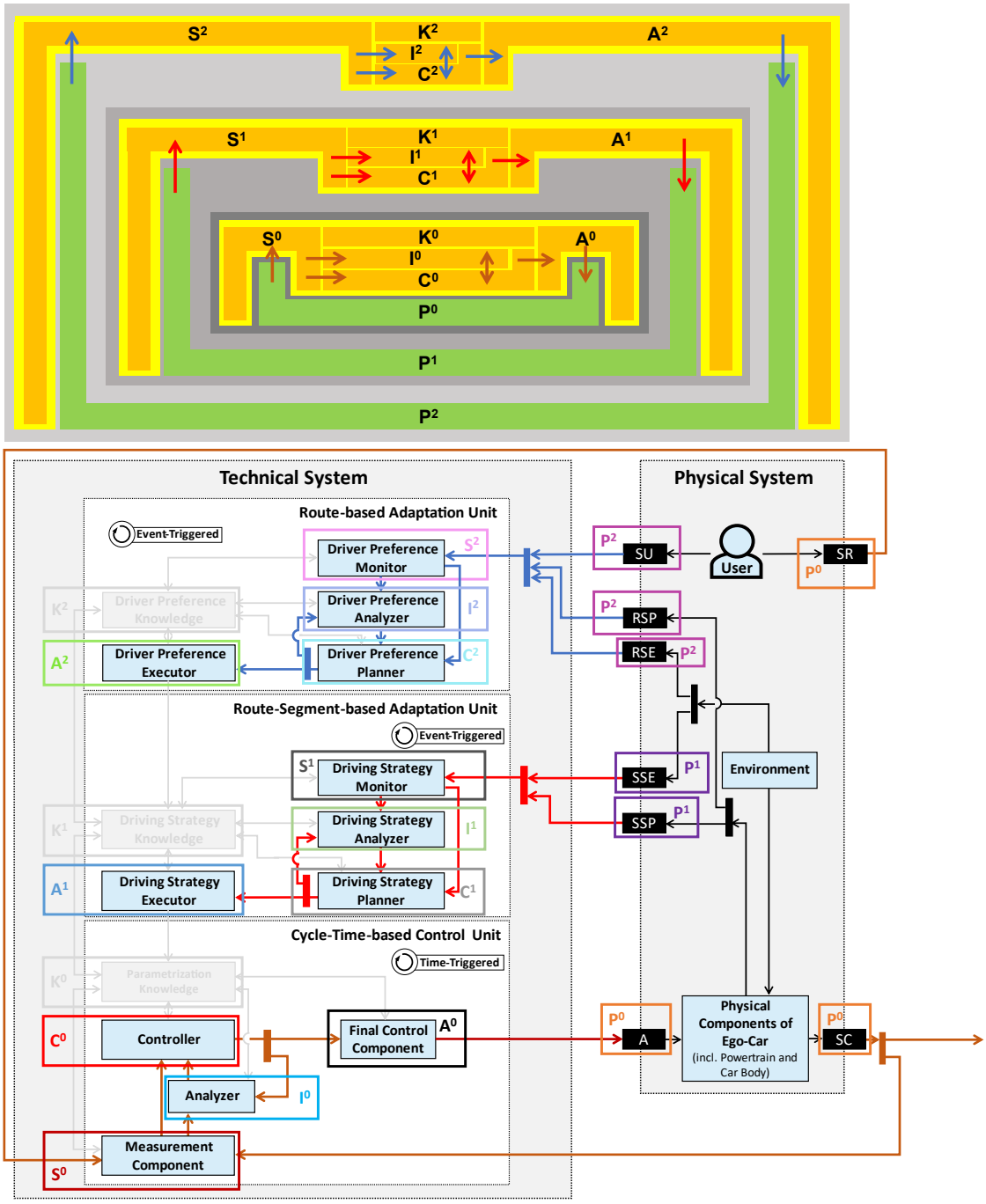


Figure 5.7: Component Interactions of ACCC in UC1

Current ACC variants are designed not only to take over longitudinal control of the car in the case of no preceding vehicle but are also required to work once a preceding vehicle as an obstacle is located in front of the car. The measurement component

provides its symptom to the controller and the analyzer. Thus, the analyzer takes the current headway and current velocity of the ego-car from the symptom as its inputs, which the analyzer uses to estimate the current velocity of the preceding vehicle.

Generally, the analyzer ("I^0") works similarly to the interpreting components ("I^2" and "I^1") on other layers, aiming to identify the current context situation by comparing the known symptoms included in its configuration with the newly observed facts included in the new symptom, which is provided by the measurement component. The only difference in the analyzer compared to the interpreting components on other layers is that the interpreting components, in the case of fulfilled requirements, would trigger the other components to continue further computation processes on the closed control loop to realize the adaptation control. Thus, these interpreting components are essential in participating in the control loop.

Although the analyzer ("I^0") is defined as an optional component in the fundamental component structure of "SICAP-K", the design of the cycle-time-based control unit in this dissertation still considers the analyzer in the architecture of the implemented ACCC prototype. As explained in Section 4.2.2, the analyzer is responsible for evaluating the physical system's current and future context since it plays the role of the interpreting component ("I^0") on the lowest layer. It sends its analysis results to the controller. If a preceding vehicle is detected (still existing in the ACCC's memory, cf. Section 5.2.3), the results include a predictive velocity trajectory of the preceding vehicle with a limited time horizon. Otherwise, there is no such trajectory. The predictive velocity trajectory is then forwarded to the controller as the analyzer's output. Thus, controller takes this trajectory as future environmental disturbances during its decision-making of control activity. In addition, it knows the current control task is to maintain the set value of the cruise velocity or headway based on the states of the variables related to the previously mentioned trajectory.

5.3.2 Dynamic Behaviors of ACCC in UC2

The second use case generally deals with the initialization, retrieval, and updating of domain knowledge about the physical system behaviors on a single layer, as presented in Section 4.2.6.2. In the designed ACCC prototype, UC2 exists on each layer, which

means that all the route-based and the route-based adaptation unit and the cycle-time-based control unit are involved.

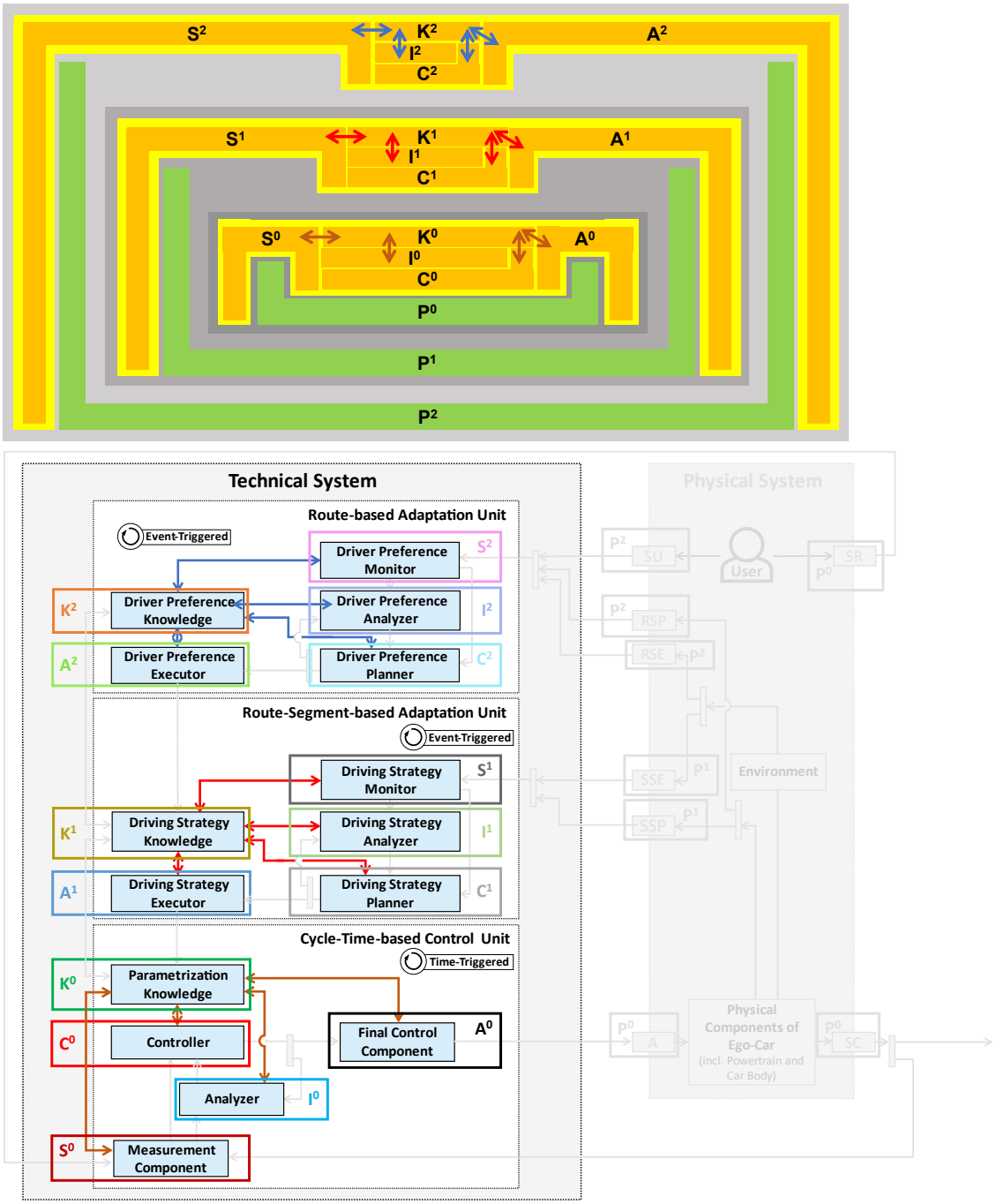


Figure 5.8: Component Interactions of ACCC in UC2

The components that participate in the system's dynamic behaviors in UC2 are colored in Figure 5.8. The other components are marked grey. On each layer, interactions are performed between the knowledge component ("K" in the "SICAP-K" structure) and the other four components ("SICA" in the "SICAP-K") on the same layer. As discussed in Section 4.2.6.2, all interactions (such as knowledge initialization, retrieval, and updating) are designed as knowledge- or data-driven processes. Once a component (either the knowledge component or one of the other components) has a change of knowledge, it disseminates the change and the updated knowledge to other corresponding components. Thus, the knowledge within the knowledge component and other components can always be synchronized. The following subsections will introduce concrete domain knowledge of the components in different subsystems and their detailed process flows.

5.3.2.1 UC2 in the Route-Based Adaptation Unit

As presented earlier, the driver preference knowledge component ("K^2") serves as a central knowledge repository on its corresponding layer. While implementing this component, development engineers must manually initialize some domain knowledge for the system's first operation. In the process of knowledge initialization, the driver preference knowledge component initializes other components by pushing its included domain knowledge to them. Thus, this means that the driver preference knowledge component is also aware of the system's internal knowledge, such as topology and the mapping relationships between the components and their corresponding required domain knowledge. The same pushing process for updating domain knowledge also happens if there is a domain knowledge change in the driver preference knowledge component, for example, when the driver preference knowledge component has acquired knowledge support from knowledge components on other layers (UC4).

From the opposite direction, the other components ("S^2", "I^2", "C^2", "A^2") are also allowed to push their newly observed facts and corresponding extracted domain knowledge into the driver preference knowledge component. Such a process happens once these components detect an inconsistency between their currently available domain knowledge and newly observed facts during manual control by the driver.

For example, the component driver preference monitor can update the map knowledge about the intermediate GPS points for route segmentation once a lane changes due to a reconstruction site. As another example, the driver preference planner can also update its knowledge of the decision tree once the observed data lies in the extrapolation value range due to a time-variant change of the driver's preferences. In another case, the knowledge about the interpretation mechanism (the cost function) for identifying qualities of different candidate strategies can also be updated once the driver preference planner has detected an inconsistency between the expected and realistic effect of the planned strategy.

Generally, it can be understood that processes in UC2 are directly driven by the input of sensory data or knowledge, which can be triggered either by the driver preference knowledge component or the other components. In the case of a detected inconsistency between the available domain knowledge and the newly observed facts, a process for knowledge synchronization will be immediately activated. After the knowledge initialization for the first system operation, the later process of knowledge synchronization can be understood as a process of component adaptation regarding domain knowledge.

5.3.2.2 UC2 in the Route-Segment-Based Adaptation Unit

The knowledge discussed above is initialized manually by development engineers in the driving strategy knowledge ("K^1") component that plays the role of a central knowledge repository on the middle layer. Subsequently, the driving strategy knowledge component pushes its available domain knowledge to other components on the same layer, aiming to initialize them for the first operation of the ACCC. The same process of pushing knowledge also happens once the domain knowledge in the driving strategy knowledge component or other components has been changed, which thus can be understood as a process of knowledge synchronization driven by sensory data or knowledge input.

For example, the route-based adaptation unit provides the planned set travel profile to the route-segment-based adaptation unit, as presented earlier. Such a process is interpreted as a change of domain knowledge in the driving strategy knowledge component. In this case, the driving strategy knowledge component pushes the updated

knowledge to the driving strategy planner. Thus, the driving strategy planner can take the set travel profile as a new specification during the planning of the middle-level driving strategy.

Considering the ACCC's logical system architecture (cf. Figure 5.8), there is no significant difference of dynamic behaviors within UC2 between components in the route-based adaptation unit and the route-segment-based adaptation unit. Once any component has detected an inconsistency between newly observed facts and its available knowledge, it triggers a new process of knowledge synchronization by pushing the new knowledge extracted from the new facts to other corresponding components.

5.3.2.3 UC2 in the Cycle-Time-Based Control Unit

In the case of the cycle-time-based control unit, UC2 refers to the interaction between the parametrization knowledge component ("K^0") and the other four components ("S^0," "I^0", "C^0", "A^0"). All components except the knowledge component require relevant domain knowledge to complete their tasks. More details about the required knowledge have been illustrated in Figure 5.6 (cf. Section 5.2.4).

For example, the measurement component generates the symptom defined as well-structured data conformed to a specific meta specification. In this case, the meta specification represents a kind of component configuration and is seen as domain knowledge. In another case, the controller in the evaluation example of this dissertation (cf. Section 5.5) is implemented using PID control. Thus, another typical example of domain knowledge is the parametrization of P-, I- and D-variables regarding their values. A similar case exists for the analyzer and the final control component, which rely on their corresponding internal processing mechanisms and the knowledge of parametrizations to process the received inputs and generate the expected outputs.

As with both adaptation units, development engineers must initialize the domain knowledge during the development of the parametrization knowledge component that serves as a central knowledge repository in the cycle-time-based control unit. The same synchronization process of pushing new knowledge between the parametrization knowledge ("K^0") and the other four components ("S^0," "I^0", "C^0", "A^0") in the control loop also exists, triggered by the component that first discovers an inconsistency between

available knowledge and newly observed facts. Although the ACCC's design supports the knowledge synchronization between different components, this dissertation only considers the process between the parametrization knowledge component and the controller. Once the location-based set trajectory of cruise velocity and headway as the middle-level driving strategy is provided by the route-segment-based adaptation unit to the parametrization knowledge component, it pushes the set trajectory to the controller. Other processes in UC2, like knowledge retrieval and knowledge update are excluded in the implemented ACCC example to reduce implementation complexity.

5.3.3 Dynamic Behaviors of ACCC in UC3

The third use case deals with adaptation control across multiple networked nodes that are instantiated as hierarchical layers in the system architecture of ACCC, as introduced in Section 4.2.6.3. As discussed earlier, the designed ACCC prototype has a three-layered architecture, as shown in Figure 5.9.

In the figure, the components involved in UC3 are colored, and the components not participating in UC3 are marked grey. Thus, the ACCC has two communication paths for adaptation control in UC3 (cf. Figure 5.9, marked with blue and red arrows). The first one exists between (1) the route-based adaptation unit (on the highest layer) and the route-segment-based adaptation unit (on the middle layer). The second adaptation control path happens between (2) the route-segment-based adaptation unit (on the middle layer) and the cycle-time-based control unit (on the lowest layer). These two communication paths will be introduced in the following subsections.

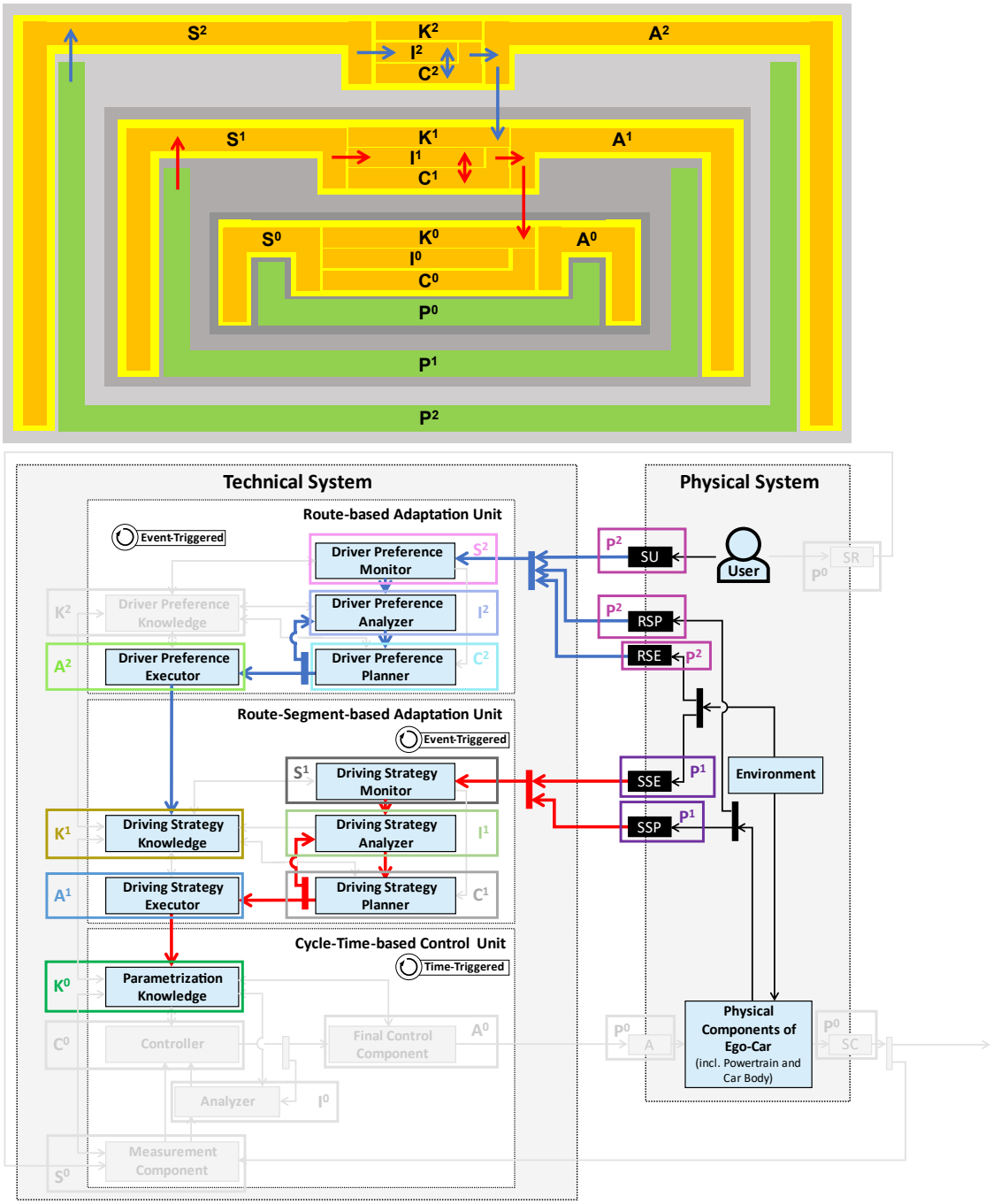


Figure 5.9: Component Interactions of ACCC in UC3

5.3.3.1 Adaptation Control Across the Highest and Middle Layers

As presented earlier, the route-based adaptation unit aims to plan a personalized route-based strategy. This personalized route-based strategy includes set travel profiles such

as accumulated travel time, accumulated energy consumption, and average acceleration[13] of the vehicle, from the origin until the end of each route segment and the destination, aiming to fulfill the preferences of a single individual driver. In this case, route segmentation is realized using a sequence of predefined geographical points representing significant intersections on the route. After determining the strategy, the route-based adaptation unit transfers the set travel profiles to the route-segment-based adaptation unit. In this case, the route-based adaptation unit plays the role of an adaptation manager to adapt its managed system, which is the route-segment-based adaptation unit.

The components that participated in the process of adaptation control between the route-based and the route-segment-based adaptation unit are colored and connected with blue arrows in Figure 5.9. In this process, the *Driver Preference Monitor*, *Driver Preference Analyzer* ("I^{2}") and *Driver Preference Planner* ("C^{2}") complete their tasks of symptom generation, decision making for the request of strategy planning and planning of the optimized high-level route-based strategy, as introduced at the beginning of Section 5.2.2. Compared to UC1, the only behavioral difference of the route-based adaptation unit in UC3 is that the planned route-based strategy by the *Driver Preference Planner* ("C^{2}") is split up into partial strategies for each route segment. Subsequently, the partial strategies are forwarded to the route-segment-base adaptation unit on a lower layer instead of directly manipulating the actuators for interaction with the physical system.

In the route-segment-based adaptation unit, the *Driving Strategy Knowledge* ("K^{1}") component receives the input of split individual strategies to replace the previously saved strategies in its knowledge repository. As presented earlier, these strategies are taken as specifications during the planning of middle-level driving strategy for the route-segment-based adaptation unit. Thus, it can be understood that the domain knowledge in the driving strategy knowledge ("K^{1}") component is adapted by its higher layer,

[13] In this dissertation, driving comfort is naively defined, which is directly represented by the ego-car's longitudinal acceleration. In fact, the driving comfort can be formulized with representative variables by applying different complicated formulas (e.g., based on the vehicle's vibration and acceleration) [132][133].

following the paradigm of adaptation composition in the proposed generic architecture style of this dissertation (cf. Section 4.2.3).

5.3.3.2 Adaptation Control Across the Middle and Lowest Layer

The second adaptation control happens between the route-segment-based adaptation unit and the cycle-time-based control unit. In this case, the route-segment-based adaptation unit is the adaptation manager, and the cycle-time-based control unit is its managed system.

As presented earlier, the route-segment-based adaptation unit as an event-triggered subsystem aims to determine the so-called middle-level driving strategy. The middle-level driving strategy includes a driver-preferred location-based set trajectory of cruise velocity or headway, depending on whether a preceding vehicle is recognized or not. A particular feature of the route-segment-based adaptation unit is that its identification of the preceding vehicle does not purely rely on the low-level sensory input data like in extant ACC variants already presented (cf. Section 3.1–3.4). In addition, it also relies on semantic interpretation on a higher level of abstraction and memory ability.

As illustrated in Figure 5.9 with red arrows, the driving strategy monitor in the route-segment-based adaptation unit firstly preprocesses the raw sensory data to generate the expected symptom. Subsequently, the driving strategy analyzer takes the symptom as its newly observed facts to compare with its known symptoms, based on their relevance aiming to identify the current situation. In the case of a known symptom and thus a successfully identified situation, the Driving Strategy analyzer triggers the driving strategy planner to determine an optimized set trajectory of cruise velocity or headway for the route segment that is current focus.

During the determination process, the provided high-level personalized strategy, including the set travel profiles by the route-based adaptation unit, is taken as a specification and required to be guaranteed as much as possible. Thus, an optimized set trajectory of cruise velocity or headway focusing on the following individual route segment would be planned based on the evaluation criteria and cost function mentioned earlier. The set trajectory consists of a sequence of quantitative values of the cruise velocity or the headway. The values represent the expected set states of the ego-car

when it moves through corresponding geographical route points within the route segment.

Finally, the set trajectory is forwarded again into the driving strategy executor, which decomposes the set trajectory as partial set trajectories consisting of value pairs of the cruise velocity or the headway for each two neighbor route points. The geographical profiles of the route points are also included in the partial set trajectories. Thus, the decomposed partial set trajectories are forwarded to the *Parametrization Knowledge* ("K^0") component in the cycle-time-based control unit to complete the whole adaptation control process. In this case, the partial set trajectories are seen as the domain knowledge of the cycle-time-based control unit adapted by the route-segment-based adaptation unit. Further details about the communication between the participated components in this process will be presented in Section 5.5.

In the ACCC, UC3 refers to a subsystem or component on a higher layer adapting another subsystem or component on a lower layer. Since the cycle-time-based control unit is deployed on the lowest layer of the ACCC's system architecture, it has no lower layer. Thus, UC3 of the cycle-time-based control unit is neglected in the ACCC's design.

5.3.4 Dynamic Behaviors of ACCC in UC4

The final use case refers to knowledge acquisition and sharing across multiple nodes, which happens when the available domain knowledge in a certain node is insufficient to identify and solve the current control problem. As presented earlier, this dissertation instantiates the nodes as hierarchical layers in the ACCC's design.

Since a higher layer has a larger observation scope of context information about the physical system, the ACCC's higher layer can provide knowledge support to the lower layers. As a system with three-layered architecture, the ACCC's dynamic behaviors in UC4 are described with two sub-cases: (1) knowledge acquisition and sharing across the route-based adaptation unit and the route-segment-based adaptation unit and (2) knowledge acquisition and sharing across the route-segment-based adaptation unit and the cycle-time-based control unit. In Figure 5.10, the components that participate in these sub-cases are colored. The irrelevant components are marked grey. The following subsections will present more detail about the component interactions in the sub-cases.

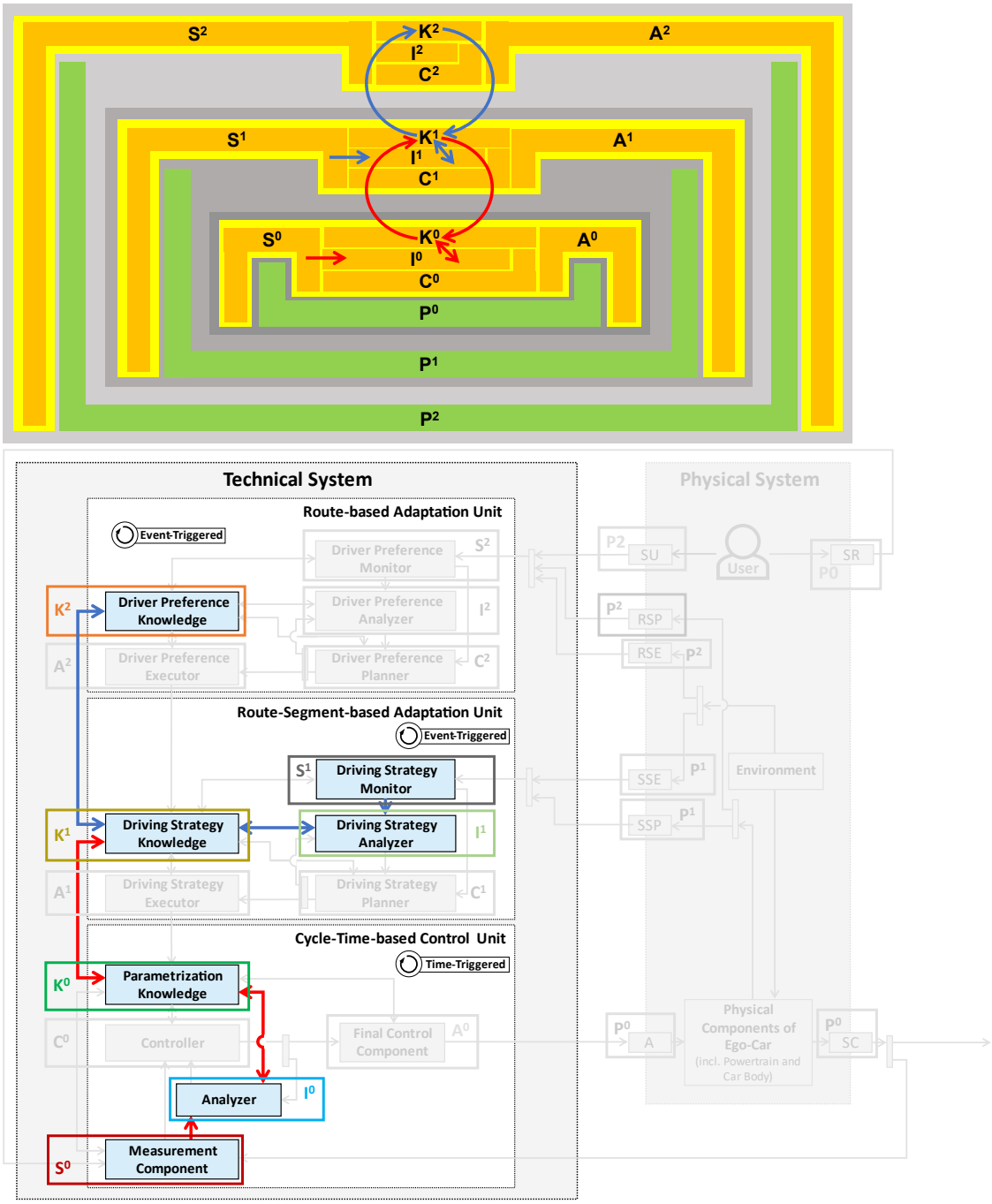


Figure 5.10: Component Interactions of ACCC in UC4

5.3.4.1 Knowledge Acquisition and Sharing Across the Highest and Middle Layers

Knowledge acquisition and sharing across the highest and middle layers of the ACCC means that the route-segment-based adaptation unit requires support from the route-

based adaptation unit. The related communication path between the components is illustrated with blue arrows in Figure 5.10.

For example, the driving strategy analyzer (“I^1”) in the ACCC has identified that its received symptom cannot be matched to any known situation. Thus, the available domain knowledge about the known situations is insufficient, and the current symptom is identified as a novelty. In this case, the driving strategy planner will not be triggered. Instead, the driving strategy analyzer requests the support of the driving strategy knowledge component. As defined in UC2 (cf. Section 5.2.3.2), the domain knowledge between the driving strategy knowledge (“K^1”) component and the other four components (“S^1”, “I^1”, “C^1”, “A^1”) on the same layer is always synchronized. Thus, this means that the driving strategy knowledge component also does not include the required domain knowledge. However, it plays its designed role of an interface component to communicate with other knowledge components deployed on other layers or even in other cars and request their support.

A typical example is as follows. Due to construction work on the road, the route segmentation may have some changes and thus lead to several new route segments, which are still unknown for the route-segment-based adaptation unit. In the case of such unknown route segments, the driving strategy knowledge component may request the support of the driving preference knowledge component in the route-based adaptation unit. Since the driving preference knowledge component focuses on the complete route instead of a single individual route segment, it may have the required domain knowledge about the profiles of these new route segments when it receives the information of route planning (e.g., from the navigation system). Thus, the driving preference knowledge component can share its knowledge with the driving strategy knowledge component.

In addition to the knowledge acquisition and sharing across the layers deployed on the same car, the driving strategy knowledge component may also request support from the route-based adaptation unit deployed on other surrounding cars by relying on the Car2X-communication capabilities. Thus, a “transfer learning” process can be realized between ACCCs on different cars. Considering a more general viewpoint, so-called fleet-based learning of driving strategy on the same route, which may benefit traffic management, would thus be realized. The developed system architecture style theoretically allows

such fleet-based knowledge acquisition and sharing. However, since this dissertation mainly focuses on the system design of ACCC on a single individual car, this cooperative knowledge acquisition and sharing across multiple cars is excluded.

5.3.4.2 Knowledge Acquisition and Sharing Across the Middle and Lowest Layers

Knowledge acquisition and sharing across the middle and lowest layers in the ACCC means that the cycle-time-based control unit requires support from the route-segment-based adaptation unit. The related communication path between the components is illustrated with red arrows in Figure 5.10.

In the case of the cycle-time-based control unit, UC4 happens once the currently available knowledge is insufficient to find a solution and thus needs to request knowledge support from other knowledge components located on other layers or domains. Such a case may happen if the analyzer lacks knowledge (e.g., if current values of variables included in the newly observed facts about the preceding vehicle's behavior included in the symptom are out of the analyzer's known value range). Thus, the analyzer would communicate with the parametrization knowledge component to eliminate the knowledge lack. The parametrization knowledge component then communicates with the driving strategy knowledge component on the higher layer to request further support.

Since the route-segment-based adaptation unit has a memory ability and can learn the driving preferences of obstacles ahead during previous trips, it may have "seen" much more preceding obstacles than the cycle-time-based control unit. Thus, the driving strategy knowledge component in the route-segment-based adaptation unit could provide the parametrization knowledge component its required knowledge. In the implemented ACCC example (cf. Section 5.5), UC4 is excluded to reduce implementation complexity, although the system's architecture design allows such a component interaction.

5.4 Applying Communication Architecture Patterns for Component Interactions in Artificial Cognitive Cruise Control

In the previous section, an architecture of artificial cognitive cruise control (ACCC) that illustrates the system's static construction is instantiated from the generic architecture

style. As introduced in Section 4.2, another important perspective to illustrate the system is the view of dynamic system behaviors, from which four use cases (UC1–UC4) were previously defined. As introduced in Section 4.2.5, generic communication architecture patterns can be applied to specify the interactions between components and thus to describe dynamic system behaviors.

In this dissertation, it is emphasized that the application of the mentioned generic communication patterns within the use cases is unlimited. It means either one communication pattern or different patterns can be applied within one use case. Since there is no preference for certain generic communication patterns, several example patterns are selected in this section to apply within the four predefined use cases. All these examples will be introduced in more detail in the following sections in order to derive a more concrete understanding of the design of dynamic system behaviors.

5.4.1 Publish-Subscribe Pattern for UC1

The first use case refers to technical process control on a single layer, which describes the technical process flow of independent control loops on each layer of the hierarchical architecture, as illustrated in Figure 5.11. On the top of Figure 5.11, a logical architecture from the view of software engineering is instantiated based on the generic architecture style. The fundamental component structure of “SICAP-K” is deployed on each layer in the logical architecture. The process flow of UC1 is illustrated with arrows, with three colors (blue, red, and brown) to identify different layers.

In Figure 5.11, it can be seen that all sensors (*SC, SU, SR, SSP, SSE, RSP,* and *RSE*) play the roles of publishers since they are responsible for data delivery. Unlike the sensors, the actuator (*A*) is a pure subscriber who only receives the message from the final control component in the technical system. Except for the sensors and actuators, all other components simultaneously play the roles of publishers and subscribers, depending on the context of the concrete communication path.

For example, the driver preference monitor generates the symptom and forwards the symptom to the driver preference analyzer and the driver preference planner. In this case, the driver preference monitor is the publisher, and the other components are the subscribers. On the middle layer, the driving strategy planner publishes the message

about middle-level driving strategy, to which the driving strategy executor will then subscribe.

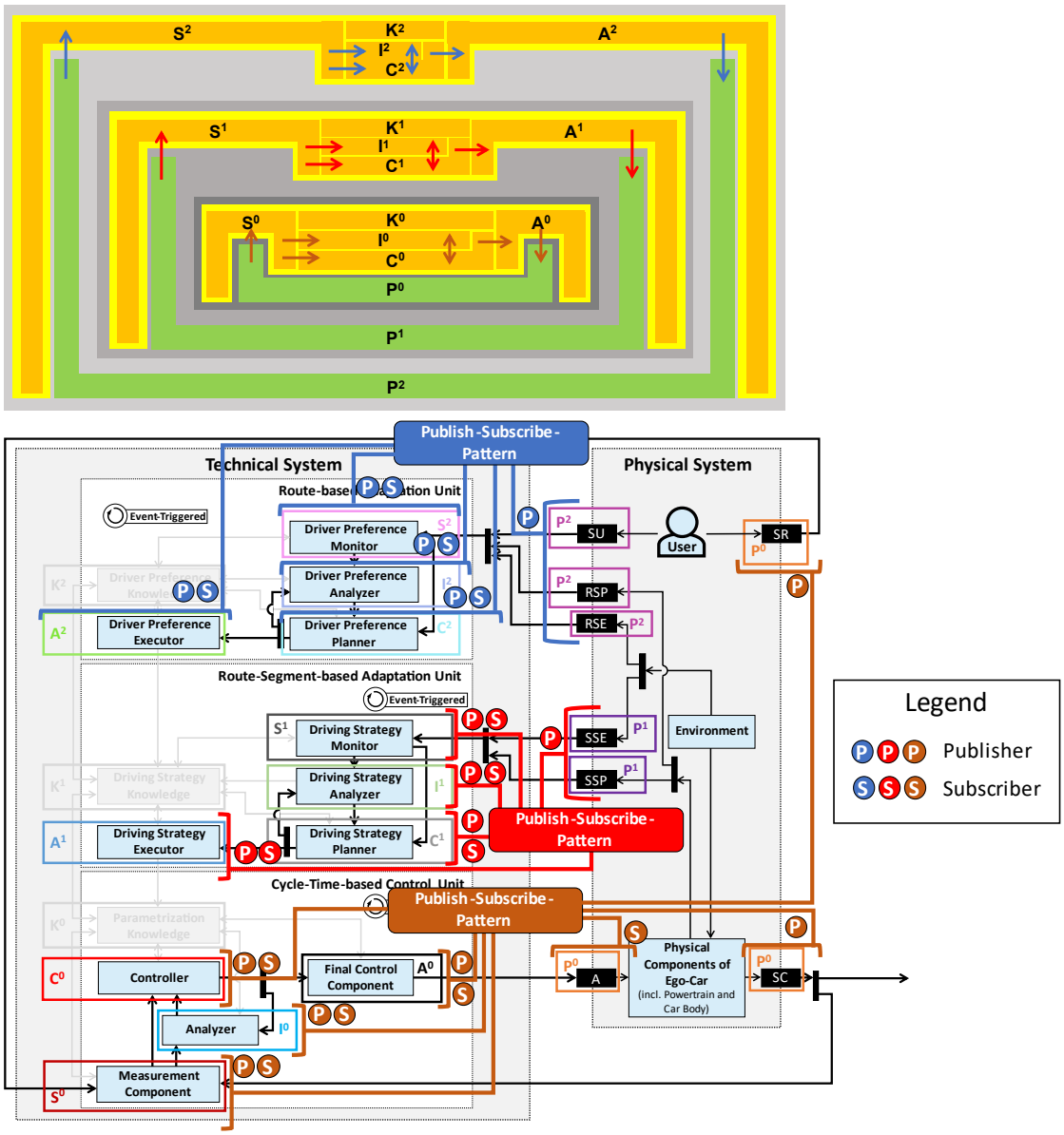


Figure 5.11: Component Roles within Interactions of UC1

All publishers and subscribers must register with a so-called change propagation infrastructure in the publish-subscribe pattern. Thus, the change propagation infrastructure can route the messages from publishers to interested subscribers [58]. As discussed earlier, the ACCC can also be designed as a distributed instead of a

centralized system. Thus, the route-based adaptation unit and the route-segment-based adaptation unit are not limited to being deployed on a single car. They can also be deployed on different domains. Additionally, the ACCC relies on the data delivery of offboard sensors, which are also not deployed on the local domain of car. In this case, the application of publish-subscribe-pattern guarantees great system scalability for the artificial cognitive cruise control.

5.4.2 Shared-Repository Pattern for UC2

The second use case (UC2) describes the process of knowledge initialization, retrieval, and updating on a single layer of the ACCC's hierarchical architecture, which refers to the interaction between the component "K*" and the other components ("S*", "I*", "C*", and "A*") on the corresponding layer, as illustrated at the top of Figure 5.12. Based on the previous introduction to UC2 in Section 4.2.5, it is known that the component "K*" plays the role of a knowledge repository to provide the domain knowledge about the physical system to other components. From the opposite direction, the other components can also update their learned knowledge into the component "K*", which means that the component "K*" is accessible for the other components.

The shared-repository pattern is selected as an example to be applied within UC2, as illustrated at the bottom of Figure 5.12. Thus, the components of driver preference knowledge, driving strategy knowledge, and parametrization knowledge are defined as shared repositories on different hierarchical architecture layers.

In addition to the shared repository, other components in the technical system play the roles of application components. For example, the driver preference monitor and the driver preference analyzer are the application components, which can access the driver preference knowledge as the central shared repository. From the opposite direction, once the driver preference knowledge triggers any update, it can also disseminate knowledge to the driver preference monitor and the driver preference analyzer, which play the roles of application components, relying on the notification mechanism (cf. Section 2.2.7).

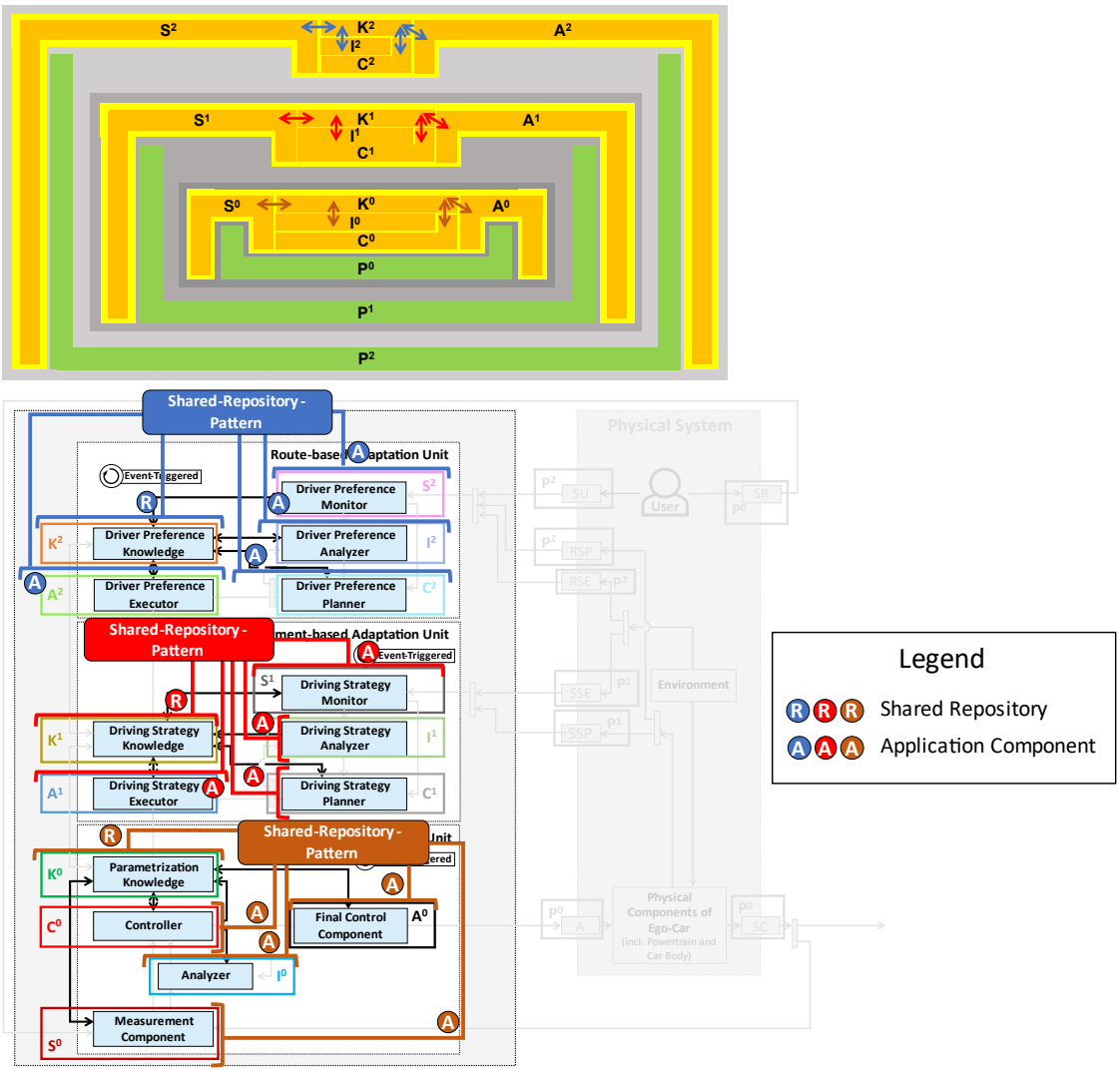


Figure 5.12: Component Roles within Interactions of UC2

5.4.3 Request-Response Pattern for UC3

UC3 refers to the hierarchical control flow across layers, which focuses on how the subsystem on a higher layer determines a high-level strategy to adapt a subsystem deployed on a lower layer, following the introduced paradigm of adaptation composition, as introduced in Section 4.2.3. Detailed communication paths between the components within UC3 have been illustrated with arrows at the top of Figure 5.13.

To specify the communication paradigms within UC3, two communication patterns, including the publish-subscribe pattern and the request-response pattern, are selected

as examples. As shown at the bottom of Figure 5.13, the communication between sensors and subsystems in the technical system relies on the publish-subscribe pattern. In this case, the sensors are the publishers, and the components in the subsystems like the driver preference monitor and the driving strategy monitor are the subscribers. The publish-subscribe pattern realizes a loose coupling between the sensors and components for monitoring sensory data, relying on high system scalability. In this case, the sensors as publishers and the monitoring components as subscribers work completely independently, and the specified topics of the messages realize all communication identification. Thus, further development of the whole system becomes much easier (e.g., integrating more sensor accesses for the technical system).

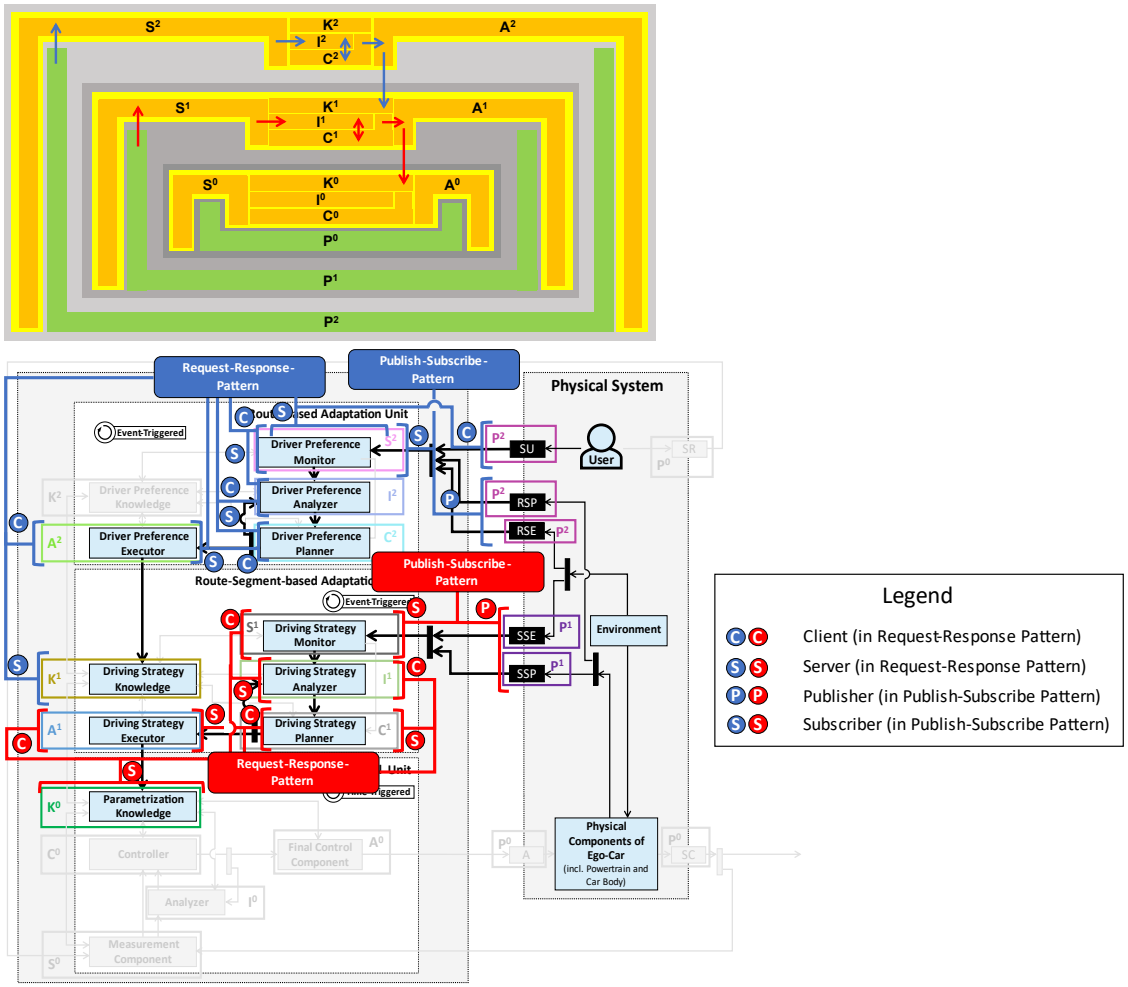


Figure 5.13: Component Roles within Interactions of UC3

In addition to the publish-subscribe pattern, the request-response pattern (also called the client-server pattern) is also selected to specify communication paths between components inside the technical system. For example, on the highest layer of the hierarchical architecture, the driver preference monitor provides its generated symptom to the driver preference analyzer. In this case, the driver preference monitor as the client initiates the interaction with the driver preference analyzer, which plays the role of a server. In this case, the client component invokes the services provided by the server component (e.g., which also happens between the driver preference executor in the route-based adaptation unit and the driving strategy knowledge component of the route-segment-based adaptation unit).

As introduced in Section 2.3.1, the communication paradigm behind the request-response pattern (also called the client-server pattern) is the remote procedure call (RPC), which can still be categorized as synchronized and asynchronized. In the ACCC, it is recommended to use the asynchronized RPC due to the feature of a distributed system, especially when the route-based adaptation unit and the route-segment-based adaptation unit are deployed on two different domains. In this case, the component of driving preference executor can keep working in parallel if there is no reply from the driving strategy knowledge component in the route-segment-based adaptation unit.

5.4.4 Blackboard Pattern for UC4

The final use case (UC4) refers to hierarchical knowledge acquisition and sharing across layers in the ACCC, which happens when the interpreting component on a certain layer such as “I^1” cannot identify the problem due to a lack of knowledge. In this case, the interpreting component “I^1” will communicate with the knowledge component “K^1” on the same layer to request support. Since knowledge between the “I^1” and “K^1” is always synchronized, the “K^1” cannot directly support the “I^1”. However, it can communicate knowledge components on neighbor layers (“K^2” and “K^0”) of the hierarchical architecture, as illustrated at the top of Figure 5.14.

To specify the component interactions within UC4, two different communication patterns, including the shared-repository-pattern and the blackboard pattern, are selected as examples in this section. As presented in UC2, the communication between the

knowledge component and the other component like the sensing, interpreting, control, and actuating component on each layer can be realized based on the shared-repository pattern. The interpreting and the knowledge component ("I^1" and "K^1") also participate in UC4. Thus, the shared-repository pattern is applied to specify their communication.

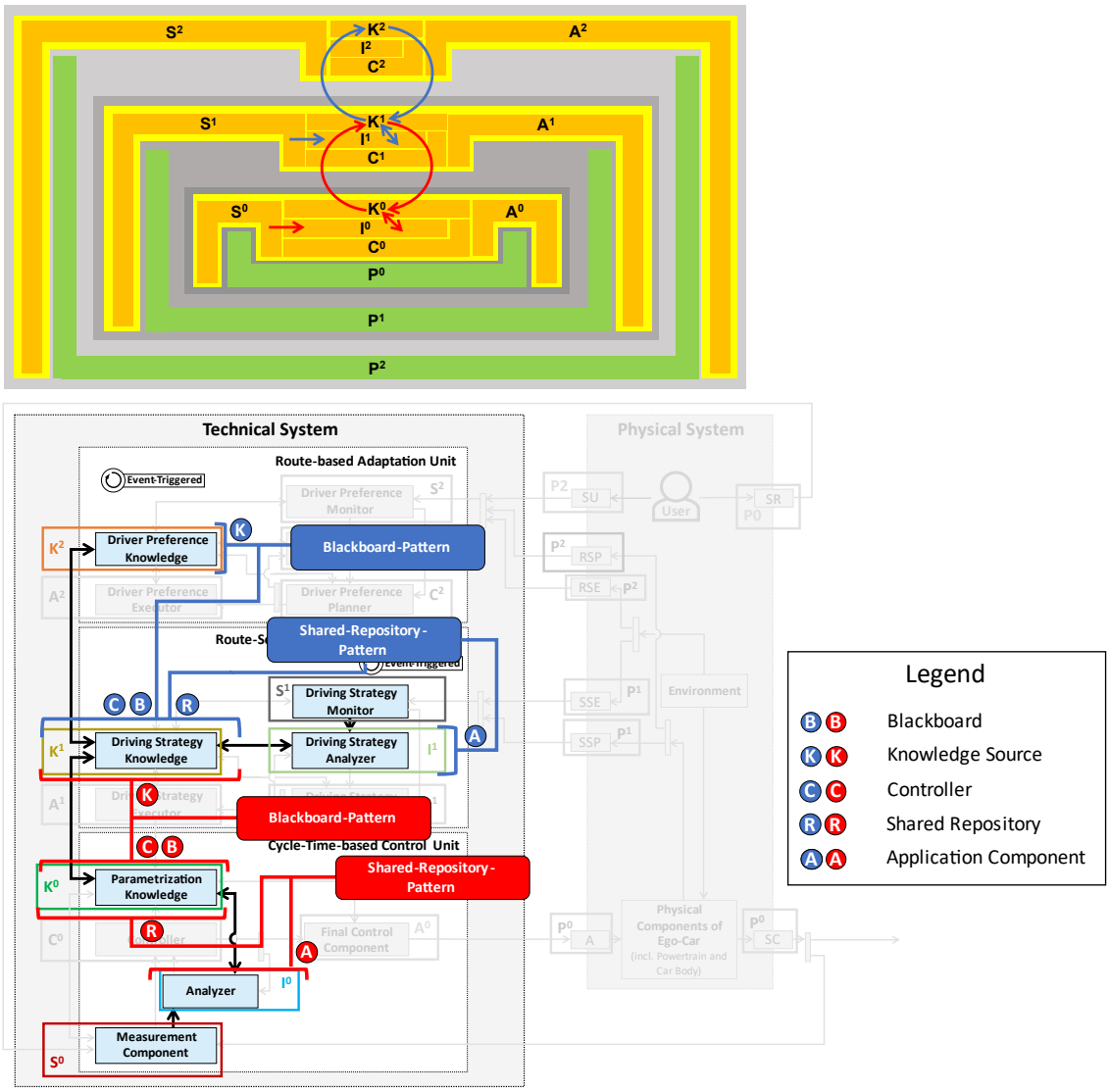


Figure 5.14: Component Roles within Interactions of UC4

In UC4, it can be understood that the subsystem on one layer of the hierarchical architecture with its knowledge base is required to solve a task based on incomplete knowledge and data and thus requires the support of knowledge bases on neighbor layers. Such a feature corresponds exactly to the application potential of the blackboard

pattern. Thus, the blackboard pattern is selected to specify communication of the knowledge components ("K^0" vs. "K^1" or "K^1" vs. "K^2") on each two hierarchical neighbor layers. In this case, the knowledge component requesting support plays the role of controller and blackboard. The knowledge component providing the support plays the role of knowledge source, which the controller activates and operates on the blackboard to contribute its knowledge (cf. Section 2.3.5).

As shown at the bottom of Figure 5.14, the driving strategy knowledge component works as the blackboard and controller in the case of communication with the driver preference knowledge component on the highest layer. Additionally, it also plays the role of knowledge source while communicating with the parametrization knowledge component on the lowest layer. Following this idea, the knowledge component may be further decomposed as two subcomponents, which are responsible for a pure knowledge database and its corresponding management mechanisms, respectively. Due to the limited scope of this dissertation, further details about this idea will be discussed in Section 6.3.

5.5 Implementation of Artificial Cognitive Cruise Control

After introducing the system design, a prototype implementation is planned in this dissertation in order to evaluate the ACCC's performance. The implementation strictly follows the designed architecture of the ACCC, consisting of the technical and physical system. In the technical system, the subsystems, namely the route-based adaptation unit (RAU), the route-segment-based adaptation unit (RSAU), and the cycle-time-based control unit (CTCU) are also implemented. In the following sections, more details about the implementation will be presented.

5.5.1 Implementation Overview

Figure 5.15 indicates an implementation overview of the ACCC prototype. The technical system in the ACCC, including three subsystems (RAU, RSAU, and CTCU) and their corresponding components, is implemented using Python.

The physical system is implemented in a hybrid manner. The well-known simulator SUMO[14], based on C++, is applied to simulate the ego-car's driving environment. Since SUMO primarily focuses on a macro simulation of traffic flow instead of a detailed simulation of a single car's internal physical processes, the car's realistic driving dynamics model is missing. Thus, an additional Python implementation of the ego-car's physical components for simulating driving dynamics was also completed. In addition, the human driver in the physical system is modeled based on the real driving data with Python.

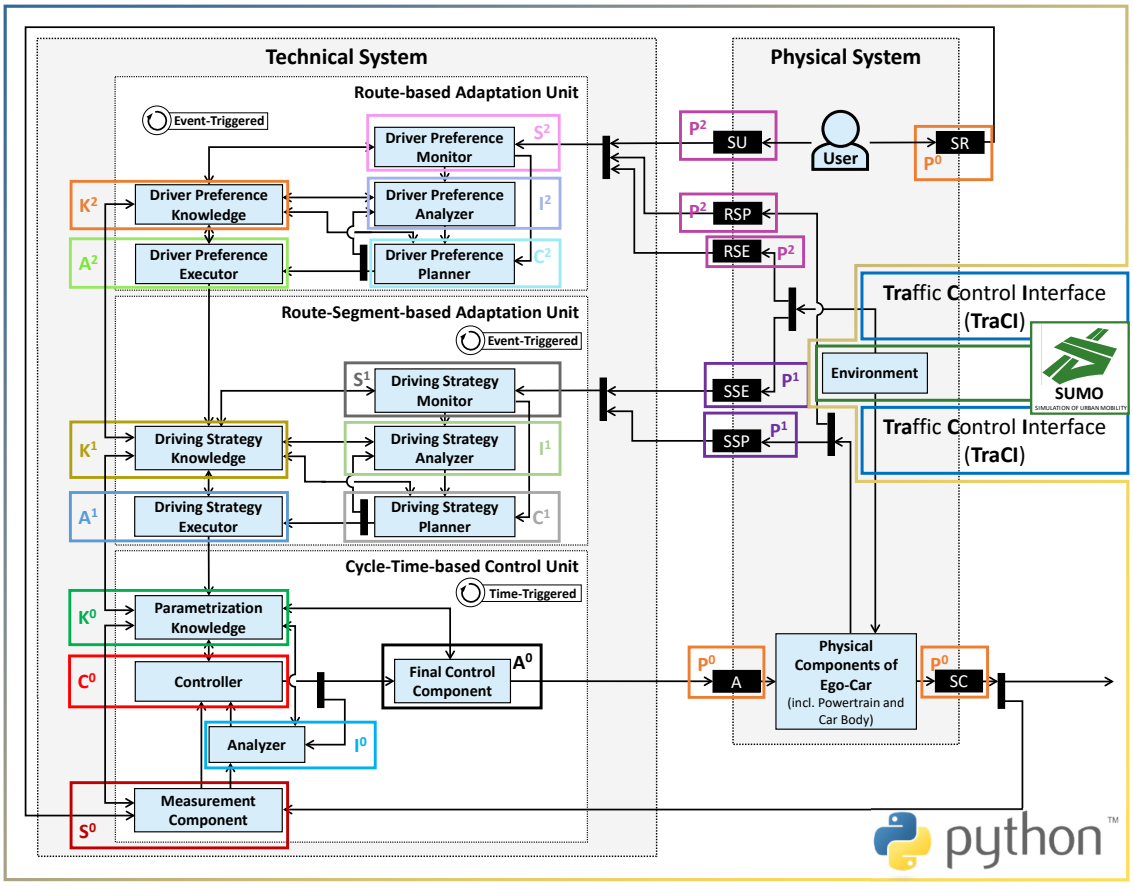


Figure 5.15: Overview of the Co-Simulation Platform in the Implementation of ACCC Prototype

[14] SUMO (**S**imulation of **U**rban **Mo**bility) is an Eclipse Foundation project and stands for an open-source traffic simulation platform developed by the German Aerospace Center and its community users. Link: https://www.eclipse.org/sumo/ (accessed on 26th Apr. 2022).

After implementing the technical and physical systems, they are integrated to build a co-simulation platform to evaluate the ACCC's performance. The SUMO and Python simulation communicate via a traffic control interface (TraCI[15]). In Figure 5.15, the actuators (component *A*, cf. Figure 5.15) and the sensors (components: *SC*, *SU, RSP, RSE, SSE,* and *SSP,* cf. Figure 5.15) are designed as the interfaces between the physical and technical systems. These actuators and sensors are not explicitly implemented as additional software components in the implemented co-simulation. Instead, they are implicitly represented by methods calls via TraCI. More concrete implementation details of the components will be introduced in the following subsections.

5.5.2 Implemented Physical System

In the built co-simulation platform, the implemented physical system covers the simulation of the human driver's activities, physical components, and driving dynamics of the ego-car and its interaction with the surrounding driving environment. These simulated processes will be introduced in more detail in the following subsections.

5.5.2.1 Driver

The first *Driver* component implements a driver model to simulate the human driver's driving activities using the ACCC. As illustrated in Figure 5.3 (cf. Section 5.2.1), the human driver is not required to control the ego-car's longitudinal movement. Instead, they need to specify a reference configuration of their preferences, including different weights of optimization criteria for planning the driving strategy, as illustrated in Figure 5.3 within Section 5.2.1. Thus, the ACCC can evaluate the qualities of different candidate high-level set travel profiles and middle-level driving strategies (cf. Figure 5.1) and plan an optimized set travel profile and driving strategy for the driver.

In the original design of the ACCC, the driver can change the values of the initialized criteria weights while the ego-car is driving so that the ACCC can adjust its planned driving strategy. In addition to the manual adjustment, the criteria weights would also be automatically adjusted by machine learning algorithms based on the observed manual

[15] TraCI is an API provided by SUMO-community to access its simulated objects like cars and pedestrians and manipulate their behaviors. Link: https://sumo.dlr.de/docs/TraCI.html (accessed on 28th Apr. 2022).

driving behaviors of the human driver. Nevertheless, in the implemented co-simulation for the ACCC's evaluation, the initialized weights are kept the same in the evaluation of this dissertation to reduce problem complexity. The criteria and their weights are defined and initialized as follows:

Criteria	Meaning	Weights	Symbol
Travel Time	Accumulated travel time of the ego-car from its current location until the end of each following route segment and the destination	0.4	w_{time}
Energy Consumption	Accumulated energy consumption of the ego-car from its current location until the end of each following route segment and the destination	0.4	$w_{consumption}$
Driving Comfort	Average acceleration of the ego-car from its current location until the end of each following route segment and the destination	0.2	$w_{comfort}$

Table 5.1: Criteria and their Weights for Planning the Set Travel Profile and Middle-level Driving Strategy

In addition to setting preferred criteria weights, the driver may also change their desired velocity and headway by human intervention. Such a case may happen, for example, if the driver is unsatisfied with the ACC and ACCC's automated driving due to a too high deviation between the current and desired profiles of the ego-car's velocity and headway.

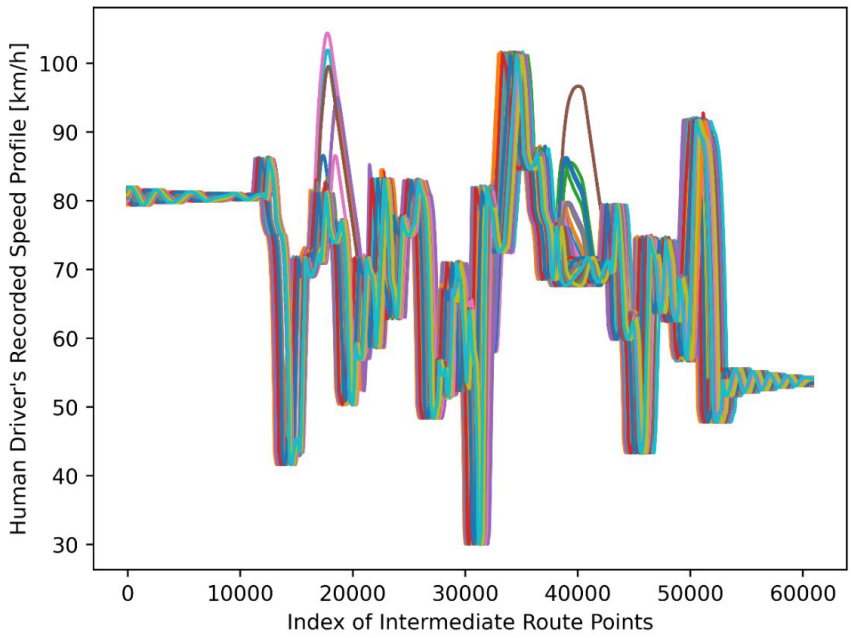


Figure 5.16: Human Driver's Recorded Sample Driving Data with Location-based Speed Profiles

For this purpose, the human driver model requires a reference profile of the driver-preferred cruise speed and headway. In the implemented ACCC, this reference profile is taken as the driver's average driving profile derived from historical trips with manual driving. For this purpose, a pool of recorded real driving data of an anonymous driver is integrated into the implemented human driver model. This data pool includes the driver's driving behavioral data during 600 repeated trips on the federal highway B241 in both directions between Clausthal-Zellerfeld (CLZ) and Goslar (GS) in Germany. As an example, Figure 5.16 shows some sample data describing speed profiles in the recorded trips from Clausthal-Zellerfeld to Goslar.

5.5.2.2 Environment

The implemented *Environment* component in the physical system is responsible for simulating the driving environment. Since the recorded driving data on the federal highway B241 is used in the implemented human driver model, this highway is chosen as the simulated driving environment during implementation. For this purpose, the traffic simulator SUMO is applied to build a realistic simulated environment. Furthermore, the simulation of traffic participants on highway B241 is also included in SUMO to make the simulation more realistic.

During implementation, the geographic route profile of B241 derived from OpenStreetMap[16] (OSM) is integrated into the SUMO simulation. Different cars as the traffic participants and their trips are randomly initialized based on a predefined catalog (including 200 diverse trips with different origins and destinations) to generate an expected nondeterministic capability in the simulation. A certain single car is selected from the catalog as the ego-car. In the SUMO simulation, the ego-car and other cars' movements in the traffic are also visualized. Figure 5.17 compares the simulated track in SUMO and the highway B241 in reality. The ego-car is visualized as the red car with a highlighted green circle in SUMO. The other cars as traffic participants are colored yellow.

[16] OpenStreetMap is a geographic database of the real world that provides diverse free map- and navigation-relared APIs and services. Link: https://www.openstreetmap.org/ (accessed on 28th Apr. 2022).

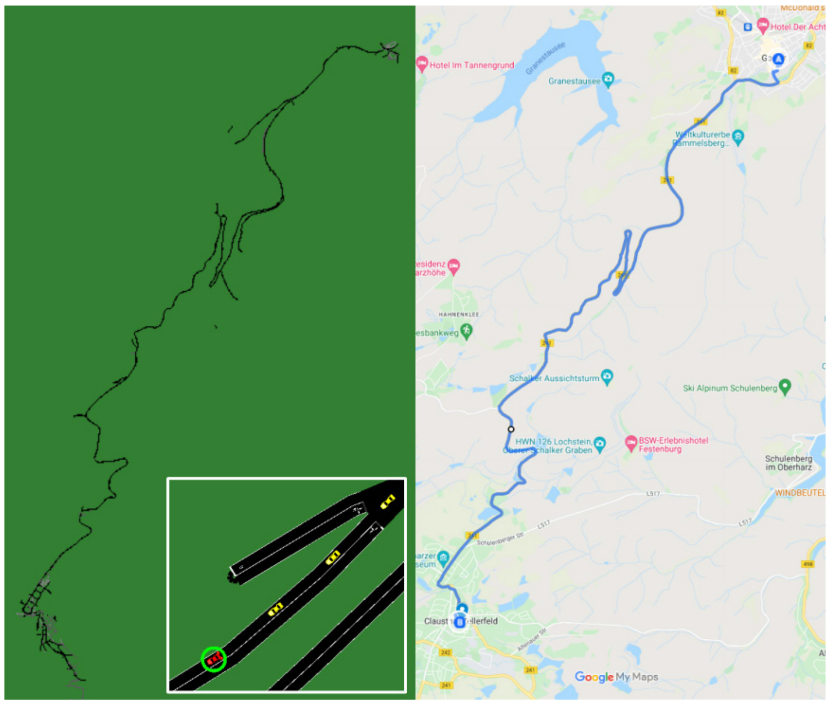


Figure 5.17: SUMO-simulated Track (left) of Germany Federal Highway B241 (right)[17]

5.5.2.3 Physical Components of Ego-car

As introduced previously, SUMO is a macro traffic simulator focusing on overall traffic flow and thus does not have detailed physical modeling for a single car, particularly a model of realistic driving dynamics. For this reason, this dissertation has developed a model of a battery electric vehicle (BEV), including the car's different drive components and driving dynamics, to simulate the ego-car for implementing the component *Physical Components of Ego-Car* (cf. Figure 5.15). The implemented BEV model is developed based on a reference implementation taken from an open-source repository[18] on GitHub.

Figure 5.18 illustrates an overview of this BEV model with a class diagram, in which drive components of the car like the traction battery, electric motor, gear box, front and rear brakes, front and rear wheels, and chassis are included. Several classes with attributes representing the drive components are defined on a meta-level. Due to high

[17] The right-side figure: Direction for Driving from Clausthal-Zellerfeld to Goslar, Germany, Google Maps, 2022, maps.google.com.

[18] Link: https://github.com/bjyurkovich/vehicle-model-python (accessed on 28th Apr. 2022).

implementation complexity, detailed attributes of the classes are not visualized in the class diagram. The defined classes are then instantiated to model the concrete drive components in the simulated BEV.

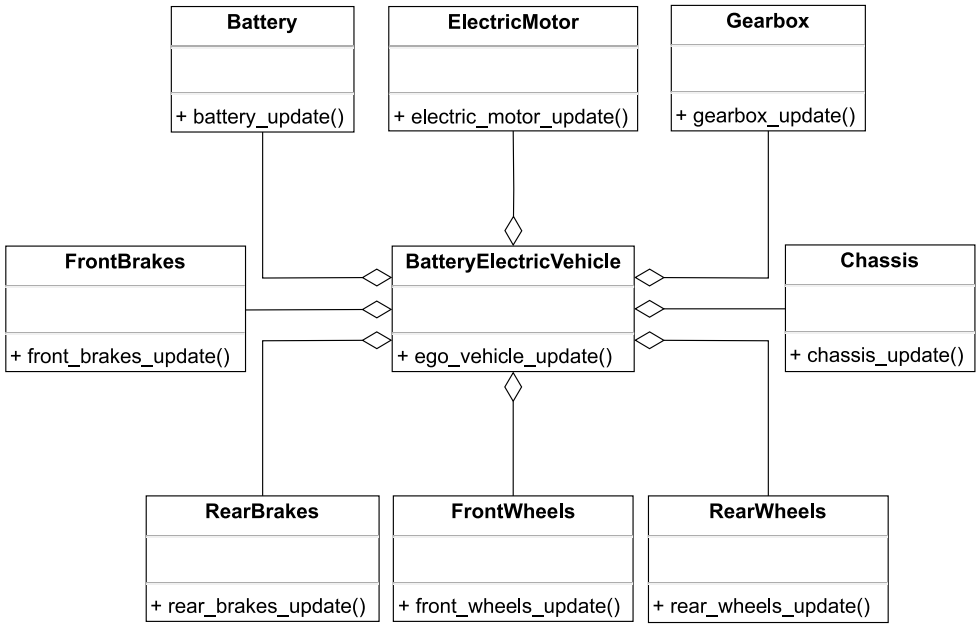


Figure 5.18: Class Diagram for the Construction of BEV Model

In each defined class, methods for updating the drive components' states during the BEV's driving are created. The built BEV model is taken to simulate the ego-car. Thus, the ego-car's overall state can be updated by calling the method *ego_vehicle_update()*, which then calls the previously mentioned methods for updating individual drive components' states like *battery_update()* and *chassis_update()*. In Figure 5.18, only the abstract methods are presented. Detailed input and output parameters are not listed to reduce the visualization complexity. More details about the attributes and the methods can be found in the accompanying code implementation.

Based on the defined classes, the ego-car's driving dynamics can be simulated by calling the methods defined in the classes of drive components. Figure 5.19 shows a process flow of the driving dynamics simulation.

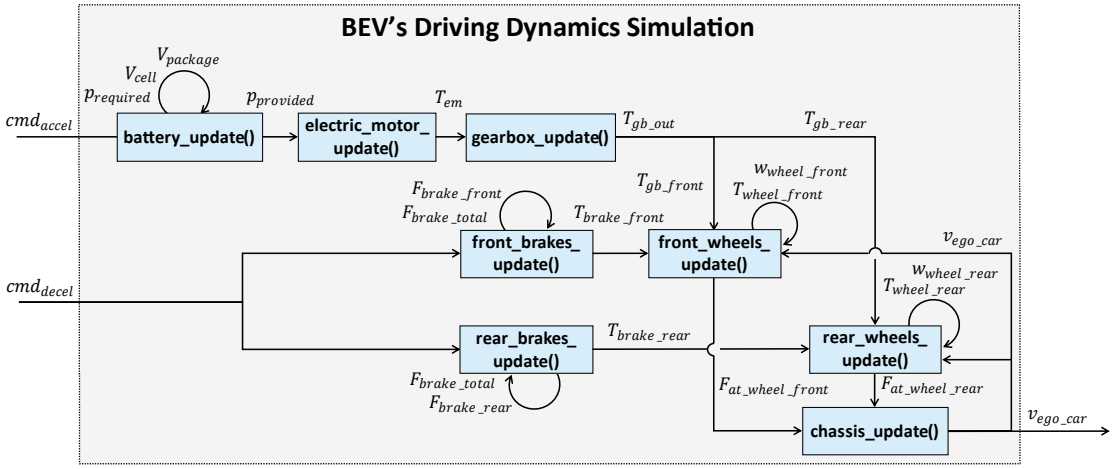


Figure 5.19: Driving Dynamics Simulation in the BEV Model

Since the BEV model is built based on the physical modeling approach, the interactions between the models of the drive components are realized by parameter passing of relevant data values. Table 5.2 provides an overview of these physical parameters.

Parameter	Meaning	Physical Unit
cmd_{accel}	Controller's command for the ego-car's acceleration, $cmd_{accel} \in [0\ 1]$	[-]
cmd_{decel}	Controller's command for the ego-car's acceleration, $cmd_{accel} \in [-1\ 0]$	[-]
$p_{required}$	Required power from the battery for the ego-car's acceleration	[w]
V_{cell} $V_{package}$	Voltage of the cell/package in the battery	[V]
$p_{provided}$	Provided power by battery for the ego-car's acceleration	[m/s^2]
T_{em}	Torque of electric motor	[Nm]
T_{gb_out}	Output torque of gearbox after transmission	[Nm]
T_{gb_front} T_{gb_rear}	Distributed torque of the gearbox to front and rear axis	[Nm]
T_{wheel_front} T_{wheel_rear}	Drive torque of front/rear wheels	[Nm]
w_{wheel_front} w_{wheel_rear}	Angular velocity of front/rear wheels	[rad/s]
F_{brake_total}	Total brake force	[N]
F_{brake_front} F_{brake_rear}	Distributed brake force to front/rear axis	[N]

$F_{at\_wheel\_front}$ $F_{at\_wheel\_rear}$	Drive force at front/rear wheels	[N]
v_{ego_car}	Ego-car's velocity	[m/s]

Table 5.2: Table of Physical Parameters in the BEV Model

In the BEV model, the traction battery is simulated using a well-known simulation approach with a so-called second-order equivalent circuit model (with two RC elements) [120], as illustrated in Figure 5.20. The equivalent circuit model simulates a single cell's charging and discharging process in the battery. The simulated battery includes three packages, and each package includes 120 cells. All cells are interpreted to be serially connected within the battery instead of in parallel. The inputs of the whole BEV model are cmd_{accel} and cmd_{decel}, representing the controller's output control commands for accelerating and decelerating the ego-car. Based on these two inputs and the maximal power of the battery, a provided output power of the battery is calculated and transferred to the next instantiated electric motor model.

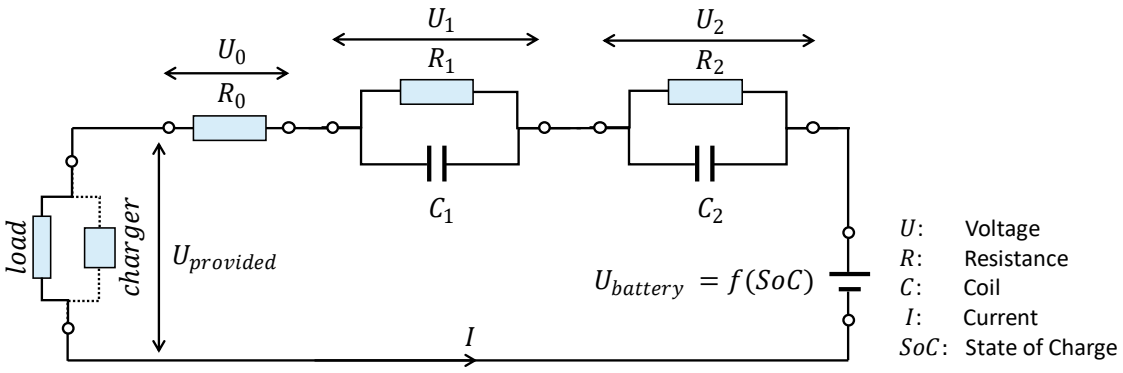


Figure 5.20: Lithium-Ion Traction Battery Model Based on the Second Order Equivalent Circuit Model [120]

The electric motor model relies on an implemented characteristic diagram regarding the provided power ($p_{provided}$) of the traction battery and the motor's output torque (T_{em}). With the help of the electric motor's output torque, the gearbox model calculates its output torque based on a predefined static transmission ratio. Both electric motor and gearbox models include application parameters to simulate the machine's working efficiencies (electric motor: 0.95, gearbox: 0.9). Fixed ratios for the distribution of total brake torque and the gearbox's total output torque to the car's front and rear axis are

also included in the BEV model. Thus, the front and rear wheels models can use the distributed torque to simulate the wheels' rotation dynamics. The outputs of the drive forces acting on the wheels ($F_{at\_wheel\_front}, F_{at\_wheel\_rear}$) will be then provided. While simulating such a process, application parameters like tire radius are specified in the model. The chassis model focuses on the car's body and movement modeling. In this model, the ego car's driving resistance is calculated following the approach published in [121]. Finally, the ego-car's velocity (v_{ego_car}) for the next timestamp can be determined. This determined velocity is then used to update the models' internal parameters like the wheels' angular velocities ($w_{wheel\_front}, w_{wheel\_rear}$).

After the physical processes in the ego-car's drive components are simulated, the determined ego-car's velocity (v_{ego_car}) will be transferred via TraCI into the SUMO simulation by calling the method *traci.vehicle.setSpeed()*[19]. Thus, SUMO can update the ego-car's visualization in SUMO by moving it to the following corresponding geographic location along the route. Visualization of the other traffic participants' movements will automatically be updated without a realistic underlying driving dynamics simulation. Due to high implementation complexity, detailed physical equations implemented for the driving dynamics simulation are not intensively introduced in this section. However, they can be found in the submitted accompanying code implementation.

5.5.3 Route-Based Adaptation Unit (RAU) in Implemented Technical System

As introduced in Section 5.1, the ACCC following the proposed architecture style is constructed with a hierarchical architecture including three layers. On each layer, a "SICAP-K" component structure is deployed. The "P" refers to the physical system that has been introduced previously. The other four "SICA-K" components (cf. Figure 5.15) on three layers are deployed as three subsystems in the technical system. The first subsystem on the highest layer of the hierarchical architecture is the route-based adaptation unit (RAU).

[19] This is a method defined in TraCI to manipulate the state of certain vehicle in the simulation. Link: https://sumo.dlr.de/docs/TraCI/Change_Vehicle_State.html (accessed on 28th Apr. 2022).

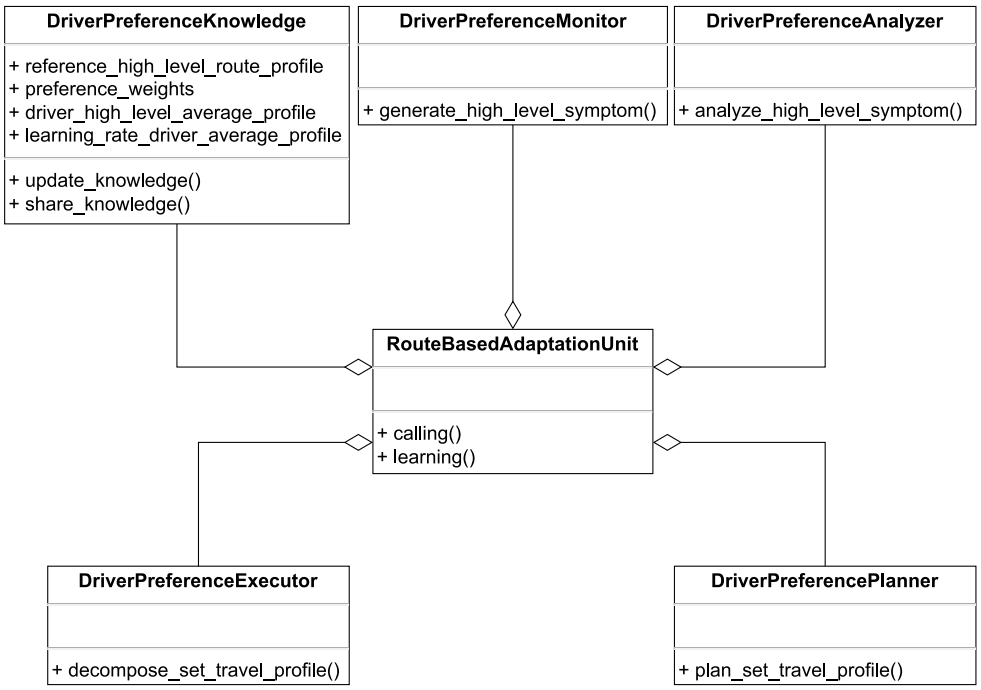


Figure 5.21: Class Diagram for the Construction of Route-Based Adaptation Unit

The RAU aims to observe the driving behaviors of the human driver while manual driving and learn their high-level driving preferences. The high-level preferences are defined in this dissertation as preferred travel time, energy consumption, and driving comfort during the previous trips on the same route. Thus, the RAU can automatically plan an optimized, high-level personalized route-based set travel profile for the driver before the trip once it is activated to take over the ego-car's longitudinal control. For this purpose, learning() and calling() are defined in the class *RouteBasedAdaptationUnit* to realize the functionalities of learning driving preferences and planning set travel profiles, as illustrated in Figure 5.21. Additionally, other classes with aggregation relationships with the class *RouteBasedAdaptationUnit* are defined to implement the "SICA-K" components in the RAU. Figure 5.21 provides an overview of these classes. The following sections will introduce more details of the classes and their instantiations representing the "SICA-K" components.

A significant point that needs to be emphasized here is that the functionality of learning driving preferences was implemented with a small difference compared to the original design of the ACCC. In the original design, the learning process can also be triggered

by the “SICA” components once they have identified a difference between the observed input facts and their available domain knowledge (cf. Section 5.3.2). After learning, these components push the updated domain knowledge to the “K” component for synchronization (cf. Section 4.2.6.2). Nevertheless, in the implementation of this dissertation, the learning process of the domain knowledge is only triggered by the “K” component (e.g., driver preference knowledge component). Then the updated knowledge is pushed to the “SICA” components for knowledge synchronization. The same implementation approach is also applied to the learning processes in the route-segment-based adaptation unit (RSAU) and the cycle-time-based control unit (CTCU).

5.5.3.1 Driver Preference Knowledge

As introduced in Section 5.2.2, the *Driver Preference Knowledge* component plays the role of a knowledge repository. It communicates with other components (“S^2”, “I^2”, “C^2”, “A^2”) in the RAU to provide them required domain knowledge. For this purpose, appropriate technologies like the database can be applied to implement the knowledge repository to store huge vehicle data. However, the ACCC’s implementation in this dissertation focuses on a co-simulation instead of the real hardware. Thus, domain knowledge is represented by different attributes in the class *DriverPreferenceKnowledge* (cf. Figure 5.21).

Domain Knowledge Attributes in the class *DriverPreferenceKnowledge*	**Sub-Attributes**
$+reference\_high\_level\_route\_profile$	$+reference\_high\_level\_route\_profile.x\_pos$ $+reference\_high\_level\_route\_profile.y\_pos$ $+reference\_high\_level\_route\_profile.accumulated\_dist$ $+reference\_high\_level\_route\_profile.index\_route\_points$
$+preference\_weights$	$+preference\_weights.time\_weight$ $+preference\_weights.comfort\_weight$ $+preference\_weights.consumption\_weight$
$+driver\_high\_level\_average\_profile$	$+driver\_high\_level\_average\_profile.accumulated\_time$ $+driver\_high\_level\_average\_profile.accumulated\_comfort$ $+driver\_high\_level\_average\_profile.accumulated\_consumption$
$+learning\_rate\_driver\_average\_profile$	-

Table 5.3: Defined Attributes of Domain Knowledge in the Class DriverPreferenceKnowledge

Subsequently, the instantiated objects are saved as individual “.npy” files in a folder of the local project repository, which is interpreted as the ACCC’s knowledge base. The

attributes representing the domain knowledge are called “domain knowledge attributes” in this section. Table 5.3 provides an overview of more details about these attributes.

The first domain knowledge refers to a high-level reference route profile. As introduced in the ACCC’s preliminary design, the ACCC needs the route profile saved in the navigation system to plan the high-level set travel profile and the middle-level driving strategy. For this purpose, the route needs to be segmented by a group of intermediate route points. For example, the simulated route from GS to CLZ is represented by 27663 intermediate route points in the implementation, and the simulated route from CLZ to GS includes 28250 route points. Since the RAU focuses on the high-level route profile, not all these intermediate route points are interesting. Instead, only significant route points representing the intersections are important for the planning high-level set travel profile.

During the ACCC prototype implementation, the resolution of the route points representing a so-called high-level reference route profile was thus massively reduced. Following the original order of the points, one route point is taken out of every 1000 route points to represent the intersections on the route. Several sub-attributes like *.x_pos* and *.y_pos* are defined to describe the features of the attribute + *reference_high_level_route_profile*. The sub-attribute *.accumulated_distance* relates to the accumulated distance from the first route point (the trip’s origin) to each following route point.

In addition to the high-level reference route profile, another domain knowledge refers to the driver’s preferred criteria weights of their driving preferences, as mentioned in Table 5.1. Three sub-attributes (*.time_weight*, *.comfort_weight*, and *.consumption_weight*) are thus defined to represent the three criteria weights. During the class instantiation, these weights are initialized by normalized values with a sum of one. Thus, the weights can represent the criteria’s importance for the driver from different perspectives.

As illustrated in Figure 5.21, the class *RouteBasedAdaptationUnit* includes a method *learning()*. While calling this method, the method *update_knowledge()* in the class *DriverPreferenceKnowledge* will be called. Such a case happens once the human driver completes a trip with the ego-car by manual driving. Based on the observed and

collected driving data during the trip ($recorded\_profile^{high\_level}$), the RAU will try to learn the driver's preferred average travel profile by updating the currently saved average travel profile ($average\_profile_{current}^{high\_level}$) to an updated profile ($average\_profile_{updated}^{high\_level}$).

The final attribute in the class *RouteBasedAdaptationUnit* represents a learning rate for learning the driver's average travel profile ($learning\_rate$), which is used to control the learning process and initialized with 0.08 in the implemented instance. The learning process can be generalized by equation (5.1) as follows:

$$\begin{aligned} & average\_profile_{updated}^{high\_level} \\ & \quad = average\_profile_{current}^{high\_level} - learning\_rate \\ & \quad \cdot (average\_profile_{current}^{high\_level} - recorded\_profile^{high\_level}) \end{aligned} \tag{5.1}$$

Either the current or the average travel profile still includes sub-profiles from three perspectives, including trajectories of the accumulated average travel time, the accumulated average energy consumption of the ego-car, and its accumulated average acceleration (representing the driving comfort), along with the route points in the high-level reference profile for representing the intersections on the route. Updating the profiles regarding the travel time, energy consumption, and acceleration (representing driving comfort) needs to be calculated separately by following Equation (5.1).

Figure 5.22 shows a visualization example of the learning process of the human driver's high-level average profile within several learning cycles. It is indicated in the figure that the average profile of the travel time and the energy consumption is continually changed along with the learning cycles.

Another method, *share_knowledge()*, is also defined in this class. This method is implemented to realize the knowledge sharing between multiple knowledge components across different layers in the ACCC's architecture (cf. Section 4.2.6.4). In the implemented co-simulation, such a process can easily be realized by exchanging the variables' values initialized in different class instances. In the following Section 5.5.4, more details about this process will be introduced.

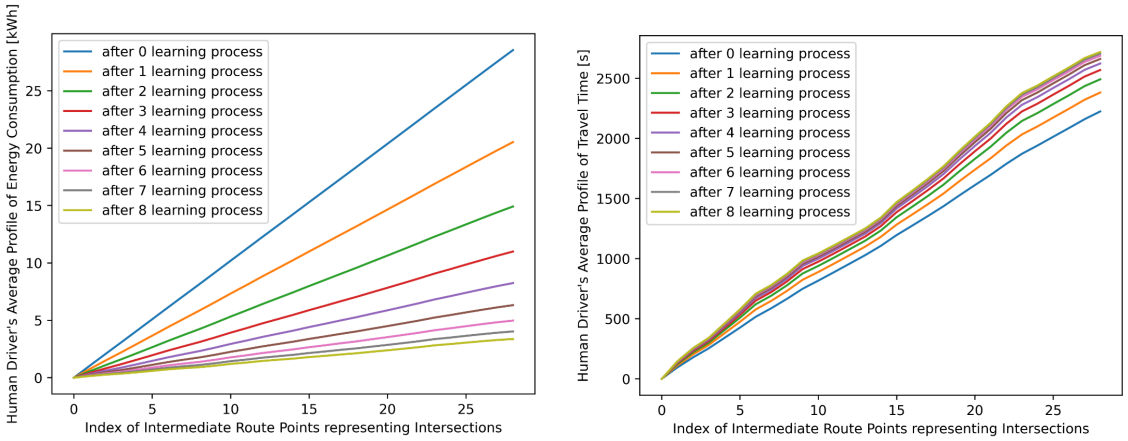


Figure 5.22: Example of Learning the Human Driver's High-Level Average Profile with Multiple Learning Cycles

5.5.3.2 Driver Preference Monitor

The component *Driver Preference Monitor* is designed to collect the sensory data from the physical system and generate a symptom (defined as structured data). For this purpose, the driver preference monitor needs to preprocess and aggregate the sensory data if necessary. The class *DriverPreferenceMonitor* defines a method *generate_high_level_symptom()* to complete the preprocessing and aggregation tasks. In this method, two input parameters are required: *trip* and *ego_pos_xy*. The parameter *trip* includes the current trip profile like the trip name, the origin, and the destination. With the help of the parameter *trip* as an indicator, the ACCC knows which reference route profile should require from the ego-car's navigation system. The parameter *ego_pos_xy* is used to indicate the ego-car's current location. The data of both input parameters are then allocated to form the symptom.

5.5.3.3 Driver Preference Analyzer

In the ACCC's design, the component *Driver Preference Analyzer* takes the symptom provided by the driver preference monitor as input to decide whether it is necessary to request the (re-)planning of a high-level set travel profile. For this purpose, a method *analyze_high_level_symptom()* is defined in the class *DriverPreferenceAnalyzer* (cf. Figure 5.21).

In the original design, the RAU plans the high-level personalized set travel profile in two cases: (1) at the beginning of the trip or (2) when the planned set travel profile significantly deviates from the current performed trip profile. In this dissertation, a simplified implementation only focusing on the first case is completed. The method *analyze_high_level_symptom()* compares the ego-car's current location with the trip's origin to identify whether the ego-car is at the beginning of the trip. In the case of the beginning of the trip, the method *analyze_high_level_symptom()* generates an output to request the planning of the high-level set travel profile.

5.5.3.4 Driver Preference Planner

The fundamental functionality of the *Driver Preference Planner* is to plan a driver-individual high-level set travel profile once the ACCC is activated. For this purpose, a method *plan_set_travel_profile()* is defined in the class *DriverPreferencePlanner* (cf. Figure 5.21) to complete the planning task. The set travel profile includes several sub-profiles regarding different perspectives: travel time, energy consumption, and the driving comfort represented by the ego-car's acceleration. Some examples of the sub-profiles have been illustrated in Figure 5.22. The individual values on the figure's y-axis refer to the accumulated travel time and energy consumption that the ego-car needs to reach the corresponding intermediate route points considering the trip starting from the origin.

In the method *plan_set_travel_profile()*, it is possible to deploy machine learning or classical optimization algorithms to complete the planning task. In such a case, criteria and weights would be required to evaluate the qualities of candidate set travel profiles, which massively increases implementation complexity. In the implemented ACCC prototype, this method is implemented more simply. During the ACCC's learning process, by calling the method *learning()*, the learned average travel profile (cf. Section 5.5.3.1) is updated and saved in the ACCC's knowledge base as a ".npy" file. In the method *plan_set_travel_profile()*, the saved average travel profile is extracted from the knowledge base and taken as the method's output.

5.5.3.5 Driver Preference Executor

The final *Driver Preference Executor* in the RAU is designed to decompose the planned set travel profile provided by the driver preference planner. Another method *decompose_set_travel_profile()* is defined in the class *DriverPreferenceExecutor* to complete the decomposition task, as illustrated in Figure 5.21. As introduced in Section 5.2.3, the decomposed set travel profile should be used as specifications for the route-segment-based adaptation unit (RSAU) deployed on the lower layer of the ACCC's architecture. Since the RSAU focuses on planning the middle-level strategy only for individual route segments, the method *decompose_set_travel_profile()* here tries to convert the planned travel profile into segment-wise specifications.

For example, a planned set travel profile includes a trajectory of the accumulated set travel time for 27 high-level intermediate route points representing intersections (cf. Figure 5.22). This means the whole route has 26 route segments, and thus 26 different partial profiles of set travel time for individual segments can be decomposed based on the travel time trajectory for the whole route. The same decomposition can also be performed for the trajectory of set consumption and set acceleration representing driving comfort. All the calculated partial set profiles for individual segments are then taken as specifications and provided to the RSAU.

5.5.4 Route-Segment-Based Adaptation Unit in Implemented Technical System

The second subsystem in the technical system is the RSAU. Like in the RAU, another "SICA-K" component structure is deployed in the RSAU. For this reason, five classes and their corresponding methods are defined to implement the RSAU, as illustrated in the class diagram in Figure 5.23.

The RSAU also has two functionalities: planning the middle-level driving strategy and learning the driver's middle-level preferences. Thus, the class *RouteSegmentBasedAdaptationUnit* defines two methods: *calling()* and *learning()*. The same implementation approach used in the RAU to deal with the learning process is also applied here. It means that the learning process of the implemented prototype is triggered by the instance of the class *DriverStrategyKnowledge* using its included methods: *update_average_profile()*, *update_decision_trees()*, and

update_distribution_pred_behaviors(). The following sections will provide more implementation details about these classes and their instantiation.

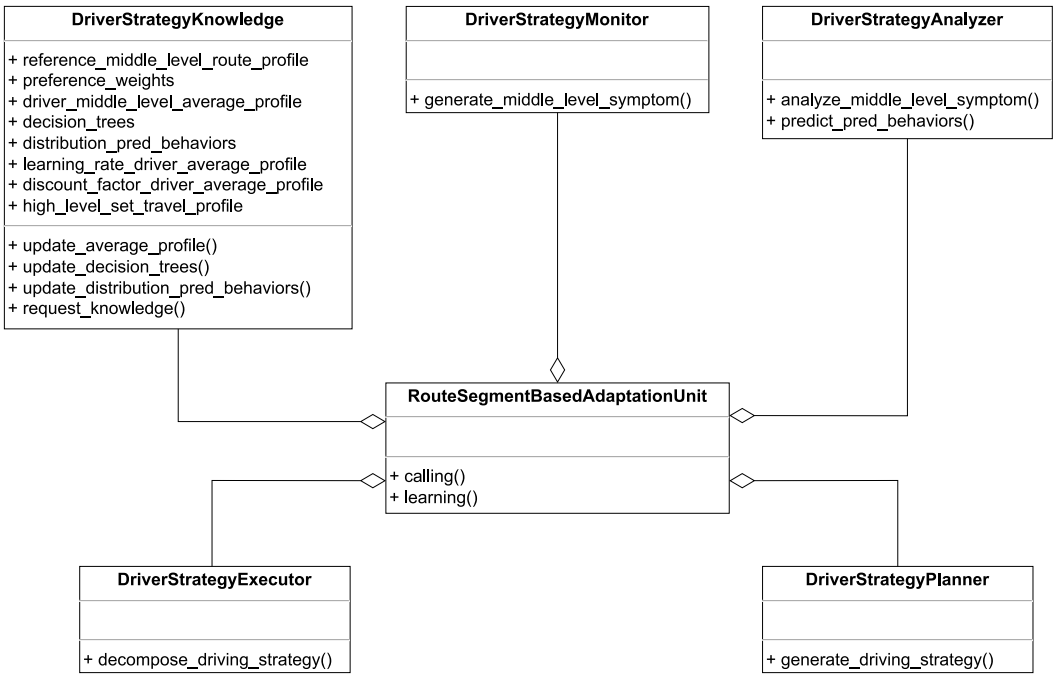


Figure 5.23: Class Diagram for the Construction of Route-Segment-Based Adaptation Unit

5.5.4.1 Driving Strategy Knowledge

In the ACCC's design, the *Driving Strategy Knowledge* component plays the role of a knowledge repository in the RSAU to store domain knowledge. Thus, the class *DrivingStrategyKnowledge* includes different attributes to deal with different knowledge types. Like the class *DriverPreferenceKnowledge* (cf. Section 5.5.3.1), different domain knowledge after the class instantiation is also saved as individual ".npy" files in a folder of the local project repository that serves as the knowledge base. After each learning process, these ".npy" files will be updated. Several sub-attributes are defined to acquire appropriate data structures for some of these attributes. Table 5.4 provides an overview of the defined attributes and their sub-attributes representing different domain knowledge.

The first attribute, *reference_middle_level_route_profile*, deals with the domain knowledge about the reference route profile in each segment, which is called the middle-level route profile in this dissertation. As presented earlier, the highway's route profile

from GS to CLZ includes 27663 intermediate route points. From these route points, 27 significant route points representing the intersections are selected from these intermediate route points to form the high-level route profile, including 26 route segments (cf. Figure 5.22). These route points are interpreted as each route segment's beginning and endpoints. Thus, a reference middle-level route profile is defined as an array with 26 elements. Each element represents a single route segment and includes a group of intermediate route points to describe a more detailed route segment profile. Several sub-attributes like *.x_pos*, *.y_pos*, and *.accumulated_distance* are then defined to describe the route segment profile, as indicated in Table 5.4.

Domain Knowledge Attributes in the class *DriverStrategyKnowledge*	**Sub-Attributes**
$+reference\_middle\_level\_route\_profile$	$+reference\_middle\_level\_route\_profile.x\_pos$ $+reference\_middle\_level\_route\_profile.y\_pos$ $+reference\_middle\_level\_route\_profile.accumulated\_dist$ $+reference\_middle\_level\_route\_profile.index\_route\_points$
$+preference\_weights$	$+preference\_weights.time\_weight$ $+preference\_weights.comfort\_weight$ $+preference\_weights.consumption\_weight$
$+driver\_middle\_level\_average\_profile$	$+driver\_middle\_level\_average\_profile.accumulated\_time$ $+driver\_middle\_level\_average\_profile.accumulated\_comfort$ $+driver\_middle\_level\_average\_profile.accumulated\_consumption$
$+decision\_trees$	-
$+distribution\_pred\_behaviors$	-
$+learning\_rate$	-
$+discount\_factor$	-
$+high\_level\_set\_profile$	$+high\_level\_set\_profile.set\_average\_travel\_time$ $+high\_level\_set\_profile.set\_average\_comfort$ $+high\_level\_set\_profile.set\_average\_consumption$

Table 5.4: Defined Attributes of Domain Knowledge in the Class DriverPreferenceKnowledge

The RSAU is designed to plan the optimized (i.e., driver-preferred) middle-level driving strategy. The middle-level driving strategy includes a location-based trajectory of cruise speed or headway for a single following route segment. Domain knowledge manifested by several attributes is thus required since the planning task can be interpreted as an optimization process.

The first *preference_weights* attribute refers to the weights of driver's preferences describing the importance of different criteria like travel time and energy consumption

(cf. Table 5.1), defined similarly in comparison to the case of class *DriverPreferenceKnowledge.* Another attribute refers to the driver's middle-level average profile, which is taken as a reference to evaluate the qualities of different candidate driving strategies. Compared to the attribute *high_level_average_profile* in the class *DriverPreferenceKnowledge*, the *high_level_set_travel_profile* is defined similarly. It is used to save the provided high-level average profile provided by the RAU. The attribute *decision_tree* describes the search space of candidate cruise speeds and headways while planning the strategy. The final two *learning_rate* and *discount_factor* attributes represent two application parameters: the learning rate and discount factor. Both parameters will then be used in a reinforcement learning (RL-) algorithm, which learns the driver-preferred middle-level driving strategy, namely their preferred location-dependent trajectory of cruise speed and headway for the intermediate route points.

Since the RSAU requires diverse domain knowledge to complete its tasks, different methods are defined to complete the learning process. The first learning process focusing on the segment-wise average driving profile of the human driver is realized by the method *update_average_profile()*. For example, the whole route is divided into 26 route segments, as introduced previously. The segment-wise average profile includes three sub-profiles representing the travel time, the ego-car's energy consumption, and the acceleration. Each sub-profile is implemented as an array with 26 elements representing the 26 route segments. The currently saved average profile ($average\_profile_{current}^{middle-level}$) will be updated ($average\_profile_{updated}^{middle-level}$) via *update_average_profile()* during the learning process by following the same calculation principle in Equation (5.1) with the learning rate ($learning\_rate$) and inputs of the recorded profile ($recorded\_profile^{middle\_level}$). Each average and current profile can still further be split into three terms regarding travel time, consumption, and driving comfort as follows:

- $average\_time\_profile_{current}^{middle-level}, recorded\_time\_profile^{middle\_level}$
- $average\_consumption\_profile_{current}^{middle-level}, recorded\_consumption\_profile^{middle\_level}$
- $average\_comfort\_profile_{current}^{middle-level}, recorded\_comfort\_profile^{middle\_level}$

Due to the high similarity of the calculation approach compared to Equation (5.1), a new equation is not listed in this section.

Another method used in the learning process is the *update_decision_trees()*. This method is implemented to update decision trees. Since the RSAU plans a trajectory of cruise speed in the case of no preceding car and plans a trajectory of headway once a preceding car is available as the middle-level driving strategy, two decision trees are defined. In the decision trees, the candidate driving strategies are not directly saved. Instead, a matrix of scores representing the strategies' qualities is included in the trees. The candidate driving strategies are represented by the data structure of the implemented decision trees in Python.

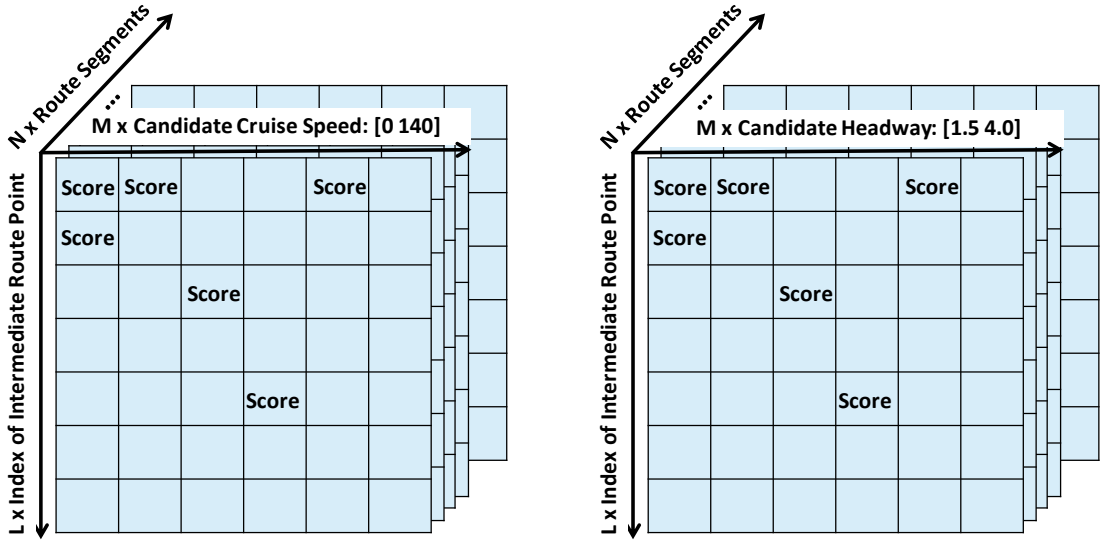


Figure 5.24: Data Structure of the implemented Decision Trees in the RSAU of ACCC

Figure 5.24 illustrates an overview of the data structure of the implemented decision trees. Each decision tree is defined as a 3D array (cf. attribute: *decision_trees* in Table 5.4). The three dimensions of the array, relate to the route segments, the index of intermediate route points within the segment, and the candidate cruise speed, respectively. For example, the whole route includes 26 route segments. Each route segment includes 200 route points. Due to the ego-car's maximum speed of 140 km/h, there are 141 candidate cruise speeds once the resolution is set to 1 km/h. Thus, a 26x200x141 array can describe the decision tree of candidate cruise speed. Following

a similar principle, the 3D array of the candidate headway can be defined. This dissertation sets the valid value range of the headway from 1.5 to 4.0 seconds, and the value resolution for headways' state segmentation is set to 0.1 seconds. Thus, the scores representing the qualities of candidate driving strategies can be saved as an element in the 3D array, as illustrated in Figure 5.24.

Learning the human driver's driving preferences for cruise speed and headway is realized by updating the saved scores in the 3D array. The Q-learning algorithm (cf. Section 2.4.1) for reinforcement learning is applied to realize this learning process. The previously mentioned learning rate and discount factor are also used in the Q-learning algorithm.

Generally, it can be understood that the Q-learning algorithm builds a function for updating the score ($score_{updated}$) based on the original score ($score_{original}$) saved in the array and a calculated reward. In addition, two application parameters, learning rate and discount factor (cf. Table 5.4) will also be used as follows:

$$score_{updated} = score_{original} + learning\_rate \cdot \left(reward + discount_{factor} \cdot score_{next\_max} - score_{original}\right) \quad (5.2)$$

The reward calculation is completed based on three terms: travel time, consumption, and driving comfort. For calculating the terms, the driver's preference weights (w_{time}, $w_{consumption}$, $w_{comfort}$, cf. Table 5.1) are used. Since fundamental theories about the Q-learning algorithm have been introduced in Section 2.4.1, a detailed calculation path for updating the score will not be introduced in this section. However, it was well-formulated in a previously published paper [122].

As introduced in Section 5.2.2, the RSAU (route-segment-based adaptation unit) includes the functionality of predicting the future context of the driving environment. For this purpose, a machine learning-based prediction approach relying on kernel density estimation (KDE) is applied in the RSAU (cf. Section 2.4.2).

Since the KDE's prediction relies on a density distribution of the preceding cars' behaviors, an attribute *distribution_pred_behaviors* (cf. Table 5.4) is defined in the class

DrivingStrategyKnowledge to save the distribution. In the implementation, this distribution is defined as a 3D array. Figure 5.25 provides an overview of the data structure of the 3D array. The 3D array for storing the preceding cars' behavioral density distribution differs from the 3D array for storing the scores of candidate driving strategies. This array is constructed by concatenating several sub-arrays representing the density distribution for individual route segments, marked in different colors within Figure 5.25.

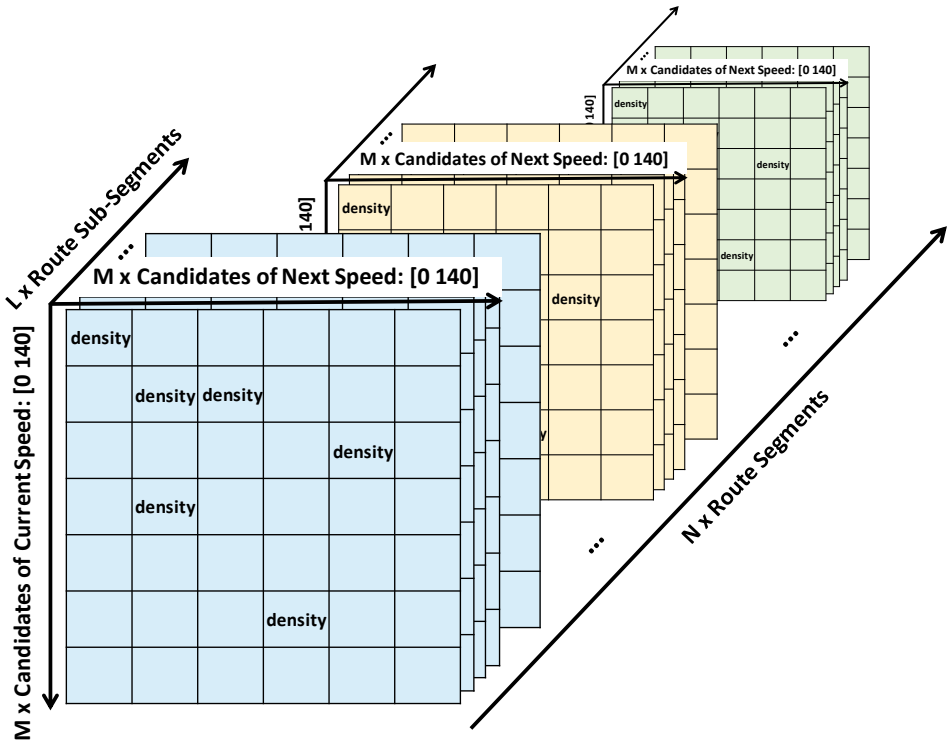


Figure 5.25: Data Structure of the Implemented Preceding Car's Behavioral Density Distribution

For example, a route includes 26 route segments separated by several route points representing the intersections. The 3D array will then include 26 colored sub-arrays. Since there are still many route points to form sub-segments within each segment, the sub-array can be further divided into several 2D arrays. Each 2D array corresponds to the distribution for a single route point. The state-space of the preceding car's speed is divided into 141 states, representing 0 km/h to 140 km/h (with a resolution of 1 km/h). Thus, a density distribution regarding the preceding car's speed for the next route point depending on its current speed for the actual route point can be formulated and saved in the 2D array. In this 2D array, the density values in corresponding speed-dependent

contexts are located. A method *update_distribution_pred_behaviors()* is defined in the class *DrivingStrategyKnowledge* to implement the learning of the preceding car's behavior distribution by updating the densities in the array.

The concrete calculation path for this update process was already included in the theoretical fundamentals of KDE (cf. Section 2.4.2) and thus will not be introduced in this section. Compared to the original KDE, the only implementation difference is that the overall density distribution is normalized (to keep the maximal density value in the distribution always equal to one) after each learning process. Thus, the influence of the new upcoming density distribution will be appropriately considered.

In this implementation of this dissertation, the KDE prediction is strongly simplified by considering a single influencing parameter of the distribution (i.e., the preceding car's current speed) to reduce implementation complexity. Theoretically, more input parameters can also be considered. A previously published paper can provide more details about the KDE-based prediction with multiple influence parameters like the preceding car's previous acceleration and speed [70].

The final method, *request_knowledge()*, is defined as an example in this dissertation to demonstrate UC4: knowledge sharing between knowledge components across layers (cf. Section 5.3.4). The RSAU calls this method to acquire the previously mentioned driver's preference weights (cf. Table 5.1). On the other side, the defined *method share_knowledge()* in the class *DriverPreferenceKnowledge* of RAU is responsible for answering the request from the RSAU.

5.5.4.2 Driving Strategy Monitor

After introducing the class *DrivingStrategyKnowledge*, the next defined class *is DrivingStrategyMonitor*, which is applied to generate a symptom for the RSAU after instantiation. For this reason, a method *generate_middle_level_symptom()* is defined in this class. A sub-class *Symptom* is defined to specify the data structure of the symptom.

5.5.4.3 Driving Strategy Analyzer

The next class, *DrivingStrategyAnalyzer*, is defined in the RSAU to analyze the current context manifested by the symptom. For this purpose, a method

analyze_middle_level_symptom() is defined in this class. In this method, several subtasks are completed. As presented earlier, the route is divided into different route segments by significant route points representing intersections. Since the RSAU focuses on planning the driving strategy for individual route segments, it only plans a trajectory of cruise speed and headway as the middle-level driving strategy until the end of the following segment. Thus, it is required that the RSAU knows which route segment the ego-car is facing. This is completed by an internal method defined in the method *analyze_middle_level_symptom()* based on the reference middle-level route profile (cf. Table 5.4) and the ego-car's current location.

After identifying the current route segment, the driving strategy analyzer aims to identify whether it is necessary to request (re-)planning the middle-level driving strategy. In the implemented driving strategy analyzer, a state machine including three states and corresponding transition conditions is defined, as illustrated in Figure 5.26.

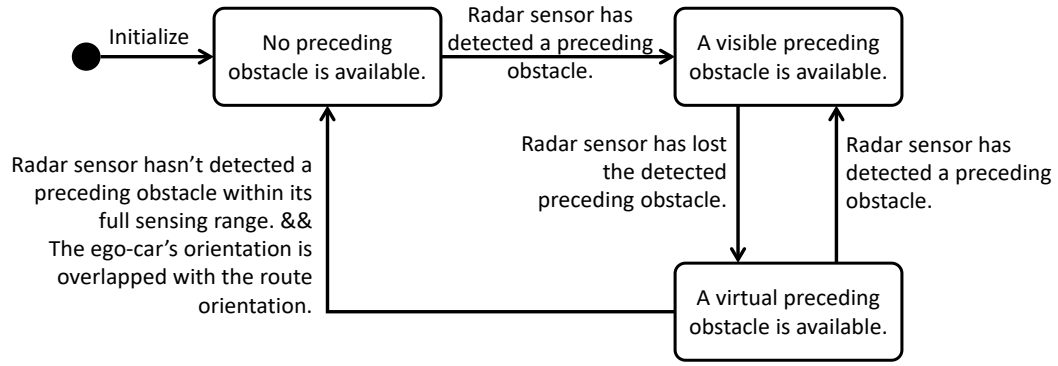


Figure 5.26: State Machine Implemented in the Method analyze_middle_level_symptom() of Driving Strategy Analyzer

With the help of this state machine, the driving strategy analyzer can identify the current state regarding the preceding obstacle, which in the built co-simulation is represented by the preceding car. Other traffic participants like pedestrians are not considered in the simulation. There are three cases where the driving strategy analyzer would request (re-)planning of the middle-level strategy:

- The ego-car enters the next route segment.
- The radar sensor detects a visible preceding car.

- The ACCC has confirmed that the previously detected invisible virtual preceding obstacle is gone.

For example, the RSAU would be initialized with the state “No preceding obstacle is available” at the beginning of the trip. In this case, once the ego-car enters a new route segment, a middle-level strategy consisting of the trajectory of cruise speed until the end of the route segment would be (re-)planned. Once the radar sensor has detected a preceding car, which in this dissertation is called a “visible preceding obstacle” (cf. Figure 5.26), a replanning of strategy would be performed since a trajectory of headway until the end of the route segment is required. In the case of such a visible preceding obstacle, the trajectory would also be updated once the ego enters a new route segment.

Compared to classical ACC and its variants currently on the market, a significant difference in ACCC is its memory capability. As introduced in Section 3.5.2, current ACCs may accelerate the ego-car before a curve once a previously detected preceding car becomes invisible due to the curve for the radar sensor. Thus unfavorable deceleration after the curve may be caused.

The ACCC deals with such an invisible preceding car in another way. Instead of forgetting it, the ACCC keeps the previously detected preceding car in memory as a virtual obstacle and considers its influence while (re-)planning the headway. For this purpose, another method, *predict_pred_behaviors()*, is defined in the class *DrivingStrategyAnalyzer*. In this method, the KDE algorithm (cf. Section 2.4.2) is applied to forecast a location-based trajectory of the preceding car’s speed along with the intermediated route points in the focusing route segment until its endpoint. Since the underlying principle of the algorithm has been introduced in Section 2.4.2, more detail will not be included in this subsection. Generally, it can be understood that the prediction is completed relying on the saved density distribution in the ACCC’s knowledge base, as introduced in Figure 5.25. The speed profiles with the highest density values for individual route points will be taken and combined as a trajectory to describe the predicted preceding obstacle’s driving behaviors.

The virtual obstacle would be deleted from the ACCC’s memory only when two conditions are fulfilled:

- The radar sensor cannot detect the preceding obstacle within its full sensing range (implemented configuration: 400 m, ±15°),
- The ego-car's orientation overlaps with the route's orientation (evaluated by a threshold value of orientation difference: ±10°).

In such a case, the ACCC would assume that the previously detected preceding obstacle is no longer relevant for planning the middle-level driving strategy.

5.5.4.4 Driving Strategy Planner

If the driving strategy analyzer requests the planning of the middle-level driving strategy, the driving strategy planner would be activated to complete the planning task. Depending on the current state regarding the preceding obstacle, the driving strategy includes either a location-based trajectory of cruise speed or headway. Once the preceding obstacle is available (either visible or virtual), its predicted driving behaviors will also be taken as inputs during the planning of the driving strategy.

The class *DrivingStrategyPlanner* defines a method *generate_driving_strategy()* to deal with the planning task. As presented earlier, a Q-learning algorithm is applied to learn the driver's middle-level driving preferences, which refers to their preferred cruise speeds and headways for different intermediate route points. The candidate cruise speeds and headways are quantitatively evaluated with scores and saved in the ACCC's knowledge base (cf. Figure 5.24). Based on these scores, the planning task becomes substantially simpler. The candidate cruise speed and headway with the highest score for each intermediate route point within the current route segment will be taken and combined into a trajectory. While planning the trajectory of headway for the ego-car, the predicted preceding car's driving behaviors in the form of a location-based speed trajectory will also be taken as output to transfer into the driving strategy executor.

5.5.4.5 Driving Strategy Executor

As introduced in the ACCC's design, the driving strategy executor is applied to decompose the planned middle-level driving strategy and convert it into an appropriate form. For this purpose, a method *decompose_driving_strategy()* is defined within the class *DrivingStrategyExecutor*.

Within this method, interpolation functions are generated based on the planned driving strategy of the ego-car:

- $cruise\_speed^{ego-car} = f(pos_x^{ego-car}, pos_y^{ego-car})$, with $pos_x^{ego-car}$, $pos_y^{ego-car}$ representing the ego-car's current location
- $headway^{ego-car} = f(pos_x^{ego-car}, pos_y^{ego-car})$, with $pos_x^{ego-car}$, $pos_y^{ego-car}$ representing the ego-car's current location

Additionally, the predicted preceding car's location-based future speed trajectory is converted into a time-dependent interpolation function:

- $travel\_distance^{preceding-car} = f(accumulated\_disappeared\_time^{preceding-car})$, with $accumulated\_disappeared\_time^{preceding-car}$ representing the accumulated disappeared time duration of the preceding car, which is derived from the predicted preceding car's speed profile and the reference middle-level route's distance profile

In the implemented ACCC prototype, these interpolation functions replace the decomposed middle-level driving strategy in the original design. After the interpolation functions are built, they are taken as outputs to transfer into the subsystem cycle-time based control unit (CTCU) deployed on the lowest layer of the ACCC's hierarchical architecture.

5.5.5 Cycle-Time-Based Control Unit in Implemented Technical System

The cycle-time-based control unit (CTCU), as the subsystem deployed on the lowest layer of the ACCC's hierarchical architecture, is designed to realize interactions with the physical world by taking over its longitudinal control while the ego-car is driving. A class *CycleTimeBasedControlUnit* is defined to implement the prototype. Since the CTCU also follows the design principle based on the same "SICA-K" component structure, five additional classes for the CTCU's included components are defined. Figure 5.27 provides an overview of these classes and their included accompanying methods.

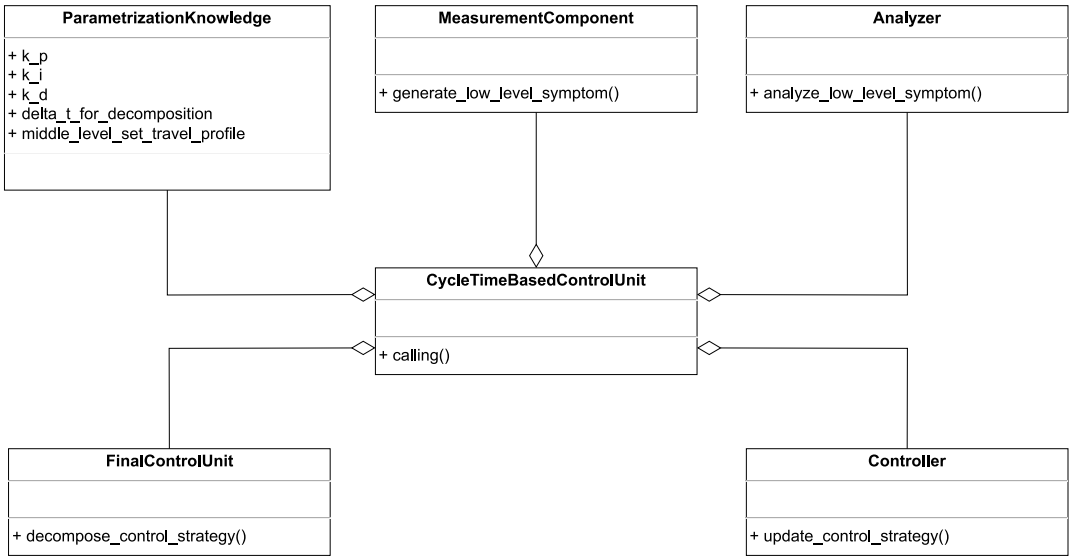


Figure 5.27: Class Diagram for the Construction of Cycle-Time-Based Control Unit

Although the CTCU in the original design of the ACCC includes a learning ability, such ability is excluded in the implemented prototype of this dissertation to reduce implementation complexity. Thus, unlike the other two subsystems, the class *CycleTimeBasedControlUnit* only has a defined method *calling()* without the method *learning()*. When calling the method *calling()* is called, the other classes and particularly their included methods will be called to complete the control task. The following sections will introduce more details about these classes and their methods.

5.5.5.1 Parametrization Knowledge

Like the driver preference knowledge and the driving strategy knowledge components, there is also a *Parametrization Knowledge* component in the CTCU that plays the role of the knowledge repository. Several attributes are defined within the class *ParametrizationKnowledge* to specify different domain knowledge data structures.

Since the CTCU controls the ego-car in its longitudinal direction, a PID controller is implemented in the CTCU prototype. Thus, the CTCU requires an appropriate parameterization for P-, I-, and D-elements. For this reason, the attributes *k_p*, *k_i*, and *k_d* are defined and will be initialized during class instantiation. As previously discussed, the learning ability is excluded in the implemented CTCU. Thus, the controller

parametrization would always stay the same in the whole evaluation based on the built co-simulation.

As introduced in Section 5.2.4, the CTCU is designed as a time-triggered subsystem. Thus, another attribute, *delta_t_for_decomposition*, is defined in the class *Parametrization Knowledge* to specify CTCU's cycle time knowledge. During the class instantiation, this attribute is initialized with the cycle time of the overall co-simulation. The final attribute, *middle_level_set_travel_profile*, is defined to deal with the provided inputs by the RSAU. The three interpolation functions mentioned in the previous subsection will be assigned to this attribute in the implemented prototype.

5.5.5.2 Measurement Component

The second defined class is the *MeasurementComponent*, including a method *generate_symptom()*. This method is designed to preprocess and aggregate the acquired sensory data. In the implemented prototype, the method is realized more simply by only assigning acquired data values to corresponding variables to form the symptom with an appropriate data structure.

5.5.5.3 Analyzer

The *Analyzer* in the CTCU is designed to analyze the context included in the acquired symptom to decide whether it is required to take the set cruise speed or the set headway as its reference variable. Thus, an internal algorithm to detect the preceding car is implemented in the method *analyze_low_level_symptom()*. In this algorithm, a preceding car will be detected as visible by the radar sensor once it is located within its sensing range (400 m) and sensing angle ($\pm$15°). The preceding car's availability will then be included in the Analyzer's output and sent to the Controller.

5.5.5.4 Controller

As illustrated in Figure 5.27, a further class *Controller* is defined for implementing the CTCU. Since the controller works based on the classical feedback control approach, an algorithm for realizing PID control is implemented within an included method *update_control_strategy()*. In the implementation, the D-element is eliminated by

initializing with zero since the PI controller is widely used as a classical approach in ACC variants [77][123].

The input of the *update_control_strategy()* is the planned middle-level driving strategy from the RSAU. As introduced in Section 5.5.4.5, three interpolation functions are included as implementation examples in the planned middle-level driving strategy. These interpolation functions are used here to generate the set values of the reference variable. For example, the function $cruise\_speed^{ego-car} = f(pos_x^{ego-car}, pos_y^{ego-car})$ can provide a set value of the driver-preferred cruise speed depending on the ego-car's current location. The function $headway^{ego-car} = f(pos_x^{ego-car}, pos_y^{ego-car})$ can generate a set value of the driver-preferred headway.

Following the basic control concept (cf. Section 2.1.1), the PI controller works based on the input of control error, calculated as the deviation between the set value of the reference variable and the current value of the controlled variable (cf. Figure 2.2). There is no problem getting the controlled variable's current value in the case of a visible preceding car by the radar sensor since the current headway can be acquired. However, suppose the preceding car is not visible and virtually "saved" in the ACCC's memory. In that case, the radar sensor cannot "see" the preceding car, and thus is impossible to provide the current headway.

Facing this issue, the final interpolation function $travel\_distance^{preceding-car} = f(accumulated\_disappeared\_time^{preceding-car})$ is designed and integrated into the RSAU's planned middle-level driving strategy. During the CTCU's operation, the time point when the preceding car disappears would be recorded to calculate the preceding car's accumulated disappeared time duration. Once a preceding virtual car becomes visible by the radar sensor, the recorded accumulated disappeared time will be reset to zero. The calculated disappeared time duration will then be taken as input to calculate the invisible preceding car's travel distance during its invisible period. From another perspective, the ego-car's travel distance during this period can also be calculated with the help of its speed profile. Thus, current headway can be derived from the deviation between both cars' travel distances.

After the control error is determined, it will then be transferred into the implemented function of the PI controller. Since the underlying algorithm of the PI controller is already well-known, its detailed calculation processes will not be described in this subsection. More details can be found in relevant literature [9][124].

5.5.5.5 Final Control Component

A class *FinalControlUnit* with a method *decompose_control_strategy()* is defined in the implementation. The *Final Control Unit* serves to decompose the determined control strategy by the Controller. In the implemented prototype, the method *decompose_control_strategy()* forwards the determined values of manipulated variables for controlling the ego-car's acceleration and deceleration ($accel\_cmd$, $decel\_cmd$) directly to the ego-car's model built in the physical system (cf. Section 5.5.2.3) due to the lack of predictive planning of the implemented prototype of Controller. Once the controller is implemented, e.g., based on the model predictive control (MPC) concept, it will provide a value trajectory of manipulated variables instead of individual value. Thus, the method *decompose_control_strategy()* could then include a more complicated algorithm to decompose the trajectory into single values for each running cycle of the feedback control loop.

5.6 Evaluation of Artificial Cognitive Cruise Control

In the previous section, implementation details of the ACCC prototype were intensively discussed. As presented at the beginning of Section 5.5, this dissertation aims to build a co-simulation platform to evaluate the ACCC prototype's performance after implementing the prototype. In this section, more detail about the evaluation work will be provided.

5.6.1 Hypotheses

For evaluation of the implemented ACCC prototype, this dissertation has selected several alternative approaches as candidates to build a performance benchmark. Since the ACCC aims to learn the preferences of a human driver, manual driving is taken as one reference approach. In addition, the classical ACC based on the basic control is also selected as a further candidate approach in the benchmark since the ACCC can

also be interpreted as a further improvement of the classical ACC. Thus, two hypotheses are proposed based on the candidate approaches in the dissertation, which will then be validated within the following evaluation work:

- Hypothesis 1: The ACCC realizes a higher performance quality of the ego-car's longitudinal control than the classical ACC based on a statically parametrized PID controller.
- Hypothesis 2: The ACCC shows a higher performance quality in the ego-car's longitudinal control than the human driver's manual driving.

After proposing the hypotheses to be validated, a systematical quantitative evaluation of different approaches' control performances must still be performed. Thus, this dissertation has used the previously mentioned criteria regarding travel time, energy consumption and driving comfort (represented by the ego-car's acceleration), and their weights ($w_{time} = 0.4$, $w_{consumption} = 0.4$, $w_{comfort} = 0.2$, cf. Table 5.1) were used while learning the driver's preferences and planning strategy by the ACCC to complete the benchmark. A cost function to quantitatively evaluate the performed strategy's quality after each trip by using three candidate approaches is built as follows:

$$Quality_{final} = Quality_{basic} + Quality_{basic} \cdot Quality_{coefficiant} \tag{5.3}$$

Equation (5.3) states that the final quality depends on a basic quality (initialized with a constant of 100 in the implementation) and a so-called quality coefficient. The quality coefficient describes the fulfillment of the driver's preferences, which thus takes the preference weights into account during the calculation as follows:

$$\begin{aligned} Quality_{coefficiant} &= Quality_{time} \cdot w_{time} + Quality_{consumption} \cdot w_{consumption} \\ &+ Quality_{comfort} \cdot w_{comfort} \end{aligned} \tag{5.4}$$

As illustrated in Equation (5.4), the quality coefficient consists of three terms: $Quality_{time}$, $Quality_{consumption}$, $Quality_{comfort}$, which are respectively calculated by following equations as follows:

$$Quality_{time} = 2 \cdot \left(\frac{1}{1 + e^{Performed\_Profile_{time} - Average\_Profile_{time}}} - 0.5\right) \quad (5.5)$$

$$Quality_{consumption} = 2 \cdot \left(\frac{1}{1 + e^{Performed\_Profile_{consumption} - Average\_Profile_{consumption}}} - 0.5\right) \quad (5.6)$$

$$Quality_{comfort} = 2 \cdot \left(\frac{1}{1 + e^{abs(Performed\_Profile_{comfort}) - abs(Average\_Profile_{comfort})}} - 0.5\right) \quad (5.7)$$

In the equations above, all profiles ($Performed\_Profile_x, Average\_Profile_x$) represent a value trajectory of corresponding variables. The deviation between two profiles means the average value derived from both trajectories' deviations of values. The designed equations above guarantee that the output values are always normalized between -1 and 1. The preference weights (w_{time}, $w_{consumption}$, and $w_{comfort}$) are also normalized with a sum of 1. Thus, the calculated quality coefficient will always stay within the value range between -1 and 1.

5.6.2 Alternative Candidate Approaches within the Benchmark

As introduced in the previous section, two additional candidate approaches are planned within the performance benchmark to evaluate the implemented ACCC prototype. For this purpose, a classical ACC with the basic control (cf. Section 3.1), including a PI controller (without planning and prediction functionalities), and a human driver model for simulating manual driving were also implemented. The following sections will introduce more implementation details about both candidate approaches.

5.6.2.1 Manual Driving with Human Driver Model

The SUMO community has developed various human driver models, like Kauss's car-following model [125] and Erdmann's lane-change model [126]. Since the ACCC only focuses on the ego-car's longitudinal control, Treiber et al.'s intelligent driver model

(IDM), a well-known car-following model, is used to implement the human driver model in this dissertation [127]. The IDM describes the driving dynamics of a single car by calculating the car's set acceleration (i.e., the human driver's preferred acceleration/deceleration) for the next time step based on the current context regarding the preceding car. In this case, the human driver's driving activities in the ego-car's longitudinal direction are simplified as a decision-making process of the car's acceleration. Several fundamental equations as follows are defined in the IDM.

Equation (5.8) aims to calculate the ego-car's acceleration in the case of an open road without any preceding car as an obstacle. Nevertheless, an additional interaction term represented by Equation (5.9) must be considered when a preceding car is available. In this case, the ego-car's acceleration can be calculated by integrating both previous equations (cf. Equation (5.10)).

$$\dot{v}_{ego}^{free} = a_{max}\left(1 - \left(\frac{v_{ego}}{v_{desired}}\right)^{\delta}\right) \tag{5.8}$$

$$\dot{v}_{ego}^{interaction} = -a_{max}\left(\frac{s_{min} + v_{ego} \cdot H_{safe}}{s_{ego}} + \frac{v_{ego} \cdot \Delta v_{ego}}{2\sqrt{a_{max} b_{comfort}} \cdot s_{current}}\right) \tag{5.9}$$

$$\dot{v}_{ego} = \dot{v}_{ego}^{free} + \dot{v}_{ego}^{interaction} \tag{5.10}$$

The variables included in the equations above are illustrated as follows in Table 5.5:

Variable	Meaning	Physical Unit
$\dot{v}_{ego}^{free}$	Ego-car's set acceleration in the case of an open road (without a preceding car)	[m/s^2]
a_{max}	Ego-car's maximal acceleration	[m/s^2]
v_{ego}	Ego-car's current velocity	[m/s]
$v_{desired}$	Ego-car's desired velocity (e.g., preferred by the human driver)	[m/s]
δ	Acceleration exponent (as an application parameter, usually set to 4)	[-]
$\dot{v}_{ego}^{interaction}$	Reduction of ego-car's acceleration due to interaction with a preceding obstacle (e.g., a preceding car)	[m/s^2]
s_{min}	Minimum spacing (ego-car is not permitted to move forward once its distance to the preceding car is lower than the minimum spacing.)	[m]
H_{safe}	Ego-car's desired headway to the preceding obstacle (e.g., preferred by the human driver)	[s]

$s_{current}$	Ego-car's current distance to the preceding car (the ego-car's length shall be excluded in this distance)	[m]
Δv_{ego}	Velocity difference of ego-car and the preceding car	[m/s]
$b_{comfort}$	Ego-car's comfortable braking deceleration (a positive number, e.g., preferred by the human driver)	[m/s^2]
$\dot{v}_{ego}$	Ego-car's set acceleration	[m/s^2]

Table 5.5: Variables and Their Meanings in the Equations of Intelligent Driver Model (IDM) [127]

Equation (5.10) illustrates that the original IDM's output is the ego-car's set acceleration. Finally, the determined set acceleration of the ego-car is used in the implementation to calculate representative control commands (cmd_{accel} and cmd_{decel}) so that the ego-car's model can provide a final set speed for the ego-car. For this purpose, the predefined maximal acceleration (+2.8 m/s^2) and deceleration (-2.6 m/s^2) of the ego-car are used to ensure that the control commands' values can be normalized between [0 1] and [-1 0] (cf. Table 5.2). Subsequently, the set speed is forwarded to the SUMO simulation since SUMO provides a method *traci.vehicle.setSpeed()* in its interface to visualize the ego-car's movement (cf. Section 5.5.2.3).

As indicated in Table 5.5, values of several application parameters in the equations like $v_{desired}$ and $b_{comfort}$ are driver-dependent, which means that different drivers may have different value configurations. Thus, this dissertation extends the original IDM by integrating a recorded real driving data pool, covering an anonymous driver's driving behaviors during 600 repeated trips on the federal highway B241 in both directions between the city CLZ and GS in Germany.

5.6.2.2 Longitudinal Automated Driving with Classical ACC

In addition to the IDM, a classical ACC based on the PI controller was also implemented. Compared to the controller in the implemented ACCC prototype, there is no significant difference in the ACC's implementation. The only difference is that the ACCC's controller generates the set values of cruise speed or headway by itself, relying on the learned preferences of the human driver. However, in the case of classical ACC, these set values are directly generated from the recorded driving data (cf. Figure 5.16). Since the recorded driving data includes paired speed and location of the ego-car (represented by GPS positions), the speed value with the minimal distance compared to the simulated ego-car's current location is taken as the human driver's preferred set cruise speed.

Since the driving data does not include the headway profile, the driver-preferred set value of headway is derived relying on an interpolation function as follows:

$$headway_{set}^{ego-car} = \frac{headway_{max}^{ego-car} - headway_{min}^{ego-car}}{v_{max}^{ego-car} - v_{min}^{ego-car}} \cdot \left(v_{current}^{ego-car} - v_{min}^{ego-car}\right) + headway_{min}^{ego-car} \quad (5.11)$$

In Equation (5.11), the application parameters $headway_{min}^{ego-car}$ and $headway_{max}^{ego-car}$ are initialized with 1.5 and 4.0 seconds. The $v_{min}^{ego-car}$ and $v_{max}^{ego-car}$ are initialized with 0 and 140 km/h.

Subsequently, the generated set values of the cruise speed and headway are transferred into the implemented algorithm in the PID controller. This dissertation will not include more details since such an algorithm in the PID controller is already well-known. It is emphasized that the PID controller has the same parametrization compared to the controller in the CTCU of the ACCC. Such an approach guarantees that the ACCC and the classical ACC can have a more comparable basis within the benchmark and thus makes the evaluation results more meaningful.

5.6.3 Evaluation Framework

After introducing the implementation details of alternative candidate approaches in the benchmark, it is necessary to provide an overview of the evaluation framework. This section describes the test scenarios and the design of experiments included in the evaluation work.

As introduced in Section 5.5.1, the federal highway B241 between the city CLZ and GS (cf. Figure 5.17) is simulated within the built co-simulation platform to evaluate the ACCC's performance. For this reason, the implemented driver model representing the human driver's manual driving and the classical ACC is also evaluated with the help of this simulated highway.

Since the anonymous driver's recorded manual driving data pool (cf. Figure 5.16) contains 600 trip profiles (including 300 trips from GS to CLZ and 300 trips from CLZ to

GS), the planned test scenarios in the simulation address these trips. A user story is summarized to describe an overview of the simulation as follows:

> *A human driver lives in the city GS and works in the city CLZ in Germany. The driver frequently drives from home to his workplace at 8 a.m. and drives back home after work every day on the weekdays. If the driver drives the ego-car manually, the ACCC observes the driver's manual driving. After each trip, the ACCC learns the driver's driving preferences based on the observed driving profile during trips. Once the driver wants to be released from the routine driving task, the ACCC is activated to take over the ego-car's longitudinal control. In such a case, the ACCC tries to fulfill the learned preferences as much as possible. Alternatively, the driver can also activate the classical ACC to control the ego-car.*

Two separate experiments were designed for the ACCC's evaluation with this user story in mind. Each experiment focuses on simulating 300 one-way trips either from GS to CLZ or from CLZ to GS. Figure 5.28 indicates the process flow of each experiment.

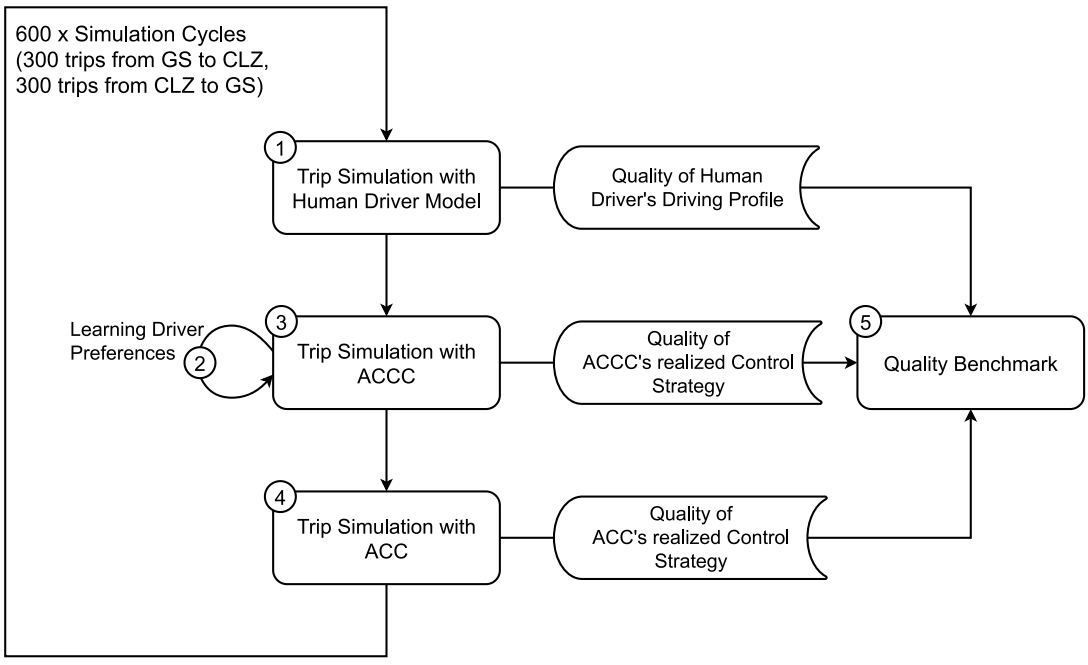


Figure 5.28: Process Flow of the Evaluation Work for the Performance Benchmark

As illustrated in Figure 5.28, multiple simulation cycles, including individual trip simulations with different candidate approaches using the human driver model, the ACCC, and the classical ACC, are planned. After each simulation with the human driver model, the ACCC would be triggered to learn the driver's preferences. After each trip is simulated with all three candidate approaches, a single simulation cycle is completed. Qualities of different performed control strategies will be calculated based on Equation (5.3) and compared to complete the performance benchmark. Thus, the best candidate approach can be identified.

5.6.4 Analysis

After designing the experiments, this dissertation has simulated the planned 600 trips with the help of the developed co-simulation platform to evaluate the performance of the implemented ACCC prototype. The evaluation results will be illustrated and analyzed in detail in this section.

Before introduction of the final evaluation results of the performance benchmark between the human driver model, the classical ACC, and the ACCC, it is meaningful first to evaluate the performance of applied individual machine learning algorithms (e.g., the Q-learning algorithm and the KDE) in the ACCC. Applying these two algorithms for planning the ego-car's driver-preferred driving strategy and predicting the preceding car's driving behaviors in the implementation of this dissertation is not a blind decision. Instead, this decision is based on the previous papers' research results [70][122], which are the basis of this dissertation. Thus, this dissertation did not separately evaluate the algorithms again.

Instead, the following sections will briefly summarize previous research work to explain why these two algorithms are selected for implementation in this dissertation. The final performance benchmark results between the manual driver, the classical ACC, and the ACCC based on the built co-simulation platform implemented in this dissertation will subsequently be introduced at the end of this section.

5.6.4.1 Performance of Planning Driving Strategy by Q-Learning Algorithm

The previous research [122] evaluated the Q-learning algorithm based on a simulation environment in three different test scenarios: (1) urban areas, (2) extra-urban areas, and

(3) motorways. Inspired by Porsche's "InnoDrive" (cf. Section 3.4), dynamic programming (DP) was selected as an alternative approach to complete the performance benchmark with the Q-learning algorithm. Since detailed concepts have been introduced in [123], this dissertation will not present all detail. Both candidate approaches in the benchmark were designed to plan an optimized driving strategy consisting of a location-based trajectory of set speed for a sequence of predefined intermediate geographical points on a given route, similar to the concept of this dissertation. Following the planned strategy, the ego-car aims to arrive at the destination under a given constraint of travel time by saving energy consumption as much as possible. Other traffic (e.g., a preceding obstacle) was ignored in the simulation.

The previous research [122] used a statistical driver model called "Move3F" to generate the human driver's driving behaviors in the simulation environment. A total of 900 trips (300 trips in urban areas, 300 trips in extra-urban areas, and 300 trips on motorways) were simulated. After each trip, the Q-learning and DP algorithms were (re-)trained to plan a driving strategy with the highest quality.

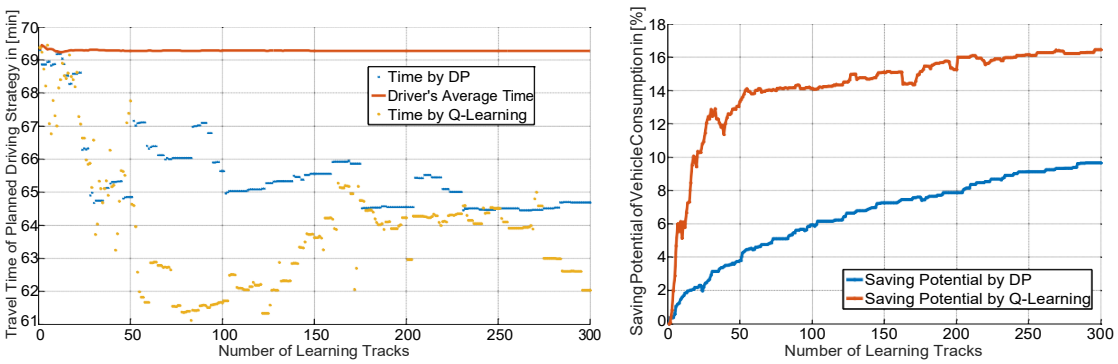


Figure 5.29: Performance Benchmark of Dynamic Programming (DP) and Q-learning (learning rate: 0.001, discount factor: 0.001) for Extra-Urban Areas [122]

Figure 5.29 shows an overview of the evaluation results for the test scenario of extra-urban areas. The left-side figure indicates that either the Q-learning or the DP algorithm guaranteed the fulfillment of the timing constraint, which was taken as the human driver's average travel time. The figure on the right shows that the ego-car could also save energy consumption by relying on the planned driving strategy. For the extra-urban scenario, almost 9.8% of the energy consumption can be saved using DP compared to

the driver's average profile. However, the Q-learning algorithm's saving potential is up to 16.5%, namely 6.7% higher than the DP algorithm.

Similar evaluation results were also found for the test scenarios of urban areas and motorways. Ater 300 learning iterations, the DP algorithm achieved an energy-saving potential of nearly 20% (cf. Figure 5.30, the blue line in the left-side figure) in the urban scenario. Compared to the DP algorithm, the Q-learning algorithm achieved a higher saving potential of 26%. In the motorway test scenario, the Q-learning algorithm also achieved a higher potential of 11% for saving energy consumption than the DP (6.5%).

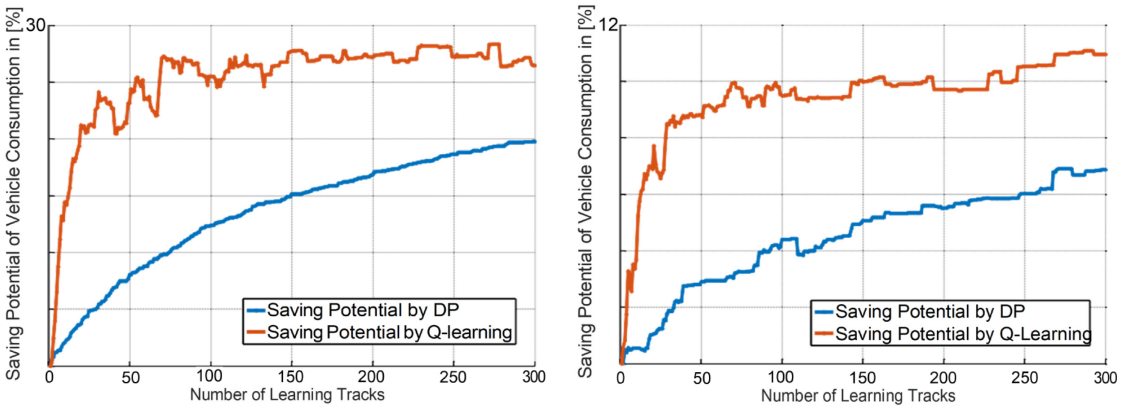


Figure 5.30: Performance Benchmark of Dynamic Programming (DP) and Q-learning (learning rate: 0.001, discount factor: 0.001) for Urban Areas and Motorways [122]

Evaluation results in previous research [122] indicated that the Q-learning algorithm is meaningful for planning the driving strategy with great optimization potential. Thus, this dissertation has used Q-learning again in the ACCC since one of its core functionalities is designed to plan the driver-preferred optimized driving strategy. The only difference compared to the previous research is that the cost function for the optimization process has changed. The same cost function introduced in Section 5.6.1 (cf. Equations (5.3)–(5.7)) was used in the Q-learning algorithm.

5.6.4.2 Performance in Predicting Preceding Car's Behaviors by Kernel Density Estimator (KDE)

Similar to the case of Q-learning, the decision to apply KDE for predicting the preceding car's driving behaviors in the ACCC is made based on another research work [70] as the basis of this dissertation. This previous research implemented four candidate

approaches: (1) KDE, (2) NARX (recurrent neural network), (3) clustering of driving styles with polynomial approximation, and (4) clustering of driving styles with the average profile.

For the candidate approaches (3) and (4), the recorded driving behaviors consisting of location-based speed trajectories of the preceding cars are firstly categorized into three classes using the ML-based clustering algorithm k-means, as illustrated on the left side of Figure 5.31 (green, red, and blue). The categorized speed profiles in each class can further be processed to determine a standard profile, which can subsequently be used for speed prediction. For example, in approach (4), all included speed profiles were used to derive an average speed profile. Instead, approach (3) used the polynomial approximation to derive the standard profile, as illustrated on the right side of Figure 5.31. During prediction, it is required to classify which category the current detected preceding car belongs to and thus decide which standard profile should be used to derive the future speed of the preceding car.

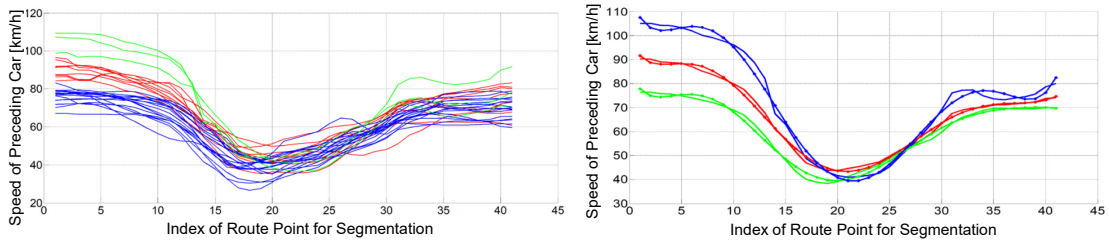


Figure 5.31: Clustering of Preceding Car Behaviors with k-means (left-side) and Polynomial Approximation and Average Profile (right-side) [70]

Unlike approaches (3) and (4), approaches (1) and (2) rely on a learned model of the preceding car's behaviors. In approach (2), a neural network (input: the preceding car's speed for previous route points, output: the preceding car's speed for the next route point) is trained based on the recorded speed data. In approach (1), the statistical density model is built based on the KDE algorithm. This dissertation will not introduce more details since detailed concepts have been published in previous research papers [71]. As illustrated in Figure 5.31, all approaches aim to predict a location-based speed trajectory. The locations are again determined by a sequence of predefined route points.

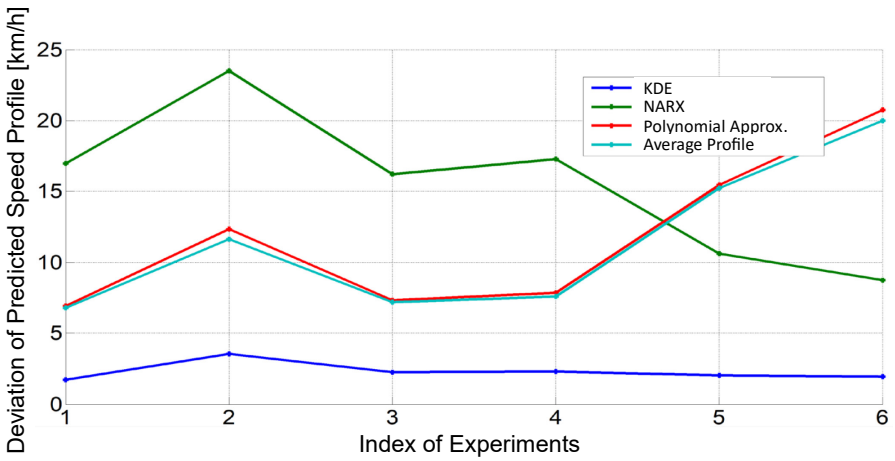


Figure 5.32: Performance of Candidate Approaches for One-Step Prediction of Preceding Car's Driving Behaviors [70]

Another point that needs to be emphasized is that the KDE algorithm considers three influencing factors (the preceding car's current acceleration, previous speed, and current speed) to predict future speed for the following route point. However, this dissertation reduces implementation complexity by using only one influencing factor: the preceding car's current speed. The evaluation results of the previous research [70] showed that the KDE has a better prediction performance than the other candidate approaches. It has significantly fewer predicted speed deviations in one-step prediction (i.e., predicting the preceding car's speed for a single following route point) than others.

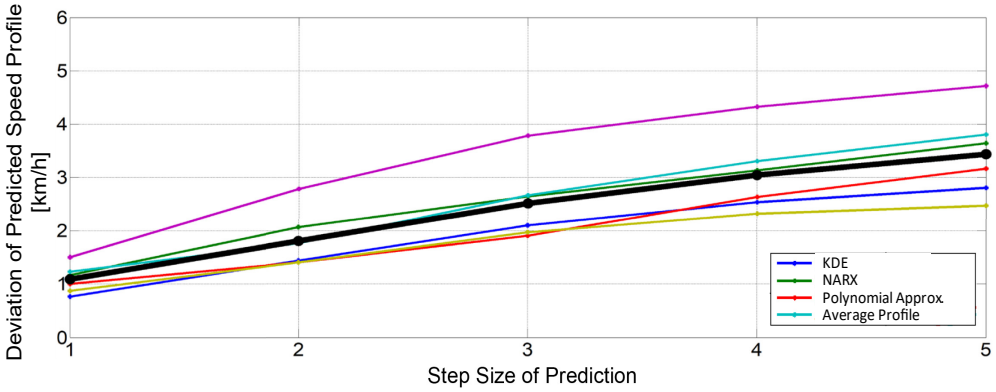


Figure 5.33: Performance of Candidate Prediction Approaches with Multiple Step Sizes of Prediction [70]

In addition, a benchmark of multi-step prediction was also performed in the previous research work. The results showed that the deviations increase along with the increasing

prediction horizon. However, the KDE still showed a great performance in the task of multi-step prediction, as illustrated in Figure 5.33[20]. For this reason, KDE was chosen as the prediction approach implemented in the ACCC of this dissertation for predicting the preceding car's speed.

5.6.4.3 Performance Benchmark between the Human Driver Model, ACC, and ACCC

After introducing the separate performance evaluation of the applied machine learning algorithms in the previous research works, the performance of the overall ACCC will be introduced in this section. As introduced in Figure 5.28, the ACCC's evaluation is completed based on a simulation of 600 trips on the federal highway B241 between the city Clausthal-Zellerfeld (CLZ) and Goslar (GS) in Germany (300 trips from CLZ to GS, 300 trips from GS to CLZ). Depending on the directions of the trips, they are named "GS2CLZ" and "CLZ2GS".

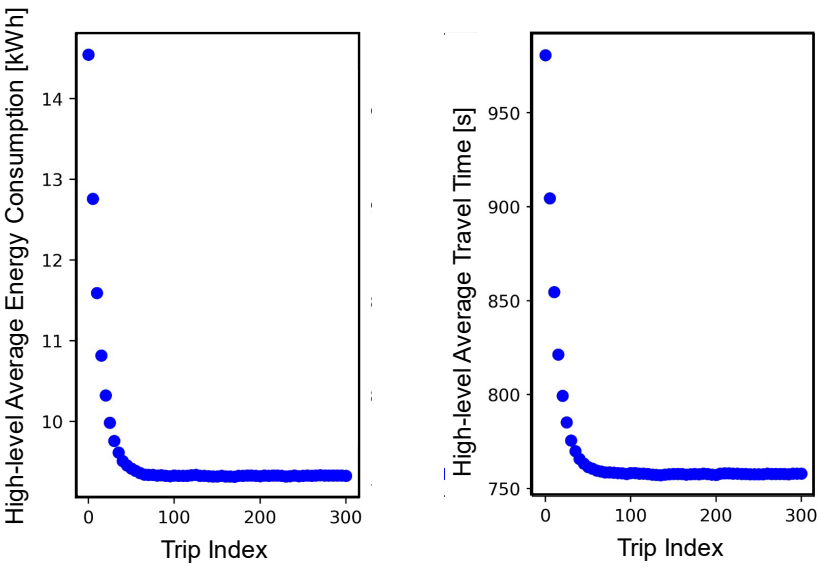


Figure 5.34: Learning of High-level Average Driver Profile in ACCC for the Trip "CLZ2GS"

Along with the simulation of these 600 trips, the ACCC learns the driver's average trip profiles. Figure 5.34 shows an example of this learning process, considering the 300 trips from CLZ to GS. The figure indicates that the initialized high average energy

[20] The black line without a label is the average deviation considering predicted speed profiles using all candidate approaches.

consumption and average travel time converged along with the learning iterations. Since the 50th trip, the average profiles begin to remain within a quite stable value range.

In addition to learning the driver's average profile, the ACCC is also designed to take over the ego-car's longitudinal control and guide it to the destination considering its planned optimized driving strategy and predicted preceding car's behaviors. After the simulation of all 600 trips, the qualities of performed strategies by the human driver model, the classical ACC, and the ACCC are quantitatively calculated with the help of the cost function introduced in Section 5.6.1 (cf. $Quality_{final}$ in Equation (5.3)).

The calculated final quality of the performed strategies is illustrated in Figure 5.35. It is indicated that the quality of all performed strategies was reduced during the simulation of the first 50 trips. Such a phenomenon arises because the high-level driver's average profile, taken as the set travel profile for the ACCC, has changed significantly due to the learning process illustrated in Figure 5.34.

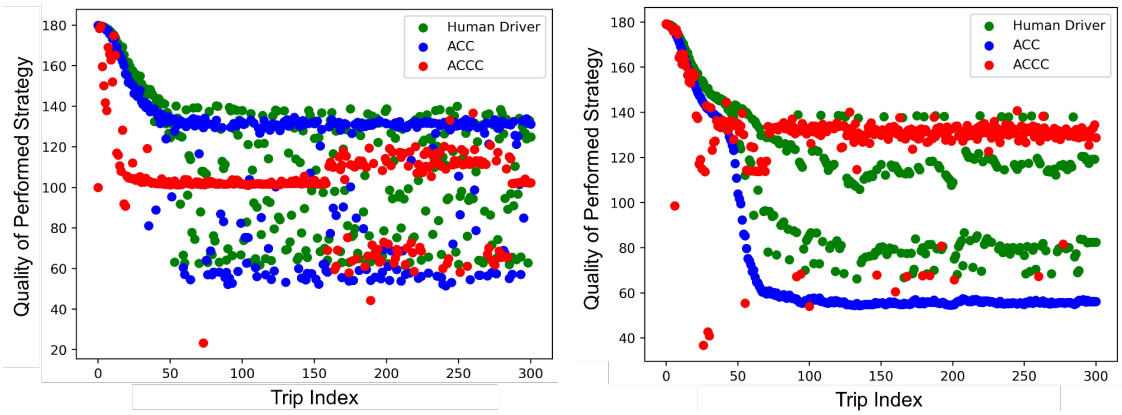


Figure 5.35: Quality Benchmark of Manual Driver, classical ACC, and ACCC for the Trip CLZ2GS (left side) and CLZ2GS (right side)

After that, the quality of the performed strategies performed by the human driver model, the classical ACC, and the ACCC changed due to the nondeterministic property of traffic simulation. However, the changes converge and stay within limited value ranges. For the trip "CLZ2GS", although the ACCC does not always realize a better-qualified driving strategy than the human driver and the classical ACC, it still has a clear potential for optimizing the driving strategy's quality. Compared to the human driver model and the classical ACC, the ACCC has achieved a better quality for around 35.2% of 300 trips.

The average quality of the human driver model and the classical ACC is quite similar: around 112.1 and 112.9. Instead, the average quality of the ACCC is lower than the others (around 100.2). However, for the trip “GS2CLZ”, the ACCC shows a significantly better quality profile than the human driver and the classical ACC (for around 98.5% of 300 trips). In this case, the average quality of the human driver and the classical ACC is around 111.2 and 72.9. Compared to them, the average quality of the ACCC is much higher and around 128.5.

Based on the discovery above, the proposed hypotheses summarized in Section 5.6.1 can be confirmed. The implemented ACCC can realize a higher performance quality of longitudinal control than the classical ACC and the human driver’s manual driving. Although the evaluation results for the trip “CLZ2GS” are not idealized, it does not mean that the ACCC works definitively worse than the classical ACC and the human driver. Since the ACCC is designed as a self-learning system, the illustrated potential of increasing the quality of longitudinal control has still not reached its limitation. By learning more diversified profiles of the human driver’s manual driving, the quality of the ACCC’s performed control strategy may further increase. However, such a plan requires a more comprehensive data recording for the human driver’s driving behaviors and is thus excluded from this dissertation's evaluation.

5.7 Summary

After evaluating the generalization capability of the proposed generic architecture style, it is meaningful to evaluate its feasibility in concrete applications. For this purpose, a further ACC variant called artificial cognitive cruise control (ACCC) is designed and implemented in this dissertation. This chapter focuses on the introduction to the ACCC.

First, some preliminary ideas about the system design of the ACCC were introduced in this chapter. Subsequently, the generic architecture style was applied to instantiate a system architecture. Subsystems like the physical system, the route-based adaptation unit, the route-segment-based adaptation unit, and the cycle-time-based control unit included in the system architecture were presented in detail. Different generic communication patterns are allocated to appropriate use cases to evaluate the dynamic system behaviors.

The designed ACCC was also implemented as a simplified prototype in this dissertation. The implemented prototype is subsequently evaluated within a co-simulation environment based on an anonymous human driver's previously recorded driving data. A benchmark compared to the classical ACC and the human driver was also included and introduced in this chapter. The benchmark results showed that the ACCC could learn and fulfill the driving preferences of the human driver. Compared to the standard ACC and the human driver, there is a significant performance improvement in satisfying the driving preferences of the human driver.

Since the driver during the ACC's operation is still required to participate in the driving task by steering the car, such a concept can massively increase the driver's reliance on the ACC and the performance of the overall semi-automated driving. In a more general sense, the concept of ACCC makes the driver more able to take over vehicle control in urgent cases that the (semi-)automated system cannot handle. Thus, the concept of such a driver-individual self-learning system may also be meaningful in developing the autonomous driving system with an automation level of L3+.

6 Conclusion

Chapter 6 aims to conclude the whole dissertation. A summary of the dissertation will first be first provided. Limitations of the dissertation are then discussed, identifying concrete pain points, and, if necessary, presenting corresponding potential approaches to eliminate them. Since this dissertation can inspire and contribute to far future work in other research directions, recommendations for future research are also included at the end of this chapter.

6.1 Summary of this Dissertation

This dissertation addresses the current problems and challenges in designing automatic control systems. Automatic control is a well-known technology applied almost everywhere. The automated control mechanisms behind automatic control systems have also progressively evolved from relying on traditional pure mechanical control elements to combining the mechanical with increasing numbers of electronic/electric control elements and accompanying software implementations. The complexity of automatic control systems grows along with expected increases in autonomy and adaptability to work appropriately under diversified operation conditions.

This dissertation has systematically reviewed the current concepts of automatic control systems based on concrete examples of the current ACC variants. This review has analyzed the architectures of different automatic control systems from the control theory viewpoint. A taxonomy of these concepts was identified in the dissertation. Additionally, a functional vision of the future ACC variants was postulated to derive issues facing the current concepts, covering two perspectives: (1) missing knowledge acquisition and adaptation and (2) limited system scalability against fixed boundary conditions (cf. Section 3.6). Since the current system architecture design approaches reached limits to the elimination of the previously mentioned issues, this dissertation focused on improving the architecture design of automatic control systems.

Along with integrating more complicated algorithms on higher levels of abstraction and their accompanying required computation mechanisms, automatic control systems are becoming so-called cyber-physical systems with heterogeneous computation

mechanisms. Additionally, they are acquiring properties of other software-intensive systems like the self-adaptive system from the research field of software engineering. Thus, this dissertation has focused on the architecture design of future automatic control systems designed as so-called self-adaptive cyber-physical systems. A new concept for such future automatic control systems called artificial cognitive control is defined in this dissertation to extend the previously mentioned taxonomy.

Against such a background, this dissertation has combined control theory and software engineering approaches and developed a generic architecture style for automatic control systems. The developed architecture style can be used to design different automatic control systems for diverse applications. A fundamental component structure with static construction is designed. Each fundamental component structure represents a node. Thus, an arbitrary networked system architecture can be constructed by connecting multiple nodes, following the design principle of so-called structural adaptation composition. The networked system architecture can be instantiated in more concrete examples like system architecture with multiple hierarchies. The generic architecture style also supports the application of different triggering mechanisms and generic communication patterns and paradigms.

In this dissertation, the current concepts of automatic control systems are used to evaluate the generalization capability of the developed generic architecture style. For this purpose, the derived logical architectures for the current automatic control systems are compared with architectures from the control theory viewpoint that focus more on control flow than system construction with components and accompanying responsibility assignment.

A concrete prototype called artificial cognitive cruise control (ACCC) was designed following the generic architecture style to evaluate the architecture style's feasibility. The previously mentioned issues regarding knowledge acquisition and adaptation and the limited system scalability are overcome by ACCC. Unlike current ACC variants on the market, ACCC learns the driving preferences of a single individual driver and satisfies this driver instead of satisfying different drivers simultaneously. In addition, it also has a memory ability to remember the previously experienced environmental driving context during the trip and learn the context as its experience.

Based on the evaluation results, ACCC has significantly improved performance compared to the human driver and the standard ACC on the market. It is emphasized that certain features of the ACCC are simplified to reduce the implementation complexity of this dissertation. For example, this dissertation only considers the preceding car as an example of the driving context. The implemented ACCC prototype can still be extended in the future, relying on the generic architecture style's benefit of great system scalability.

6.2 Limitations of this Dissertation

In this dissertation, the proposed generic architecture style has massively improved the architecture design of complicated automatic control systems in the future. In addition, the designed ACCC as a further ACC variant has shown significant performance improvement over the current ACC variant. However, there are always limitations in research work, which is also the case in this dissertation. In the following sections, the limitations of this dissertation will be discussed from different perspectives.

6.2.1 Limited Separation of Concerns in Knowledge Component of the Generic Architecture Style

The first limitation of this dissertation refers to the limited separation of concerns in the knowledge component ("K*") of the fundamental component structure defined in the generic architecture style. The generic architecture style derives a system architecture consisting of multiple networked nodes, each of which includes a fundamental component structure (cf. Section 4.2). In the fundamental component structure, the knowledge component is defined as a knowledge repository for storing the domain knowledge like the symptom, which includes structured data about observed facts of the physical system. As introduced earlier, the knowledge component is required by the other four components ("S*", "C*", "A*", and "I*"). In this sense, the knowledge component can be implemented as a reactive database or knowledge base that other components can access to fetch the required data or knowledge with an appropriate representation format.

However, the knowledge component also communicates other components by pushing domain knowledge to them, considering UC2 and UC4 from another perspective. In this case, the knowledge component remains proactive and triggers the communication itself. In this sense, the knowledge component plays the role of a knowledge management system responsible for knowledge synchronization across the overall fundamental component structure.

Based on the understanding above, it is clear that the knowledge component is currently designed as a single centralized point responsible for diversified knowledge-relevant processes. Such a design concept would make the responsibility assignment unclear and thus violate the general architecture design principle of separation of concerns. A concrete example can be found once the blackboard pattern is applied within UC4. Suppose the interpreting component cannot identify the current problem and requests support from the knowledge component. In such a case, the knowledge component ("K*") forwards the request to other knowledge components ("K^{*+1}", "K^{*-1}") in neighbor nodes and asks for their support. Thus, the knowledge component that requests the support plays two different roles of *controller* and *blackboard* simultaneously in the backboard pattern (cf. Figure 5.14). Such an unfavorable responsibility assignment makes the communication between the components more complicated.

In this dissertation, all knowledge-relevant responsibilities like storage and management are assigned to a single component. Following the concept of separation of concerns in software architecture design, these responsibilities may need to be separated in the future. For example, knowledge storage and management can be assigned to two different components. These two components must not be designed on the same level of abstraction as the components in the fundamental component structure. Instead, they could also be the subcomponents of the knowledge components. Thus, the case where a single component plays multiple roles simultaneously in the blackboard pattern can be eliminated. Generally, there is still great potential to further develop the presented generic architecture style.

6.2.2 Missing Impact Investigation on Applying Communication Architecture Pattern

Another limitation of this dissertation is applying generic communication architecture patterns and their underlying communication paradigms, as introduced in Section 4.2.5. In this dissertation, the choice of an appropriate generic communication architecture pattern for a certain use case (UC) is not limited in the current design of the generic architecture style. The developers must consider the requirements and constraints of their specific applications and decide the appropriate communication paradigm with the consideration of corresponding boundary conditions. However, different communication paradigms due to their features may also influence the final performance of an automatic control system. Thus, an impact investigation on applying communication architecture patterns and accompanying underlying paradigms is still missing.

Against such a background, it may be meaningful to perform a comprehensive case study to achieve such a target. Different applications of automatic control systems could be investigated and categorized, considering their boundary conditions, requirements, and constraints in the use cases. From another perspective, the features of the communication patterns and their underlying communication paradigms, including advantages and disadvantages, may also be analyzed. Thus, an allocation between the communication problems and accompanying solutions might be found and serve as a template solution. Thus, the developers can easily identify an appropriate communication pattern and apply it to design automatic control systems for concrete use cases.

6.2.3 Uncomprehensive Evaluation of Generic Architecture Style

In this dissertation, the generalization capability and feasibility of the generic architecture style are empirically evaluated. For example, the logical architectures derived from the architecture style are compared with accompanying architectures of current automatic control systems from the control theory viewpoint to check the architectural consistency. In addition, the architecture style is also applied to design the ACCC that serves as an empirical prototype to evaluate the feasibility of the architecture style.

Although this dissertation has empirically evaluated the generic architecture style, a more comprehensive evaluation can still be planned in the future. For example, the well-

known architecture tradeoff analysis method (ATAM) [59], designed to help determine an appropriate software system architecture by focusing on risks and sensitive pain points, may be helpful for a comprehensive evaluation.

In addition to applying standard methods to evaluate the architecture pattern, it may also be meaningful to evaluate the architecture style in more different applications. The ACC and its variants are taken as the main application examples in this dissertation. However, in reality, there are thousands of different applications requiring automatic control systems. Due to the diversity of these applications, it may also be helpful to evaluate the architecture style in applications with completely different boundary conditions to increase the plausibility of the evaluation results.

6.2.4 Missing Extensive Evaluation of Artificial Cognitive Cruise Control

The final limitation of this dissertation deals with the evaluation of the ACCC. As introduced in Section 5.6, the ACCC is designed as an advanced driver assistance system (ADAS) deployed on the vehicle. In this dissertation, the evaluation of the ACCC is completed within a co-simulation environment. Although fundamental functionalities of the ACCC have been evaluated, non-functional performance (e.g., against timing constraints) is still unknown. Thus, as a potential improvement in the future, a more extensive evaluation of the ACCC could be performed in reality.

For this purpose, the implemented system must be deployed on a real car or a comparable physical prototype environment. As introduced in Section 5.5, some code is currently implemented with programming language Python, which does not have great timing performance for real-life applications. Thus, the implementation still needs to be revised considering timing performance.

Another potential improvement for the evaluation deals with applying generic communication patterns and underlying paradigms. As is well known, the performance of different communication patterns cannot be evaluated well within a simulation environment since the boundary conditions in the simulation are not comparable with reality. Thus, an intensive evaluation of different communication patterns' influences on the ACCC's final performance could also be meaningful in future work.

6.3 Recommendation for Future Research

In the previous section, the limitations of this dissertation are introduced. Suggestions for further improvements to eliminate their limitations were also proposed. In addition to the limitations, this dissertation can also guide further research work in different directions. For this purpose, some recommendations with potential future research directions are identified in this section.

6.3.1 Architecture Design from the Viewpoint of Multi-Agent Systems

In this dissertation, the developed generic architecture style focuses on designing an automatic control system with a networked architecture consisting of a set of nodes. Each node includes a fundamental component structure comprising several standard components (“SICAP-K”). The connections of nodes are built following the design principle of structural adaptation composition, considering the node level of the overall architecture. It means that for every two connected nodes, there is always a “manager node” and a “managed node”, and the “manager node” is able to adapt the “managed node”.

In this dissertation, the design of the generic architecture style relies on software engineering approaches for the self-adaptive system. In other words, the architecture design focuses especially on the perspective of adaptation between nodes. Each node has a complete fundamental component structure, which means that it can work independently. Thus, each node can be seen as an independent subsystem in the overall system. They cooperate mutually to complete the overall control task. Thus, it is indicated that automatic control systems with such networked architectures acquire similar features as so-called multi-agent systems [107] [108].

With this understanding in mind, the design of nodes and their relationships could be diversified. The automatic control systems with networked architectures would acquire the properties of distributed artificial intelligence. For example, notions like deliberative and reactive agents or passive, active and cognitive agents have been proposed in related works of distributed artificial intelligence [96][128], and these can be applied in the future design of automatic control systems. Instead of only focusing on adaptation, the roles of nodes and their relationships can be designed differently. Thus, a meaningful

research direction for this dissertation in the future is combining software engineering approaches regarding multi-agent systems and control theory to further develop the generic architecture style.

6.3.2 Heterogeneous Knowledge Acquisition and Adaptation

The final recommendation for future research deals with the update process of domain knowledge. As presented earlier, the domain knowledge in the generic architecture style needs to be described in an appropriate uniform representation format. In this dissertation, only the case of homogenous domain knowledge is considered (cf. Section 2.2.7.4). This means that the knowledge representation and the allocations between the notions and their underlying meanings are specified in advance and remain static.

However, in reality, automatic control systems are increasingly relying on system connectivity. The systems would acquire high-level domain knowledge (e.g., by relying on linguistic methods direct from external domains). In this case, there is no guarantee that the knowledge representations and the allocations will always remain the same. Suppose the acquired domain knowledge is described in another format, or the same meaning is described with unknown notions. In such a case, identifying the underlying meaning and integrating the received knowledge on a higher semantic level into the system's knowledge base is still a great challenge for designing the automatic control system. Thus, heterogeneous knowledge acquisition and adaptation may be another meaningful research direction for future automatic control systems.

Fortunately, such high-level semantic fusion of heterogeneous knowledge is not a complete novelty in computer science. In web applications, the topic of integrating human knowledge represented by different natural languages is very common. Different mature approaches for so-called knowledge fusion based on the technologies such as ontology have been proposed [130][131]. In the future, it may be meaningful to integrate the capability of heterogeneous knowledge fusion into automatic control systems. Such a capability would particularly benefit the research work of the multi-agent system to realize a high-level distributed artificial intelligence.

Bibliography

[1] G. N. Saridis, “Toward the realization of intelligent machines ,” presented at the Proc. IEEE, 1979 , vol. 67, pp. 1115–1133.

[2] G. N. Saridist, “Analytic formulation of the principle of increasing precision with decreasing intelligence for intelligent machines ,” in *Automatica*, 1989, vol. 25, no. 3, pp. 461–467, 1989. doi : 10.1016/0005-1098(89)90016-2.

[3] J. S. Albus, R. Lumina, J. Fiala, and A. J. Wavering, “NASREM - The NASA/NBS standard reference model for telerobot control system architecture,” in *Proc. 20th Int. Symp. Ind. Robot.*, Oct. 1989.

[4] F. F. Ingrand, M. P. Georgeff, and A. S. Rao , and M. P. Georgeff, “An architecture for real-time reasoning and system control,” *IEEE Expert*, vol. 7, no. 6, pp. 33–44, 1992, doi: 10.1109/64.180407.

[5] A. Meystel, “Representation of descriptive knowledge for nested hierarchical controllers,”, *IEEE 27th Conf. Decis. and Control*, Dec. 1988. doi: 10.1109/CDC.1988.194639.

[6] S. V. Ilyukhin, T. A. Haley, and R. K. Singh, “A survey of automation practices in the food industry,” in *Food Control*, 2001, vol. 12, no. 5, pp. 285–296, 2001, doi: 10.1016/S0956-7135(01)00015-9.

[7] M. Dotoli, A. Fay, M. Miskowicz, and C. Seatzu, “A survey on advanced control approaches in factory automation,” in *IFAC-PapersOnLine*, 2015, vol. 8, no. 3, pp. 394–399, doi: 10.1016/j.ifacol.2015.06.113.

[8] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, “A survey of motion planning and control techniques for self-driving urban vehicles,” in *IEEE Trans. Intell. Veh.*, 2016, vol. 1, no. 1, pp. 33–55, doi: 10.1109/TIV.2016.2578706.

[9] B. Heinrich and W. Schneider, Grundlagen Regelungstechnik. Wiesbaden, Germany: Springer Fachmedien, 2019. doi: 10.1007/978-3-658-26741-4.

[10] K. Ogata, Modern Control Engineering, 5th ed. Amsterdam, Netherlands: Pearson Education Inc., 2002.

[11] A. Trächtler and J. Gausemeier, Eds., *Steigerung der Intelligenz mechatronischer Systeme*, Berlin/Heidelberg, Germany: Springer, 2018.

[12] D. E. Seborg, T. F. Edgar, and S. L. Shah, “Adaptive control strategies for process control : A survey,” in *AIChE*, 1986, vol. 32, no. 6, pp. 881–1056.

[13] A. Armenta, “Control automation: Data science for control systems,” 2021, vol. c, pp. 1–5.

[14] J. Chen and S. Hong, Eds., “Real-Time and Embedded Computing Systems and Applications,” in *9th International Conference, RTCSA*, 2003, Berlin/Heidelberg, Germany: Springer International Publishing, 2003.

[15] M. Gharbi, A. Koschel, A. Rausch, and G. Starke, *Basiswissen für Softwarearchitekten: Aus- und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture – Foundation Level*, 2nd ed. Heidelberg, Germany: dpunkt.verlag, 2015.

[16] L. Bass, P. C. Clements, and R. Kazman, Software Architecture in Practice. 1997.

[17] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide*, 2nd ed. Reading, MA, USA: Addison Wesley Longman, Inc., 2015.

[18] A. Meystel, “Multiresolutional architectures for autonomous systems with incomplete and inadequate knowledge representation,” in *Artificial Intelligence in Industrial Decision Making, Control and Automation. Microprocessor-Based and Intelligent Systems Engineering*, Dordrecht, Germany: Springer, pp. 159–223, 1995.

[19] K.-E. Årzén, “An architecture for expert system based feedback control *,” in *Automatica*, 1989, vol. 25, no. 6, pp. 813–827, doi: 10.1016/0005-1098(89)90050-2.

[20] K. Rehfeldt, “Entwurf einer Modellierungssprache für Cyber-Physische Systeme,” Master thesis, Institute for Informatics , Technische Universität Clausthal,Germany, 2015.

[21] A. Rausch, C. Bartelt, and K. Rehfelt, “Quo vadis cyber-physical systems research areas of cyber-physical ecosystems,” in *CTSE 2015: Proc. 1st Int*

Workshop Control Theory Softw. Eng., 2015, pp. 22–25, doi: 10.1145/2804337.2804341.

[22] S. Behere and M. Törngren, “A functional reference architecture for autonomous driving,” in *Inf. Softw. Technol.*, 2016, vol. 73, pp. 136–150, doi: 10.1016/j.infsof.2015.12.008.

[23] S. Behere and M. Törngren, “A functional architecture for autonomous driving,” in *WASA 2015 Proc. of the 2015 ACM Workshop on Automot. Softw. Architecture*, 2015, pp. 3–10, doi: 10.1145/2752489.2752491.

[24] M. C. Huebscher and J. A. McCann, “A survey of Autonomic Computing - Degrees, models, and applications,” in *ACM Comput. Surv.*, 2008, vol. 40, no. 3, pp. 1–31, doi: 10.1145/1380584.1380585.

[25] B. H. C. Cheng et al., “Software engineering for self-adaptive systems,” in *Lecture Notes in Computer Science*, 2009, vol. 5525, pp. 1–26.

[26] K. J. Åström and R. M. Murray, Feedback systems: An introduction for scientists and engineers, Princeton, NJ, USA: Princeton Univ. Press, 2012.

[27] W. S. Levine, *The control handbook: control system fundamentals*, 2nd ed. Evanston, IL, USA: Routledge. 2011. doi: 10.1201/b10383.

[28] P. Iwanek, “Systematik zur Steigerung der Intelligenz mechatronischer Systeme im Maschinen- und Anlagenbau,” Ph.D. dissertation , Faculty mechanical engineering, Universität Paderborn, Germany, 2017.

[29] H. P. Geering, *Regelungstechnik: Mathematische Grundlagen, Entwurfsmethoden, Beispiele*. Berlin/Heidelberg, Germany: Springer-Verlag,1994.

[30] K.-D. Tieste and O. Romberg, *Keine Panik vor Regelungstechnik: Erfolg und Spaß im Mystery-Fach des Ingenieurstudiums*, Wiesbaden, Germany: Springer Fachmedien, 2011.

[31] K. Miettinen, “Nonlinear multiobjective optimization,” in *International Series in Operations Research & Management Science*, vol. 12., Boston, MA, USA: Springer, 1988, doi: 10.1007/978-1-4615-5563-6.

[32] J. H. Kessler, M. Krüger, and A. Trächtler, “Continuous objective-based control for self-optimizing systems with changing operation modes,” *ECC 2014*, pp. 2096–2102, 2014, doi: 10.1109/ECC.2014.6862182.

[33] A. M. Meystel and J. S. Albus, *Intelligent systems : Architecture, design, and control*, Hoboken, NJ, USA: Wiley, 2001, ISBN: 978-0-471-19374-6.

[34] K. J. Astrom, “Toward intelligent control,” in *Control Syst. Mag.*, 1989, vol. 9, no. 3, pp. 60–64, doi: 10.1109/37.24813.

[35] K. Kawamura and S. Gordon, “From intelligent control to cognitive control,” *2006 World Autom. Congr.*, 2006, pp. 1–8, 2006, doi: 10.1109/WAC.2006.376003.

[36] T. Wong, M. Wagner, and C. Treude, “Self-Adaptive Systems: A systematic literature review across categories and domains,” 2020, arXiv: 2101.00125.

[37] P. J. Antsaklis, K. M. Passino, N. Dame, and S. J. Wang, “An introduction to autonomous control systems,” in *Proc. 5th IEEE Int. Symp. Intell. Control*, 1990, pp. 21–26.

[38] J. C. Fiala, “NIST technical note 1255: Manipulator servo level task decomposition,” Gaithersburg, MD, USA: National Institute of Standards and Technology, 1988, doi: 10.6028/nist.tn.1255.

[39] A. Meystel, “Planning in a hierarchical nested autonomous control system,” in *Proc. SPIE 0727, Mobile Robots I*, Feb 1986, doi: 10.1117/12.937784.

[40] A. Meystelt and E. Messinag, “The challenge of intelligent systems,” *Proc. 2000 IEEE Int. Symp. Intell. Control*, 2000, pp. 211–216, doi: 10.1109/ISIC.2000.882925.

[41] R. A. Brooks, “A robust layered control system for a mobile robot” Cambridge, MA, USA: MIT, 1985.

[42] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, “A survey of deep learning techniques for autonomous driving,” in *J. Field Robot.*, 2019, vol. 37, no. 3, pp. 362–386, doi: 10.1002/rob.21918.

[43] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, “An architecture for

autonomy," in *Int. J. Robot. Res.*, 1998, vol. 21, no. 2, pp. 1–38.

[44] H. Yavuz and A. Bradshaw, "A new conceptual approach to the design of hybrid control architecture for autonomous mobile robots," in *J. Intell. Robot. Syst.,* 2002, vol. 34, pp. 1–26, doi: 10.1023/A:1015522622034.

[45] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," in *IEEE Comput. Soc.*, Jan 2003, pp. 41–50.

[46] IBM, "An architectural blueprint for autonomic computing," White paper, 2005. https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf (accessed on 28th Apr. 2022).

[47] P. Oreizy et al., "An architecture-based approach to self-adaptive software," in *IEEE Intell. Syst. Appl.*, May 1999, vol. 14, no. 3, pp. 54–62, doi: 10.1109/5254.769885.

[48] T. Warnecke, K. Rehfeldt, and A. Rausch, "Managing communication paradigms with a dynamic adaptive middleware," *Adapt. 2018 10th Int. Conf. Adapt. Self-Adaptive Syst. Appl.*, Feb 2018, pp. 24–33.

[49] P. Arcaini, E. Riccobene, and P. Scandurra, "Modeling and analyzing MAPE-K feedback loops for self-adaptation," in *2015 IEEE/ACM 10th Int. Symp. Softw. Eng. Adapt. Self-Managing Syst.*, Florence, Italy, 2015, pp. 13–23, doi: 10.1109/SEAMS.2015.10.

[50] D. G. De La Iglesia and D. Weyns, "MAPE-K formal templates to rigorously design behaviors for self-adaptive systems," in *ACM Trans. Auton. Adapt. Syst.*, 2015, vol. 10, no. 3, pp. 1–31, doi: 10.1145/2724719.

[51] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, "Feedback control of computing systems," Hoboken, NJ, USA: Wiley, 2004.

[52] J. Kramer and J. Magee, "Self-managed systems: An architectural challenge," in *Futur. Softw. Eng. (FoSE)*, 2007, pp. 259–268, doi: 10.1109/FOSE.2007.19.

[53] J. Cámara, G. Moreno, and D. Garlan, "Reasoning about human participation in self-adaptive systems," in *10th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst.*

(SEAMS), 2015, pp. 146–156, doi: 10.1109/SEAMS.2015.14.

[54] H. Müller, M. Pezzè, and M. Shaw, “Visibility of control in adaptive systems,” in *Proc. 2nd Int. Workshop Ultra-Large-Scale Softw.-Intensive Syst. (ULSSIS’08)*, May 2008, pp. 23–26, doi: 10.1145/1370700.1370707.

[55] N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien, and R. Casallas, “DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems chapter,” in *Software Engineering for Self-Adaptive Systems II*, R. de Lemos, H. Giese, H. A. Müller, M. Shaw, Eds., 2013, pp. 265–293, doi: 10.1007/978-3-642-35813-5_11.

[56] A. S. Tanenbaum and M. Van Steen, *Distributed systems: Principles and paradigms*, Upper Saddle River, NJ, USA: Pearson, 2006.

[57] A. Schill and T. Springer, *Verteilte Systeme: Grundlagen und Basistechnologien*, New York, NY: Springer, 2011.

[58] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, patterns for concurrent and networked objects*, vol. 2, Hoboken, NJ, USA: Wiley, 2000.

[59] J. Ingeno, *Software architect’s handbook: Become a successful software architect by implementing effective architecture concepts*, Birmingham, UK: Packt, 2018.

[60] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2015.

[61] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-oriented software architecture: A pattern language for distributed computing*, vol. 4, Hoboken, NJ, USA: Wiley, 2007.

[62] Object Management Group, *CORBA component model specification*, April 2006. https://www.omg.org/spec/CCM/4.0/PDF (accessed on 28th Apr. 2022).

[63] F. Halsall, *Multimedia Communications*, ed. 2, Boston, MA, USA: Addison-Wesley, 2001.

[64] B. W. Wah, X. Su, and D. Lin, “A survey of error-concealment schemes for real-time audio and video transmissions over the Internet,” in *Proc. Int. Symp. Multimed. Softw. Eng.*, 2000, pp. 17–24, doi: 10.1109/MMSE.2000.897185.

[65] G. S. Blair and J.-B. Stefani, *Open distributed processing and multimedia*, Boston, MA, USA: Addison-Wesley, 1998.

[66] P. Lalanda, “Shared repository pattern,” in *Proc. 5th Pattern Lang. Programs Conf. (PLoP 1998)*, 1998, p. 10.

[67] E. Alpaydin, *Introduction to machine learning*, vol. 579, 2nd ed. Cambridge, MA, USA: MIT Press, 2010.

[68] C. J. C. H. Watkins and P. Dayan, “Technical note: Q-Learning,” *Mach. Learn.*, 1992, vol. 8, pp. 279–292, doi: 10.1023/A:1022676722315.

[69] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018, 2020.

[70] M. Zhang, K.-F. Storm, and A. Rausch, “Recognition and forecast of driving behavior based on self-learning algorithms,” in *Hybrid and Electric Vehicles*, 2017.

[71] E. Rutten, N. Marchand, and D. Simon, “Feedback control as MAPE-K loop in autonomic computing,” in *Software Engineering for Self-Adaptive Systems III*, de Lemos, R., Garlan, D., Ghezzi, C., Giese, H., Eds., Cham, Switzerland: Springer, 2015.

[72] K. Dar, “MAPE-K adaptation control loop,” Lecture Slides, 2012. https://www.uio.no/studier/emner/matnat/ifi/INF5360/v12/undervisningsmateriale/MAPE-K%20adap%20control%20loop.pdf (accessed 28th Apr. 2022).

[73] L. Xiao and F. Gao, “A comprehensive review of the development of adaptive cruise control systems,” in *Vehicle Syst. Dyn.*, 2010, vol. 48, no. 10, pp. 1167–1192, doi: 10.1080/00423110903365910.

[74] A. Vahidi and A. Eskandarian, “Research advances in intelligent collision avoidance and adaptive cruise control,” in *Transactions on Intelligent Transportation Systems*, 2003, vol. 4, no. 3, pp. 143–153, doi: 10.1109/TITS.2003.821292.

[75] F. Flehmig, F. Kästner. K. Knödel, and M. Knoop , “Eco-ACC für Elektro- und Hybridfahrzeuge,” in *Fahrerassistenzsysteme und Effiziente Antriebe*, W. Siebenpfeiffer, Ed., p. 11–17, 2015, doi: 10.1007/978-3-658-08161-4_2.

[76] Audi AG, “Audi annual report 2015,” 2015. https://www.audi.com/content/dam/gbp2/company/investor-relations/events-and-presentations/investor-presentations/englisch/2018/3.1_GB_2015_en.pdf (accessed on 28th Apr. 2022).

[77] B. Abendroth and R. Bruder, “Capabilities of humans for vehicle guidance,” in *Handbook of Driver Assistance Systems*, H. Winner, S. Hakuli, F. Lotz, and C. Singer, Eds., Cham, Switzerland: Springer, 2016, pp. 3–18, doi: 10.1007/978-3-319-12352-3_1.

[78] M. Roth et al., “Porsche InnoDrive – An Innovative Approach for the Future of Driving,” *Aachen Colloq. Automob. Engine Technol.*, 2011, pp. 1453–1467.

[79] I. D. Landau, R. Lozano, M. M’Saad, and A. Karimi, “Adaptive control: Algorithms, analysis and applications,” in *Communications and control engineering*, 2nd ed. London, UK: Springer, 2011, doi: 10.1007/978-0-85729-664-1.

[80] T. Radke, “Energieoptimale Längsführung von Kraftfahrzeugen durch Einsatz vorausschauender Fahrstrategien,“ in *Kalsruher Schriftenreihe Fahrzeugsystemtechnik*, vol. 19, 2013.

[81] I. Koglbauer, J. Holzinger, A. Eichberger, and C. Lex, “Drivers’ interaction with adaptive cruise control on dry and snowy roads with various tire-road grip potentials,” *J. Adv. Transp.*, 2017, vol. 2017, no. 549683,7 doi: 10.1155/2017/5496837.

[82] B. D. Seppelt and J. D. Lee, “Making adaptive cruise control (ACC) limits visible,” in *Int. J. Human-Comp. Stud.*, 2006, vol. 65, no. 3, pp. 192–205, doi: 10.1016/j.ijhcs.2006.10.001.

[83] B. D. Seppelt, F. N. Lees, and J. D. Lee, “Driver distraction and reliance: Adaptive cruise control in the context of sensor reliability and algorithm limits,” in *Proc. 3rd International Driving Symp. of Human Factors in Driv. Assess., Training*

and Vehicle Design, 2005, vol. 3, pp. 255–261, doi: 10.17077/drivingassessment.1168.

[84] T. Seyffarth, “Bildbasierte Abstandsregelung für Kraftfahrzeuge,” Ph.D. dissertation, Dept. IT, TU München, München, Germany, 2012. https://mediatum.ub.tum.de/doc/1085164/491990.pdf (accessed on 28th Apr. 2022).

[85] S. Birch, “Honda introduces ‘industry first’ intelligent adaptive cruise control,” 2015. https://www.sae.org/news/2015/01/honda-introduces-industry-first-intelligent-adaptive-cruise-control#:~:text=Automatic%20crash%20avoidance%20and%20braking,cutting%2Din%20and%20endangering%20another (accessed on 28th Apr. 2022).

[86] G. R. Widmann et al., “Comparison of lidar-based and radar-based adaptive cruise control systems,” in *SAE Tech. Pap.*, pp. 3–4, 2000, doi: 10.4271/2000-01-0345.

[87] A. A. D. Medeiros, “A survey of control architectures for autonomous mobile robots,” in *J. Brazilian Comput. Soc.*, 1998, pp. 1–12, doi: 10.1590/S0104-65001998000100004.

[88] K. J. Hunt, D. Sbarbaro, R. Żbikowski, and P. J. Gawthrop, “Neural networks for control systems - A survey,” in *Automatica*, 1992, vol. 28, no. 6, pp. 1083–1112, doi: 10.1016/0005-1098(92)90053-I.

[89] G. Han, W. Fu, W. Wang, and Z. Wu, “The lateral tracking control for the intelligent vehicle based on adaptive pid neural network,” in *Sensors*, 2017, vol. 17, no. 6, p. 1244, doi: 10.3390/s17061244.

[90] C. Wilson, F. Marchetti, M. Di Carlo, A. Riccardi, and E. Minisci, “Intelligent control: A taxonomy,” in *8th Int. Conf. Syst. Control*, 2019, p. 333–339, doi: 10.1109/ICSC47195.2019.8950603.

[91] T. Chakraborty, S. Yamaguchi, and S. K. Datta, “Sensor fusion and adaptive cruise control for self-driving platoon,” in *2018 IEEE 7th Glob. Conf. Consum. Electron. GCCE*, 2018, pp. 634–635, doi: 10.1109/GCCE.2018.8574639.

[92] B.-J. Chang, Y.-L. Tsai, and Y.-H. Liang, “Platoon-based cooperative adaptive cruise control for achieving active safe driving through mobile vehicular cloud computing,” in *Wirel. Pers. Commun.*, 2017, vol. 97, pp. 5455–5481, doi: 10.1007/s11277-017-4789-8.

[93] R.-H. Huang, B.-J. Chang, Y.-L. Tsai, and Y.-H. Liang, “Mobile edge computing-based vehicular cloud of cooperative adaptive driving for platooning autonomous self-driving,” in *2017 IEEE 7th Int. Symp. Cloud Serv. Comput. (SC2)*, 2017, pp. 32–39, doi: 10.1109/SC2.2017.13.

[94] B.-J. Chang, R.-H. Hwang, Y.-L. Tsai, B.-H. Yu, and Y.-H. Liang, “Cooperative adaptive driving for platooning autonomous self-driving based on edge computing,” in *Int. J. Appl. Math. Comput. Sci.*, 2019, vol. 29, no. 2, pp. 213–225, doi: 10.2478/amcs-2019-0016.

[95] R. W. Brennan, X. Zhang, Y. Xu, and D. H. Norrie, “A reconfigurable concurrent function block model and its implementation in real-time Java,” in *Integr. Comput. Aided. Eng.*, 2002, vol. 9, no. 3, pp. 263–279, doi: 10.3233/ica-2002-9306.

[96] R. W. Brennan, “Toward real-time distributed intelligent control: A survey of research themes and applications,” *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.*, 2007, vol. 37, no. 5, pp. 744–765, doi: 10.1109/TSMCC.2007.900670.

[97] E. Gat, “On Three-Layer Architectures,” in *Artif. Intell. Mob. Robot.*, 1997, pp. 195–210, doi: 10.1.1.165.5283.

[98] E. Gat, “Three-layer architectures,” in *Artificial intelligence and mobile robots: Case studies of successful robot systems*, 1998, pp. 195–210.

[99] F. Qureshi, D. Terzopoulos, and R. Gillett, “The cognitive controller: A hybrid, deliberative/reactive control architecture for autonomous robots,” in *Innov. in Appl. Artif. Intell. (IEA/AIE)*, 2004, pp. 1102–1111, doi: 10.1007/978-3-540-24677-0_113.

[100] F. M. Adolf and F. Thielecket, “A sequence control system for onboard mission management of an unmanned helicopter,” in *AIAA InfoTech Aerosp. Conf. Exhib.*, 2007, pp. 594–605, doi: 10.2514/6.2007-2769.

[101] J. Rasmussen, “Skills, rules, and knowledge; signals, signs, and symbols, and other distinctions in human performance models,” in *IEEE Trans. Syst. Man Cybern.*, 1983, vol. SMC-13, no. 3, pp. 257–266, doi: 10.1109/TSMC.1983.6313160.

[102] E. Ahle and D. Söffker, “Interaction of intelligent and autonomous systems - part II: Realization of cognitive technical systems,” in *Math. Comput. Model. Dyn. Syst.*, 2008, vol. 14, no. 4, pp. 319–339, doi: 10.1080/13873950801983852.

[103] A. A. Zheltoukhov and L. A. Stankevich, “A survey of control architectures for autonomous mobile robots,” in *2017 IEEE Russia Section Young Researchers in Elect.l and Electron. Eng. Conf. (ElConRus)*, 2017, pp. 1094–1099, doi: 10.1109/EIConRus.2017.7910746.

[104] Lexico.com, “Cognition,” Lexico.com. https://www.lexico.com/definition/cognition (accessed 28th Apr. 2022).

[105] M. M. Botvinick, T. S. Braver, D. M. Barch, C. S. Carter, and J. D. Cohen, “Conflict monitoring and cognitive control,” in *Psychol. Rev.*, 2001, vol. 108, no. 3, doi: 10.1037/0033-295x.108.3.624.

[106] K. R. Hammond and D. A. Summers, “Cognitive control,” in *Psychol. Rev.*, 1972, vol. 79, no. 1, pp. 58–67, doi: 10.1037/h0031851.

[107] M. Brass, Jan Derrfuss, B. Forstmann, and D. Y. von Cramon, “The role of the inferior frontal junction area in cognitive control,” in *Trends Cogn. Sci.*, 2005, vol. 9, no. 7, pp. 312–314, doi: 10.1016/j.tics.2005.05.001.

[108] H. Feldman and K. J. Friston, “Attention, uncertainty, and free-energy,” in *Front. Hum. Neurosci.*, 2010, vol. 4, no. December, pp. 1–23, doi: 10.3389/fnhum.2010.00215.

[109] C. E. Shannon, “A mathematical theory of communication,” in *Bell Syst. Tech. J.*, 1948, vol. 27, no. 4, pp. 623–656, doi: 10.1002/j.1538-7305.1948.tb00917.x.

[110] C. E. Shannon and W. Weaver, *The mathematical theory of communication*, Urbana, IL, USA: Univ. of Illinois Press, 1964.

[111] S. Haykin, M. Fatemi, P. Setoodeh, and Y. Xue, “Cognitive control,” in *Proc. IEEE*,

2012, vol. 100, no. 12, pp. 3156–3169, doi: 10.1109/JPROC.2012.2215773.

[112] A. Filieri et al., “Software engineering meets control theory,” *2015 IEEE/ACM 10th Int. Symp. Softw. Eng. Adapt. Self-Managing Syst. (SEAMS)*, 2015, pp. 71–82, doi: 10.1109/SEAMS.2015.12.

[113] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, “Control-theoretical software adaptation: A systematic literature review,” in *IEEE Transactions on Software Engineering*, 2016, vol. 44, no.8, pp. 784–810, doi: 10.1109/TSE.2017.2704579.

[114] T. F. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu, “Introduction to control theory and its application to computing systems,” in *Performance Modeling and Engineering*, Z. Liu, C. H. Yia, Eds., Boston, MA, USA: Springer, pp. 185–215, 2008.

[115] C. Pahl and P. Jamshidi, “Software architecture for the Cloud-A Roadmap Towards Control-Theoretic, Model-Based Cloud Architecture,” in *Software Architecture*, D. Weyns, R. Mirandola, and I. Crnkovic, Eds., Cham, Switzerland: Springer, 2015, doi: 10.1007/978-3-319-23727-5_17.

[116] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada, “Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures,” in *2016 12th Int. ACM SIGSOFT Conf. Qual. Softw. Archit. (QoSA)*, 2016, pp. 70–79, doi: 10.1109/QoSA.2016.13.

[117] C. Kunkel, “Tödlicher Unfall : Robotaxi hatte Software-Fehler,” in *Süddeutsche Zeitung*, 2019, pp. 1–7, https://www.sueddeutsche.de/auto/uber-unfall-robotaxi-amerika-ursache-1.4670087 (accessed 28th Apr. 2022).

[118] D. Lee, “Google self-driving car hits a bus,” in *BBC News*, 2016, https://www.bbc.com/news/technology-35692845 (accessed 28th Apr. 2022).

[119] A. J. Hawkins, “Google’s ‘worst’ self-driving accident was still a human’s fault,” in *The Verge*, 2016, https://www.theverge.com/2016/9/26/13062214/google-self-driving-car-crash-accident-fault (accessed on 28th Apr. 2022).

[120] C. Zhang, W. Allafi, Q. Dinh, P. Ascencio, and J. Marco, “Online estimation of

battery equivalent circuit model parameters and state of charge using decoupled least squares technique," in *Energy*, 2018, vol. 142, pp. 678–688, doi: 10.1016/j.energy.2017.10.043.

[121] S. Breuer and A. Rohrbach-Kerl, *Fahrzeugdynamik: Mechanik des bewegten Fahrzeugs*, Wiesbaden, Germany: Springer Vieweg, 2015.

[122] M. Zhang, K.-F. Storm, and A. Rausch, "Enhancement of driving strategy of electric vehicle by consideration of individual driver intention," in *30th Int. Electr. Veh. Symp. Exhib.*, 2017, pp. 1–12.

[123] P. F. and J. V. Josef Nilsson, Niklas Strand, "Driver performance in the presence of adaptive cruise control related failures: Implications for safety analysis and fault tolerance," in *2013 43rd Annual IEEE/IFIP Conf. Dependable Syst. and Networks Workshop (DSN-W)*, 2013, p. 1–10, doi: 10.1109/DSNW.2013.6615531.

[124] K. J. Åström and T. Hägglund, *PID controllers: theory, design and tuning*, 2nd ed. Research Triangle Park, NC: USA: Instrument Society of America, 1995.

[125] S. Krauss, "Microscopic modeling of traffic flow: Investigation of collision-free vehicle dynamics," Forschungsanstalt fuer Luft - und Raumfahrt e.V., Cologne, Germany, Rep. DLR-FB-98-08, 1998.

[126] J. Erdmann, "Lane-changing model in SUMO," presented at *SUMO2014*, 2014, pp. 77–88.

[127] M. Treiber, A. Hennecke, and D. Helbing, "Congested traffic states in empirical observations and microscopic simulations," in *Phys. Rev. E*, 2000, vol. 62, no. 2, pp. 1805–1824, doi: 10.1103/PhysRevE.62.1805.

[128] Y. Kubera, P. Mathieu, and S. Picault, "Everything can be agent!," in *9th Int. Conf. Auton. Agents Multiagent Syst. (AAMAS 2010)*, 2010, vol. 1–3, pp. 1547–1548.

[129] M. Niazi and A. Hussain, "Agent-based computing from multi-agent systems to agent-based models: A visual survey," in *Scientometrics*, 2011, vol. 89, pp. 479–499, doi: 10.1007/s11192-011-0468-9.

[130] X. Dong et al., "Knowledge vault: A web-scale approach to probabilistic

knowledge fusion," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min.*, 2014, pp. 601–610, doi: 10.1145/2623330.2623623.

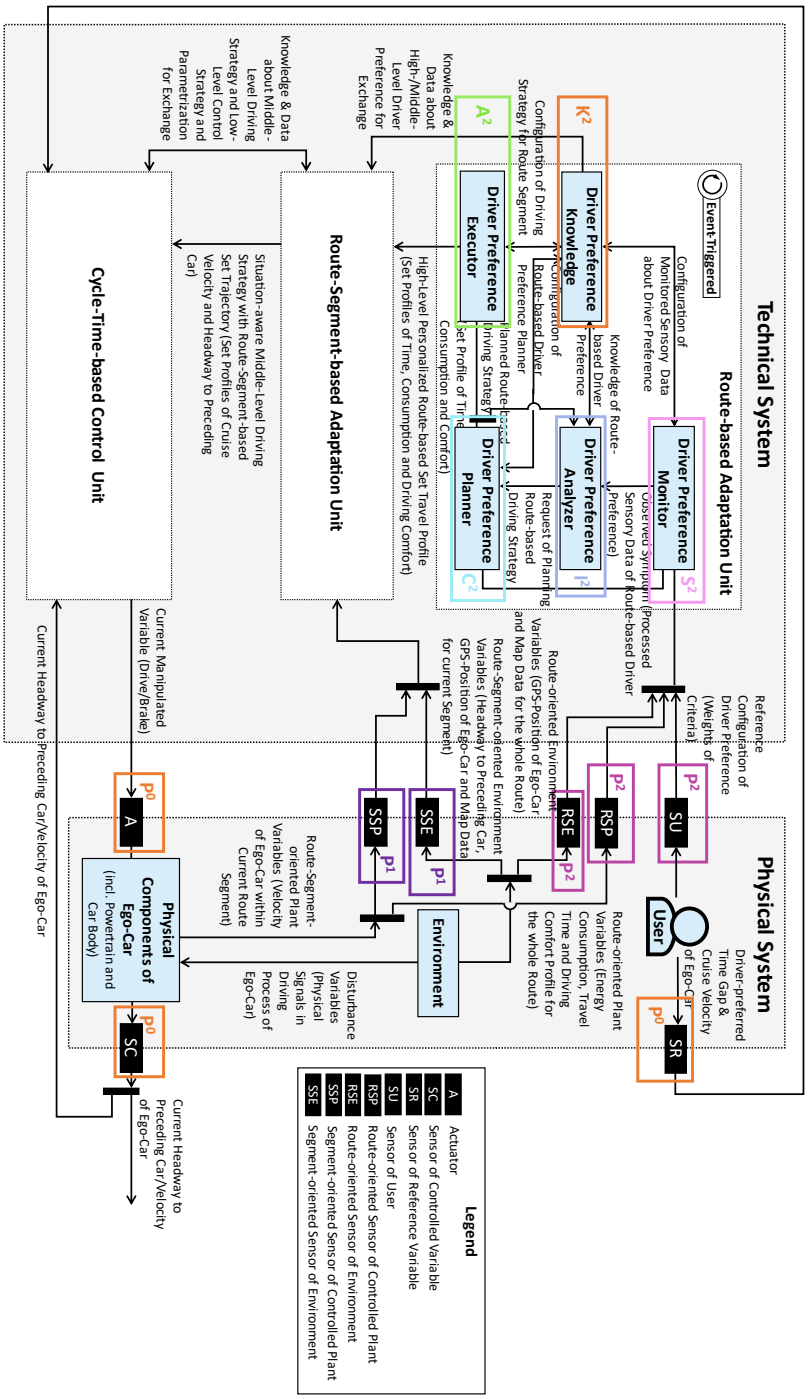
[131] T.-T. Kuo, S.-S. Tseng, and Y.-T. Lin, "Ontology-based knowledge fusion framework using graph partitioning," in *Developments in Applied Artificial Intelligence (IEA/AIE 2003), Lect. Notes Artif. Intell*, vol. 2718, Chung, P.W.H., Hinde, C., Ali, M., Eds., Berlin/Heidelberg, Germany: Springer, pp. 11–20, 2003, doi: 10.1007/3-540-45034-3_2.

[132] Y. Li and H. T. Bi, "Vehicle driving comfort analysis for a cable-stayed bridge considering vehicle bridge coupled vibration," in *2010 WASE Int. Conf. Inf. Eng. (ICIE)*, 2010, vol. 3, pp. 422–425, doi: 10.1109/ICIE.2010.278.

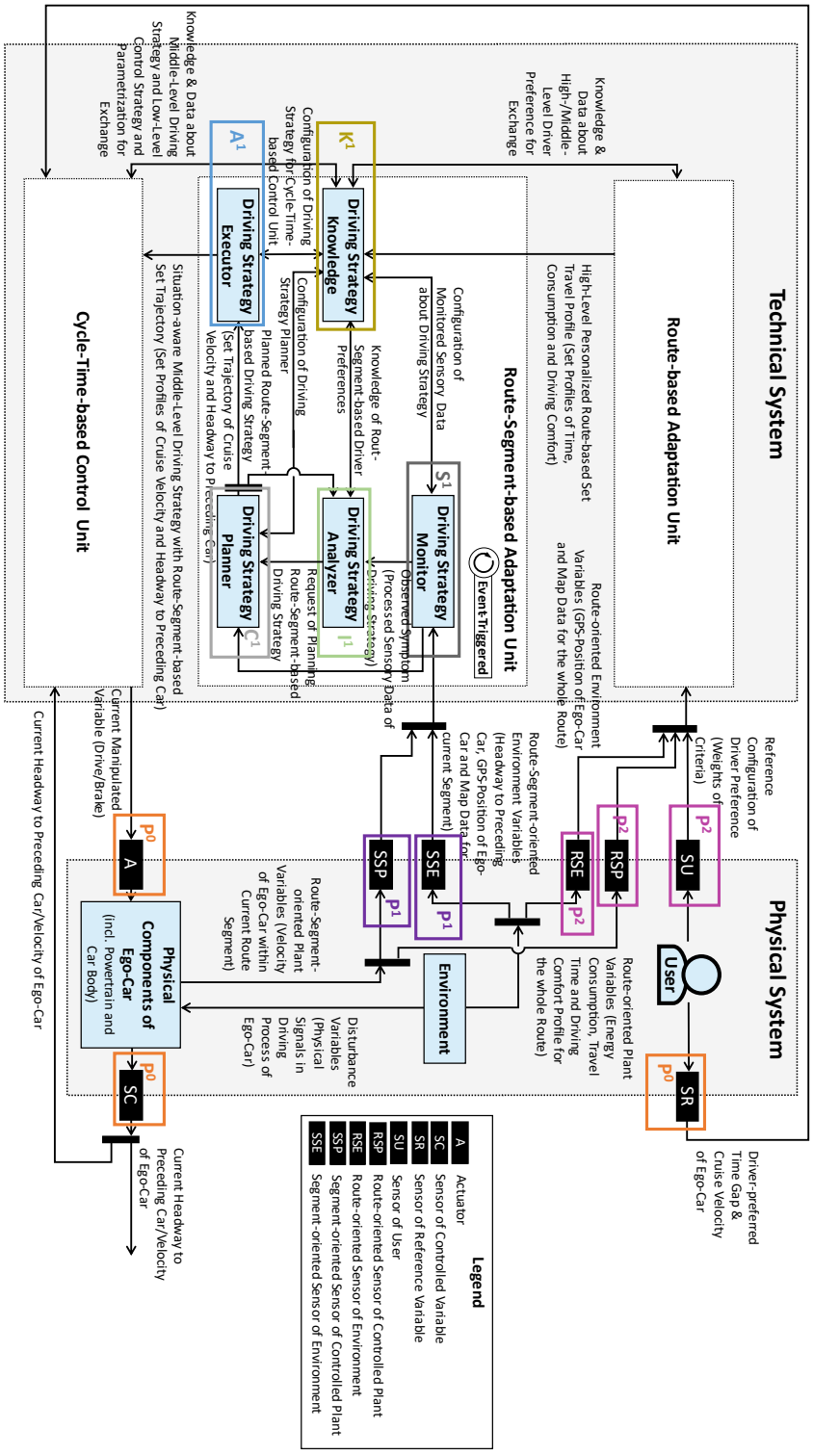
[133] Y. Luo, T. Chen, S. Zhang, and K. Li, "Intelligent hybrid electric vehicle acc with coordinated control of tracking ability, fuel economy, and ride comfort," in *IEEE Trans. Intell. Transp. Syst.*, 2015, vol. 16, no. 4, pp. 2303–2308, doi: 10.1109/TITS.2014.2387356.

Appendix

A.1 Detailed View of Route-Based Adaptation Unit in Artificial Cognitive Cruise Control



A.2 Detailed View of Route-Segment-Based Adaptation Unit in Artificial Cognitive Cruise Control



A.3 Detailed View of Cycle-Time-Based Control Unit in Artificial Cognitive Cruise Control