Chapter 4
AUTOSAR Standard

Darko Durisic
Volvo Car Group, Gothenburg, Sweden

Abstract In this chapter, we describe the role of the AUTOSAR (AUTomotive Open System ARchitecture) standard in the development of automotive system architectures. AUTOSAR defines the reference architecture and methodology for the development of automotive software systems, and provides the language (meta-model) for their architectural models. It also specifies the architectural modules and functionality of the middleware layer known as the *basic software*. We start by describing the layers of the AUTOSAR reference architecture. We then describe the proposed development methodology by identifying major roles in the automotive development process and the artifacts they produce, with examples of each artifact. We follow up by explaining the role of the AUTOSAR meta-model in the development process and show examples of the architectural models that instantiate this meta-model. We also explain the use of the AUTOSAR meta-model for configuring basic software modules. We conclude the chapter by showing trends in the evolution of the AUTOSAR standard and reflect on its future role in the automotive domain.

4.1 Introduction

The architecture of automotive software systems, as software-intensive systems, can be seen from different views, as presented in Sect. 2.7 and described in more detail by Kruchten in the *4+1 architectural view model* [Kru95]. Two of these architectural views deserve special attention in this chapter, namely the logical and the physical views, so we describe them briefly here as well.

Logical architecture of automotive software systems is responsible for defining and structuring high-level vehicle functionalities such as auto-braking when a pedestrian is detected on the vehicle's trajectory. These functionalities are usually realized by a number of logical software components, e.g., the *PedestrianSensor* component detects a pedestrian and requests full auto-brake from the *BrakeControl* component. These components communicate by exchanging information, e.g., about the pedestrian detected in front of the vehicle. Based on the types of functionalities they realize, logical software components are usually grouped into subsystems that in turn are grouped into logical domains, e.g., active safety and powertrain.

M. Staron, *Automotive Software Architectures*,
DOI 10.1007/978-3-319-58610-6_4

The physical architecture of automotive software systems is usually distributed over a number of computers (today usually more than 100) referred to as Electronic Control Units (ECUs). ECUs are connected via electronic buses of different types (e.g., Can, FlexRay and Ethernet) and are responsible for executing one or several high-level vehicle functionalities defined in the logical architecture. This is done by allocating logical software components responsible for realizing these functionalities to ECUs, thereby transforming them into runnable ECU application software components. Each logical software component is allocated to at least one ECU.

Apart from the physical system architecture that consists of a number of ECUs, each ECU has its own physical architecture that consists of the following main parts:

- Application software that consists of a number of allocated software components and is responsible for executing vehicle functionalities realized by this ECU, e.g., detecting pedestrians on the vehicle's trajectory.
- Middleware software responsible for providing services to the application software, e.g., transmission/reception of data on the electronic buses, and tracking diagnostic errors.
- Hardware that includes a number of drivers responsible for controlling different hardware units, e.g., electronic buses and the CPU of the ECU.

The development of the logical and physical architectural views of automotive software systems and their ECUs is mostly done following the MDA (Model-Driven Architecture) approach [Obj14]. This means that the logical and physical system architecture and the physical ECU architecture are described by means of architectural models. Looking into the automotive architectural design from the process point of view, car manufacturers (OEMs, Original Equipment Manufacturers) are commonly responsible for the logical and physical design of the system, while a hierarchy of suppliers is responsible for the physical design of specific ECUs, implementation of their application and middleware software and the necessary hardware [BKPS07].

In order to facilitate this distributed design and development of automotive software systems and their architectural components, the AUTOSAR (AUTomotive Open Systems ARchitecture) standard was introduced in 2003 as a joint partnership of automotive OEMs and their software and hardware vendors. Today, AUTOSAR consists of more than 150 global partners [AUT16a] and is therefore considered a de facto standard in the automotive domain. AUTOSAR is built upon the following major objectives:

1. Standardization of the reference ECU architecture and its layers. This increases the reusability of application software components in different car projects (within one or multiple OEMs) developed by the same software suppliers.
2. Standardization of the development methodology. This enables collaboration between a number of different parties (OEMs and a hierarchy of suppliers) in the software development process for all ECUs in the system.

3. Standardization of the language (meta-model) for architectural models of the system/ECUs. This enables a smooth exchange of architectural models between different modeling tools used by different parties in the development process.
4. Standardization of ECU middleware (basic software, BSW) architecture and functionality. This allows engineers from OEMs to focus on the design and implementation of high-level vehicle functionalities that can, in contrast to ECU middleware, create competitive advantage.

In the next four sections (4.2–4.6), we show how AUTOSAR achieves each one of these four objectives. In Sect. 4.6, we analyze trends in the evolution of AUTOSAR and how it can be measured. In Sect. 4.7, we present current initiatives regarding the future role of AUTOSAR in the development of automotive software systems. Finally, in the last two sections (4.8 and 4.9), we provide guidelines for further readings and conclude this chapter with a brief summary.

4.2 AUTOSAR Reference Architecture

The architectural design of ECU software based on AUTOSAR is done according to the three-layer architecture that is built upon the ECU hardware layer, as presented in Fig. 4.1.

The first layer, *Application software*, consists of a number of software components that realize a set of vehicle functionalities by exchanging data using interfaces

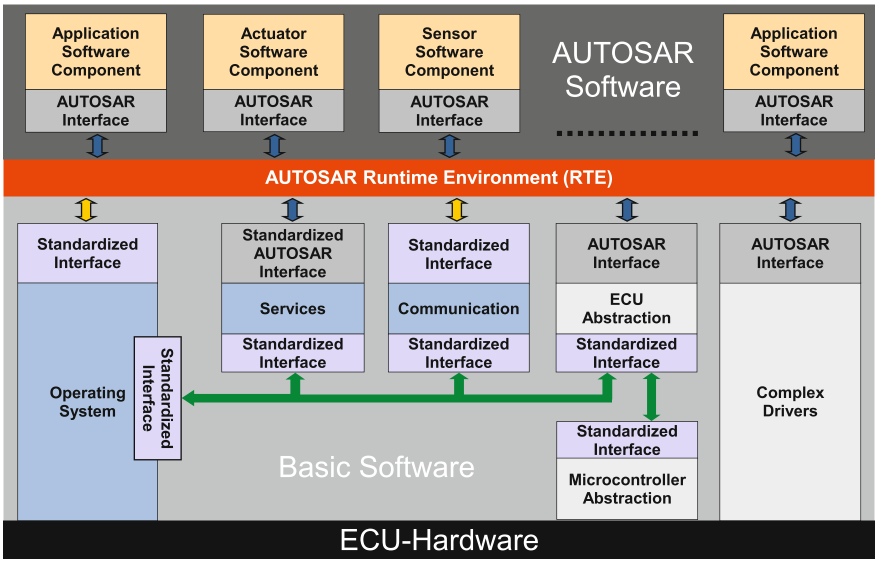

Fig. 4.1 AUTOSAR layered software architecture [AUT16g]

defined on these components (referred to as ports). This layer is based on the logical architectural design of the system. The second layer, *Run-time environment* (RTE), controls the communication between software components, abstracting the fact that they may be allocated onto the same or different ECUs. This layer is usually generated automatically, based on the software component interfaces. If the software components are allocated to different ECUs, transmission of the respective signals on the electronic buses is needed, which is done by the third layer (*Basic software*).

The *Basic software* layer consists of a number of BSW modules and it is responsible for the non-application-related ECU functionalities. One of the most important basic software functionalities is the *Communication* between ECUs, i.e., signal exchange. It consists of BSW modules such as *COM* (Communication Manager) that are responsible for signal transmission and reception. However, AUTOSAR basic software also provides a number of *Services* to the *Application software* layer, e.g., diagnostics realized by *DEM* (Diagnostic Event Manager) and *DCM* (Diagnostic Communication Manager) modules that are responsible for logging error events and transmitting diagnostic messages, respectively, and the *Operating System* for scheduling ECU runnables. The majority of BSW modules are configured automatically, based on the architectural models of the physical system [LH09], e.g., periodic transmission of a set of signals packed into frames on a specific electronic bus.

Communication between higher-level functionalities of ECU *Basic software* and drivers controlling the ECU hardware realized by the *Microcontroler Abstraction* BSW modules is done by the *ECU Abstraction* BSW modules, e.g., bus interface modules such as *CanIf*, which is responsible for the transmission of frames containing signals on the CAN bus. Finally, AUTOSAR provides the possibility for application software components to communicate directly with hardware, thus bypassing the layers of the AUTOSAR software architecture, by means of custom implementations of *Complex Drivers*. This approach is, however, considered non-standardized.

Apart from the *Complex Drivers*, *RTE* and modules of the *Basic Software* layer are completely standardized by AUTOSAR, i.e., AUTOSAR provides detailed functional specifications for each module. This standardization, together with the clear distinction between the *Application software*, *RTE* and *Basic software* layers, allows ECU designers and developers to specifically focus on the realization of high-level vehicle functionalities, i.e., without the need to think about the underlying middleware and hardware. Application software components and BSW modules are often developed by different suppliers who specialize in one of these areas, as explained in more detail in the following section.

4.3 AUTOSAR Development Methodology

On the highest level of abstraction, automotive vendors developing architectural components following the AUTOSAR methodology can be classified into one of the following four major roles in the automotive development process:

- **OEM**: responsible for the logical and physical system design.
- **Tier1**: responsible for the physical ECU design and implementation of the software components allocated to this ECU.
- **Tier2**: responsible for the implementation of ECU basic software.
- **Tier3**: responsible for supplying ECU hardware, hardware drivers and corresponding compilers for building the ECU software.

In most cases, different roles represent different organizations/companies involved in the development process. For example, one car manufacturer plays the role of OEM, two software vendors play the roles of Tier1 and Tier2, respectively, and one "silicon" vendor plays the role of Tier3. However, in some cases, these roles can also be played by the same company, e.g., one car manufacturer plays the role of OEM and Tier1 by doing logical and physical system design, physical ECU design and implementation of the allocated software components (in-house development), or one software vendor plays the role of Tier1 and Tier2 by doing implementation of both the software components and the BSW modules. The development process involving all roles and their tasks is presented in Fig. 4.2.

OEMs start with *logical system design* (1) by modeling a number of composite logical software components and their port interfaces representing data exchange points. These components are usually grouped into subsystems that are in turn

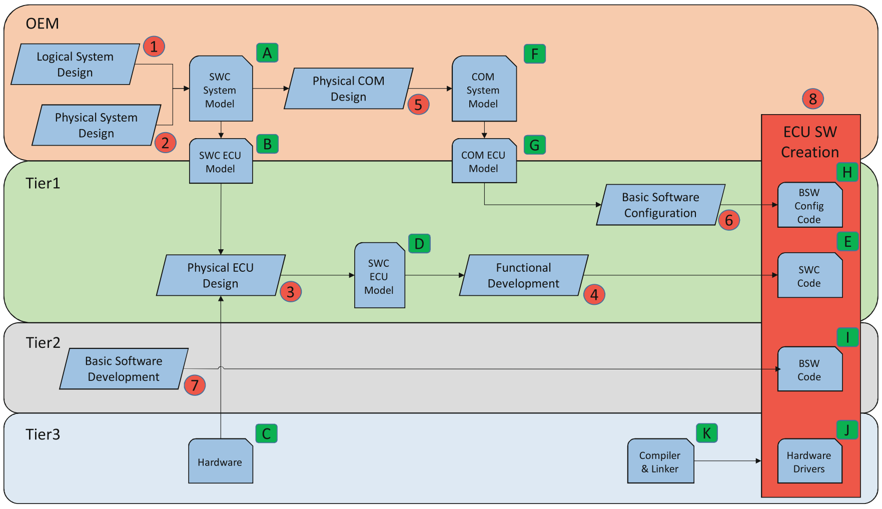

Fig. 4.2 AUTOSAR development process

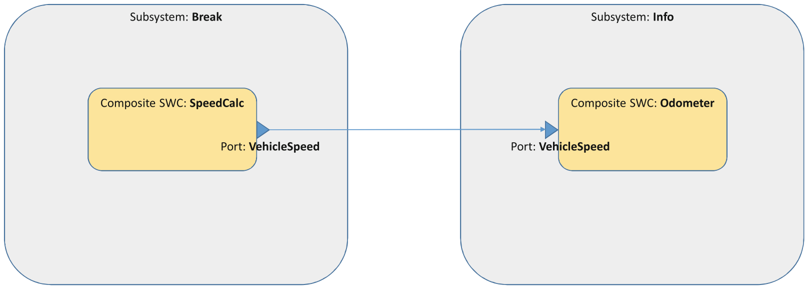

Fig. 4.3 Example of the logical system design done by OEMs (1)

grouped into logical domains. In the later stages of the development process, usually in the *physical ECU design* (3), composite software components are broken down into a number of atomic software components, but this could have been done already in the logical system design phase by OEMs. An example of the logical system design of the minimalistic system created for the purpose of this chapter that calculates vehicle speed and presents its value to the driver is presented in Fig. 4.3.

The example contains two subsystems, *Break* and *Info*, each of which consists of one composite software component, *SpeedCalc* and *Odometer*, respectively. The *SpeedCalc* component is responsible for calculating vehicle speed and it provides this information via the *VehicleSpeed* sender port. The *Odometer* component is responsible for presenting the vehicle speed information to the driver and it requires this information via the *VehicleSpeed* receiver port.

As soon as a certain number of subsystems and software components have been defined in the *logical system design* phase (1), OEMs can start with the *physical system design* (2), which involves modeling a number of ECUs connected using different electronic buses and deployment of software components to these ECUs. In case two communicating software components (with connected ports) are allocated to different ECUs, this phase also involves the creation of system signals that will be transmitted over the electronic bus connecting these two ECUs. An example of the physical system design of our minimalistic system is presented in Fig. 4.4.

The example contains two ECUs, *BreakControl* and *DriverInfo*, connected using the *Can1* bus. The *SpeedCalc* component is deployed to the *BreakControl* ECU while the *Odometer* component is deployed to the *DriverInfo* ECU. As these two components are deployed to different ECUs, information about vehicle speed is exchanged between them in a form of system signal named *VehicleSpeed*.

After the *physical system design* phase (2) is finished, detailed design of the car's functionalities allocated to composite software components deployed to different ECUs (*physical ECU design*) can be performed by the Tier1s (3). As different ECUs are usually developed by different Tier1s, OEMs are responsible for extracting the relevant information about the deployed software components from the generated *SWC system model* (A) into the *SWC ECU model* (B), known as the *ECU Extract*. The main goal of the physical ECU design phase is to break down the composite software components into a number of atomic software components that will in the

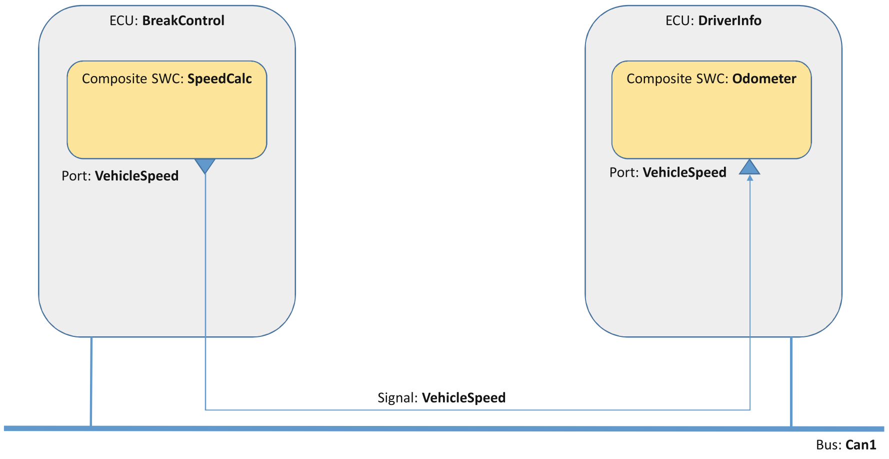

Fig. 4.4 Example of the physical system design done by OEMs (2)

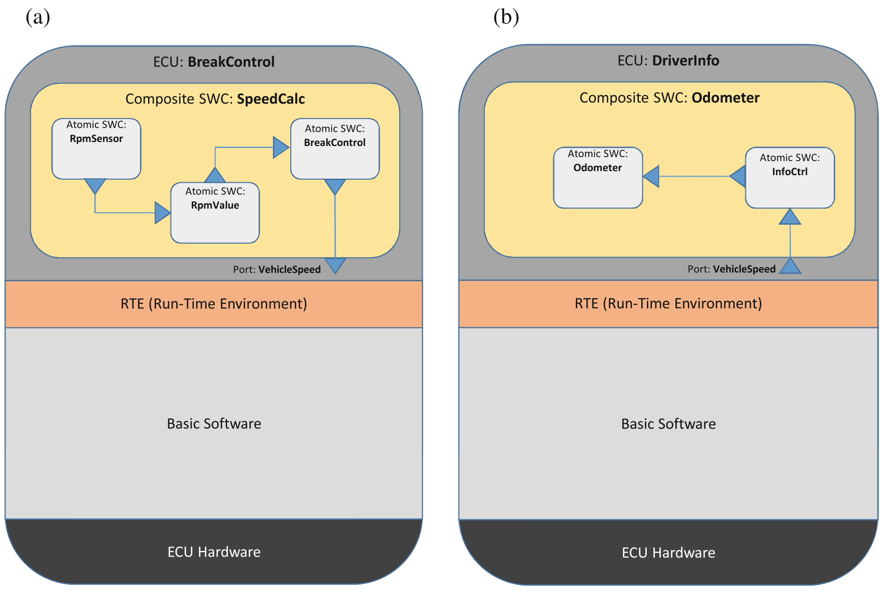

Fig. 4.5 Example of the physical ECU design done by the Tier1s (3). (**a**) BreakControl ECU. (**b**) DriverInfo ECU

end represent runnable entities at ECU run-time. An example of the physical ECU design of our minimalistic system is presented in Fig. 4.5.

The example shows detailing of the *SpeedCalc* and *Odometer* composite software components into a number of atomic software components that will represent runnables in the final ECU software. *SpeedCalc* consists of the *RpmSensor* sensor component that measures the speed of axis rotation, the *RpmValue* atomic software component that calculates the value of the rotation and the *BreakControl* atomic software component that calculates the actual vehicle speed based on the value of the axis rotation. *Odometer* consists of the *InfoControl* atomic software component that receives information about the vehicle speed and the *Odometer* atomic software component that presents the vehicle speed value to the driver.

The ECU design phase is also used to decide upon the concrete implementation data types used in the code for the data exchanged between software components based on the choice of the concrete ECU *hardware* (C) delivered by the Tier3s. For example, data can be stored as floats if the chosen CPU has support for working with the floating points.

Based on the detailed *SWC ECU model* containing the atomic software components (D), the Tier1s can continue with the *functional development* of the car's functionalities (4) allocated onto these components. This is usually done with a help of behavioral modeling with modeling tools such as Matlab Simulink, as explained in Sect. 5.2, that are able to generate source *SWC code* for the atomic software components (E) automatically from the Simulink models [LLZ13]. This part is outside of the AUTOSAR scope.

During the physical ECU design and functional development phases performed by the Tier1s, OEMs can work on the *physical COM design* (5) that aims to complete the system model with packing of signals into frames that are transmitted on the electronic buses. This phase is necessary for configuring the communication (COM) part of the AUTOSAR *basic software configuration* (6). An example of the physical COM design of our minimalistic system is presented in Fig. 4.6.

The example shows one frame of eight bytes named *CanFrm01* that is transmitted by the *BreakControl* ECU on the *Can1* bus and received by the *DriverInfo* ECU. It transports the *VehicleSpeed* signal in its first two bytes.

After the physical COM design phase has been completed for the entire system, OEMs are responsible for creating *COM ECU model* extracts (G) from the generated

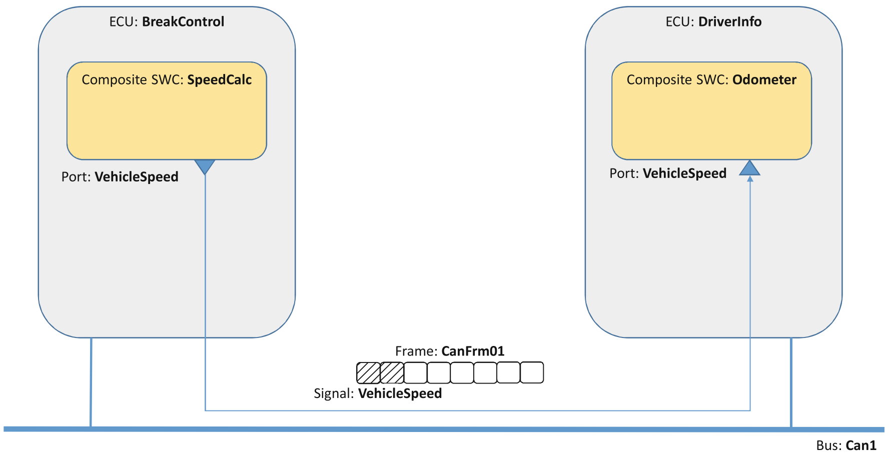

Fig. 4.6 Example of the physical COM design done by OEMs (5)

COM system model (F) for each ECU that contains only ECU-relevant information about the COM design. This step is similar to the step taken after the logical and physical system design, related to the extraction of ECU-relevant information about application software components. These ECU extracts are then sent to the Tier1s, which use them as input for configuring the COM part of the ECU *basic software configuration* (6) and, together with configuring the rest of BSW (diagnostics services, operating system, etc.), generate the complete *BSW configuration code* (H) for the developed ECU. An example of the BSW configuration design of our minimalistic system is presented in Fig. 4.7.

The example shows different groups of BSW modules, i.e., *Operating System*; *Services* including modules such as *DEM* and *DCM*; *Communication* including modules such as *COM*; and *ECU Abstraction* including modules such as *CanIf* needed for the transmission of frames on the Can bus in our example.

The actual ECU *basic software development* (7) is done by the Tier2s, based on the detailed specifications of each BSW module provided by the AUTOSAR standard, e.g., *COM*, *CanIf* or *DEM* modules. The outcome of this phase is a complete *BSW code* (I) for the entire basic software that is usually delivered by the Tier2s in the form of libraries. The *hardware drivers* for the chosen hardware (J), in our example the *CAN* driver, are delivered by the Tier3s.

The last stage in *ECU software creation* (8) is to compile and link the functional *SWC code* (E), the *BSW configuration code* (H), the functional *BSW code* (I) and the *hardware drivers* (J). This is usually done using the *compiler and linker* (K) delivered by the Tier3s.

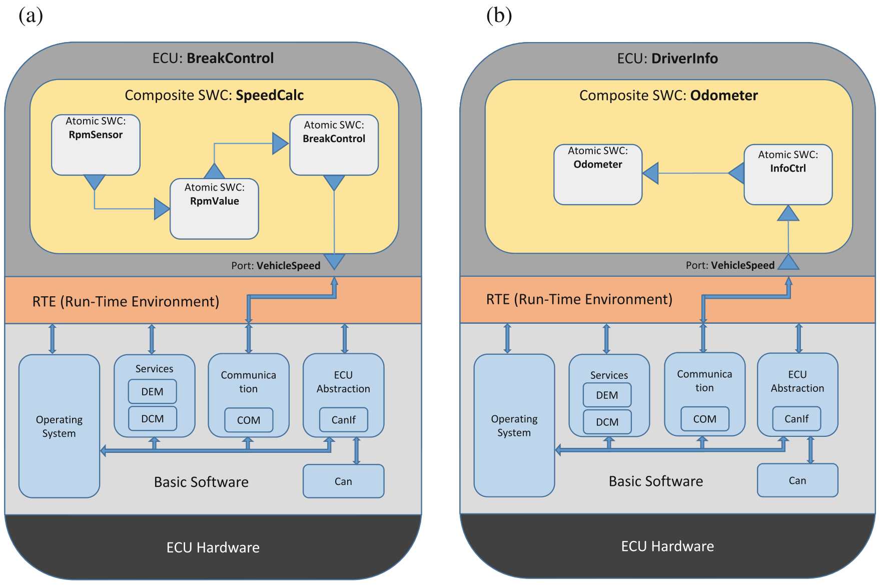

Fig. 4.7 Example of the BSW configuration design done by the Tier1s (6). (**a**) BreakControl ECU. (**b**) DriverInfo ECU

Despite the fact that the described methodology of AUTOSAR is reminiscent of the traditional waterfall development approach, except from the decoupled development of the ECU functional code and the ECU BSW code, in practice it represents just one cycle of the entire development process. In other words, steps (1)–(6) are usually repeated a number of times, adding new functionalities to the system and its ECUs. For example, new composite software components are introduced in the logical system design (1), requiring new signals in the physical system design (2); new atomic software components are introduced as part of the new composite software components in the physical ECU design (3) and implemented in the functional development (4); and new frames to transport the new signals are introduced in the physical COM design (5) and configured in the BSW configuration design (6) phase. Sometimes even the ECU hardware (C) and its compiler/linker (K) and drivers (J) can be changed between different cycles, in case it cannot withstand the additional functionality.

Examples of AUTOSAR based logical system design (1), physical system design (2), physical ECU design (3) and physical COM design (5) are presented in Sect. 4.4. Examples of AUTOSAR-based basic software development (7) and basic software configuration (6) are presented in Sect. 4.5. As already stated, functional development of software components (4) is outside of the scope of AUTOSAR and this chapter.

4.4 AUTOSAR Meta-Model

As we have seen in the previous section, a number of architectural models, as outcomes of different phases in the development methodology, are exchanged between different roles in the development process. In order to ensure that modeling tools used by OEMs in the logical (1), physical (2) and system design communication (5) phases are able to create models that could be read by the modeling tools used by the Tier1s in the physical ECU design (3) and BSW configuration phases (6), AUTOSAR defines a meta-model that specifies the language for these exchanged models [NDWK99]. Therefore, models (A), (B), (D), (F) and (G) represent instances of the AUTOSAR meta-model that specifies their abstract syntax in the UML language. The models themselves are serialized into XML (referred to as ARXML, the AUTOSAR XML), which represents their concrete syntax, and are validated by the AUTOSAR XML schema that is generated from the AUTOSAR meta-model [PB06].

In this section, we first describe the AUTOSAR meta-modeling environment in Sect. 4.4.1. We then show an example use of the AUTOSAR meta-model in the logical system design (1), physical system design (2), physical ECU design (3) and physical COM design (5) phases in Sect. 4.4.2 using our minimalistic system presented in the previous section and present examples of these models in the ARXML syntax. Finally, we discuss the semantics of the AUTOSAR models described in the AUTOSAR template specifications in Sect. 4.4.3.

4.4.1 AUTOSAR Meta-Modeling Environment

As opposed to the commonly accepted meta-modeling hierarchy of MOF [Obj04] that defines four layers [BG01], the AUTOSAR modeling environment has a five-layer hierarchy, as presented below (the names of the layers are taken from the AUTOSAR *Generic Structure* specification [AUT16f]):

1. The **ARM4**: MOF 2.0, e.g., the MOF Class
2. The **ARM3**: UML and AUTOSAR UML profile, e.g., the UML Class
3. The **ARM2**: Meta-model, e.g., the SoftwareComponent
4. The **ARM1**: Models, e.g., the WindShieldWiper
5. The **ARM0**: Objects, e.g., the WindShieldWiper in the ECU memory

The mismatch between the number of layers defined by MOF and AUTOSAR lies in the fact that MOF considers only layers connected by the linguistic instantiation (e.g., *SystemSignal* is an instance of UML *Class*), while AUTOSAR considers both linguistic and ontological layers (e.g., *VehicleSpeed* is an instance of *System-Signal*) [Küh06]. To link these two interpretations of the meta-modeling hierarchy, we can visualize the AUTOSAR meta-modeling hierarchy using a two-dimensional representation (known as OCA—Orthogonal Classification Architecture [AK03]), as shown in Fig. 4.8. Linguistic instantiation ("L" layers corresponding to MOF layers) are represented vertically and ontological layers ("O" layers) horizontally.

The *ARM2* layer is commonly referred to as the "AUTOSAR meta-model" and it ontologically defines, using UML syntax (i.e., AUTOSAR meta-model is defined as an instance of UML), AUTOSAR models residing on the *M1* layer (both the

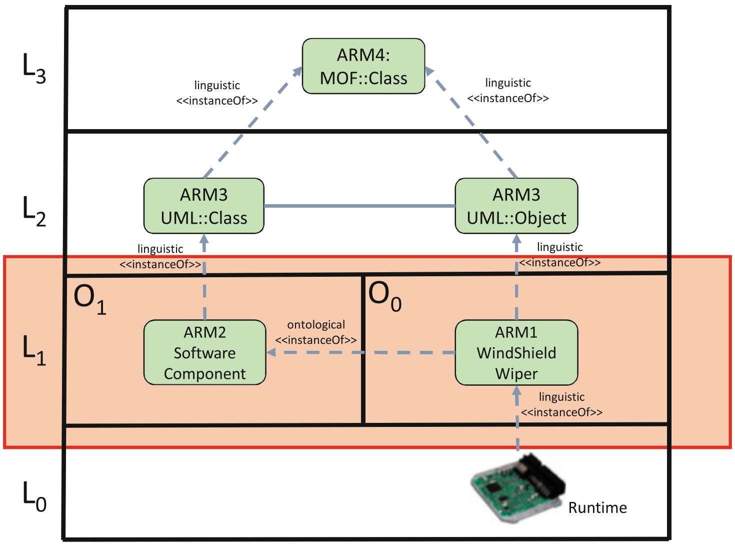
L_3
L_2
L_1
L_0
ARM4: MOF::Class
linguistic <<instanceOf>>
linguistic <<instanceOf>>
ARM3 UML::Class
ARM3 UML::Object
linguistic <<instanceOf>>
linguistic <<instanceOf>>
O_1
O_0
ARM2 Software Component
ontological <<instanceOf>>
ARM1 WindShield Wiper
linguistic <<instanceOf>>
Runtime

Fig. 4.8 AUTOSAR meta-model layers [DSTH16]

AUTOSAR meta-model and AUTOSAR models are located on the *L1* layer). The AUTOSAR meta-model also uses a UML profile that extends the UML meta-model on the *ARM3* layer, which specifies the used stereotypes and tagged values.

Structurally, the AUTOSAR meta-model is divided into a number of top-level packages referred to as "templates", where each template defines how to model one part of the automotive system. The modeling semantics, referred to as design requirements and constraints, are described in the AUTOSAR template specifications [Gou10]. The AUTOSAR templates and their relations are presented in Fig. 4.9.

Probably the most important templates for the design of automotive software systems are the *SWComponentTemplate*, which defines how to model software components and their interaction; *SystemTemplate*, which defines how to model

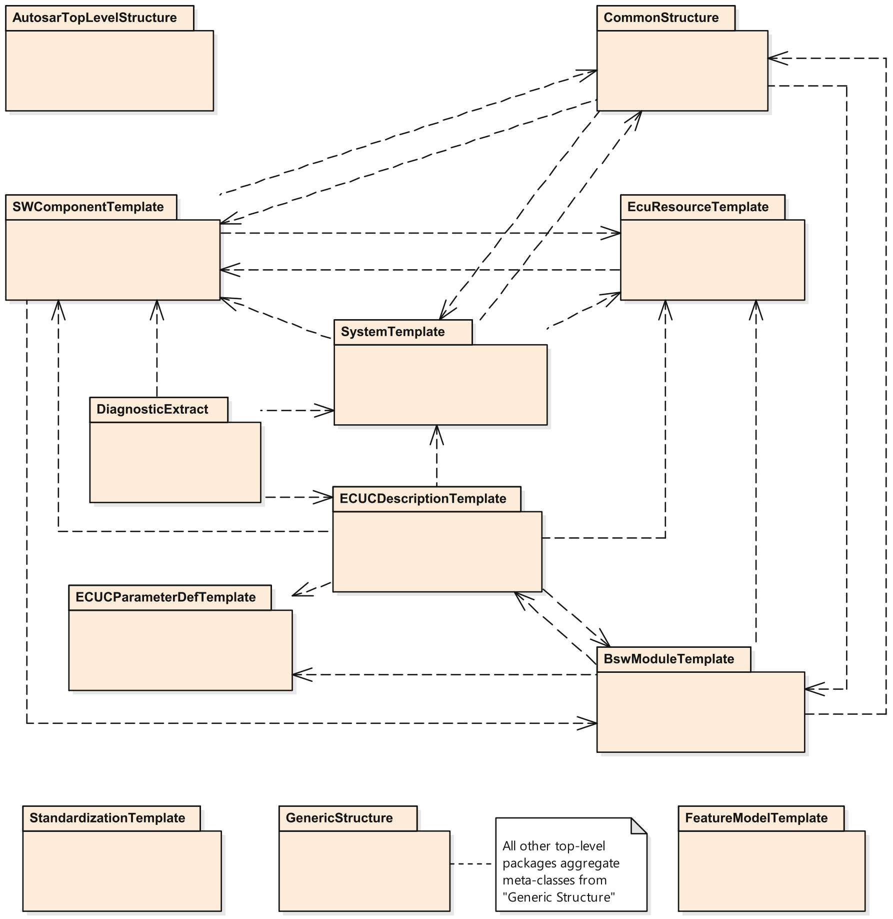

Fig. 4.9 AUTOSAR templates [AUT16f]

ECUs and their communication; and *ECUCParameterDefTemplate* and *ECUCDescriptionTemplate*, which define how to configure ECU basic software. In addition to these templates, AUTOSAR *GenericStructure* template is used to define general concepts (meta-classes) used by all other templates, e.g., handling different variations in architectural models related to different vehicles. In the next subsection, we provide examples of these templates and AUTOSAR models that instantiate them.

4.4.2 Architectural Design Based on the AUTOSAR Meta-Model

A simplified excerpt from the *SWComponentTemplate* that is needed for the logical system and physical ECU design of our minimalistic example that calculates vehicle speed and presents its value to the driver is presented in Fig. 4.10.

The excerpt shows the abstract meta-class *SwComponent* that can be either *AtomicSwComponent* or *CompositeSwComponent*, which may refer to multiple *AtomicSwComponent*s. Both types of *SwComponent*s may contain a number of *Port*s that can either be *ProvidedPort*s providing data to the other components in the system, or *RequiredPort*s requiring data from the other components in

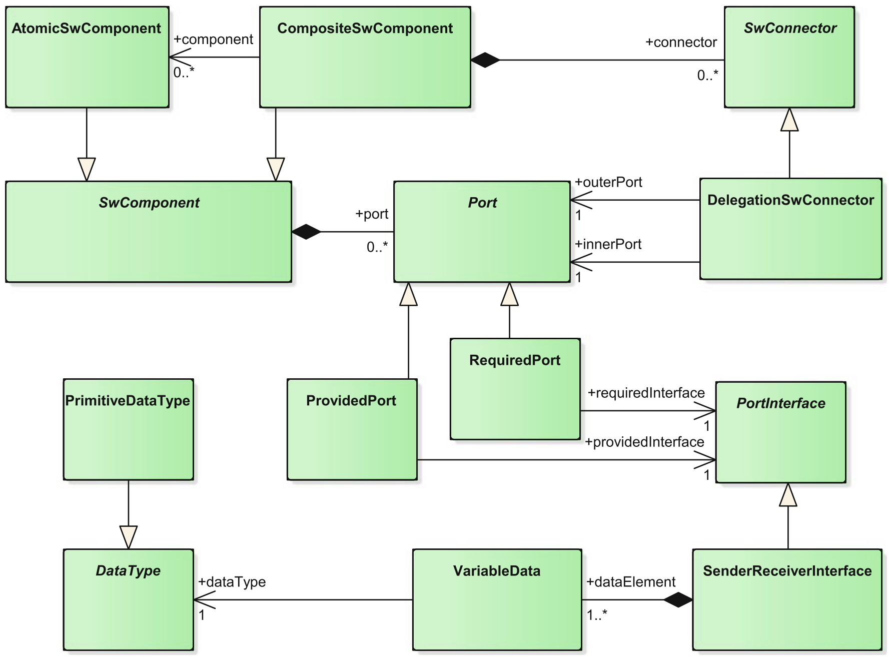

Fig. 4.10 Logical and ECU design example (*SwComponentTemplate*)

the system. Ports on the *CompositeSwComponent*s are connected to the ports of the *AtomicSwComponent*s using *DelegationSwConnector*s that belong to the *CompositeSwComponent*s, i.e., *DelegationSwConnector* points to an *outerPort* of the *CompositeSwComponent* and an *innerPort* of the *AtomicSwComponent*. Finally, *Port*s refer to a corresponding *PortInterface*, e.g., *SenderReceiverInterface* or *ClientServerInterface*, that contains the actual definition of the *DataType* that is provided or required by this port (e.g., unsigned integer of 32 bits or a struct that consists of an integer and a float).

The model of our example of the logical system design presented in Fig. 4.3 that instantiates the *SWComponentTemplate* part of the meta-model is shown in Fig. 4.11 in ARXML syntax. We chose ARXML as it is used as a model exchange format between OEMs and Tier1s, but UML could be used as well.

The example shows the definition of the *SpeedCalc* composite software component (lines 1–11) with the *VehicleSpeed* provided port (lines 4–9), and the *Odometer* composite software component (lines 12–22) with the *VehicleSpeed*

```
1    <COMPOSITE-SW-COMPONENT UUID="...">
2       <SHORT-NAME>SpeedCalc</SHORT-NAME>
3       <PORTS>
4           <PROVIDED-PORT UUID="...">
5               <SHORT-NAME>VehicleSpeed</SHORT-NAME>
6               <PROVIDED-INTERFACE-REF DEST="SENDER-RECEIVER-INTERFACE">
7                   /.../VehicleSpeedInterface
8               </PROVIDED-INTERFACE-REF>
9           </PROVIDED-PORT>
10      </PORTS>
11  </COMPOSITE-SW-COMPONENT>
12   <COMPOSITE-SW-COMPONENT UUID="...">
13      <SHORT-NAME>Odometer</SHORT-NAME>
14      <PORTS>
15          <REQUIRED-PORT UUID="...">
16              <SHORT-NAME>VehicleSpeed</SHORT-NAME>
17              <REQUIRED-INTERFACE-REF DEST="SENDER-RECEIVER-INTERFACE">
18                  /.../VehicleSpeedInterface
19              </REQUIRED-INTERFACE-REF>
20          </REQUIRED-PORT>
21      </PORTS>
22  </COMPOSITE-SW-COMPONENT>
23  <SENDER-RECEIVER-INTERFACE UUID="...">
24      <SHORT-NAME>VehicleSpeedInterface</SHORT-NAME>
25      <DATA-ELEMENTS>
26          <VARIABLE-DATA UUID="...">
27              <SHORT-NAME>VehicleSpeed</SHORT-NAME>
28              <DATA-TYPEREF DEST="PRIMITIVE-DATA-TYPE">
29                  /.../UInt16
30              </DATA-TYPE-REF>
31          </VARIABLE-DATA>
32      </DATA-ELEMENTS>
33  </SENDER-RECEIVER-INTERFACE>
34  <PRIMITIVE-DATA-TYPE UUID="...">
35      <SHORT-NAME>UInt16</SHORT-NAME>
36  </PRIMITIVE-DATA-TYPE>
```

Fig. 4.11 AUTOSAR model example: logical design

required port (15–20). Both ports refer to the same sender-receiver interface (lines 23–33) that in turn refers to the unsigned integer type of 16 bits (lines 34–36) for the provided/required data.

According to the AUTOSAR methodology, these composite software components are, after their allocation to the chosen ECUs, broken down into a number of atomic software components during the physical ECU design phase. The partial model of our minimalistic example of the physical ECU design presented in Fig. 4.5 that instantiates the *SWComponentTemplate* part of the meta-model is shown in Fig. 4.12 in ARXML syntax.

The example shows the definition of the *BreakControl* atomic software component (lines 31–41) with the *VehicleSpeed* provided port (lines 34–39) that is referenced (lines 12–17) from the *SpeedCalc* composite software component (lines 1–30). We can also see the delegation connector *Delegation1* inside the *SpeedCalc*

```
1   <COMPOSITE-SW-COMPONENT UUID="...">
2       <SHORT-NAME>SpeedCalc</SHORT-NAME>
3       <PORTS>
4           <PROVIDED-PORT UUID="...">
5               <SHORT-NAME>VehicleSpeed</SHORT-NAME>
6               <PROVIDED-INTERFACE-REF DEST="SENDER-RECEIVER-INTERFACE">
7                   /.../VehicleSpeedInterface
8               </PROVIDED-INTERFACE-REF>
9           </PROVIDED-PORT>
10      </PORTS>
11      <COMPONENTS>
12          <COMPONENT>
13              <SHORT-NAME>BreakControl</SHORT-NAME>
14              <COMPONENT-REF DEST="ATOMIC-SW-COMPONENT">
15                  /.../BreakControl
16              </COMPONENT-REF>
17          </COMPONENT>
18      </COMPONENTS>
19      <CONNECTORS>
20          <DELEGATION-SW-CONNECTOR UUID="...">
21              <SHORT-NAME>Delegation1</SHORT-NAME>
22              <INNER-PORT-REF DEST="P-PORT-PROTOTYPE">
23                  /.../BreakControl/VehicleSpeed
24              </INNER-PORT-REF>
25              <OUTER-PORT-REF DEST="P-PORT-PROTOTYPE">
26                  /.../SpeedCalc/VehicleSpeed
27              </OUTER-PORT-REF>
28          </DELEGATION-SW-CONNECTOR>
29      </CONNECTORS>
30  </COMPOSITE-SW-COMPONENT>
31  <ATOMIC-SW-COMPONENT UUID="...">
32      <SHORT-NAME>BreakControl</SHORT-NAME>
33      <PORTS>
34          <PROVIDED-PORT UUID="...">
35              <SHORT-NAME>VehicleSpeed</SHORT-NAME>
36              <PROVIDED-INTERFACE-REF DEST="SENDER-RECEIVER-INTERFACE">
37                  /.../VehicleSpeedInterface
38              </PROVIDED-INTERFACE-REF>
39          </PROVIDED-PORT>
40      </PORTS>
41  </ATOMIC-SW-COMPONENT>
```

Fig. 4.12 AUTOSAR model example: ECU design

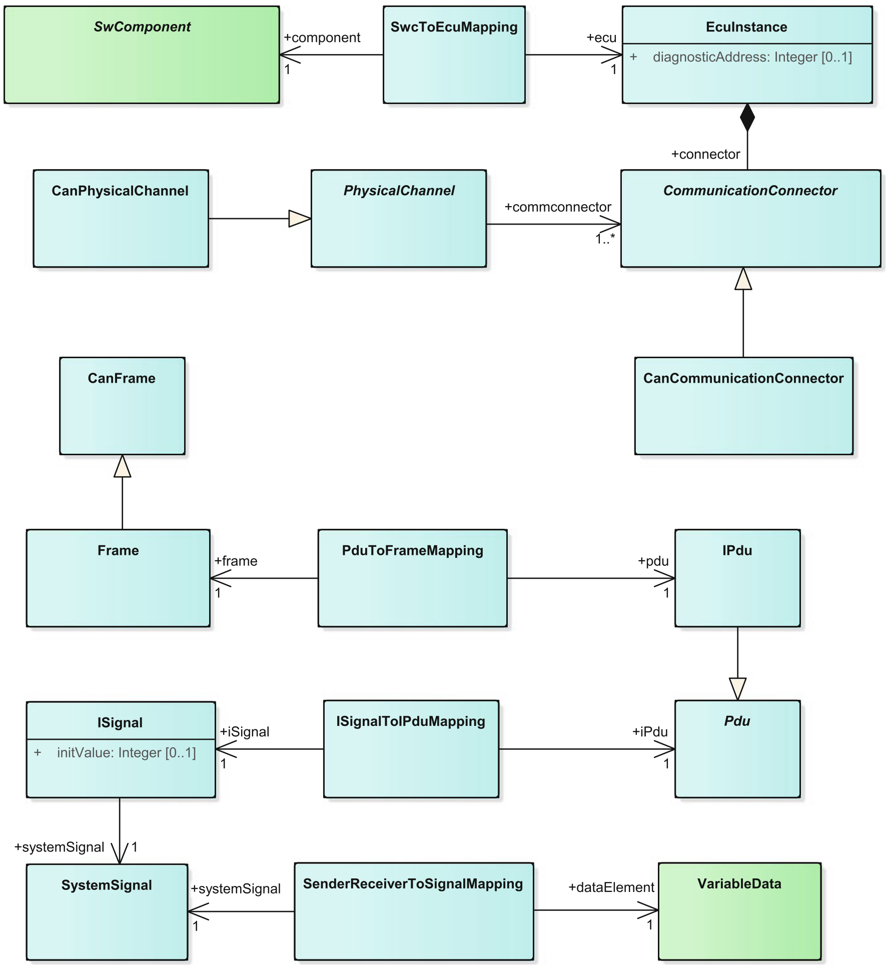

Fig. 4.13 Physical and COM design (*SystemTemplate*)

composite software component (lines 20–28) that connects the provided ports in the *SpeedCalc* and *BreakControl* software components.

A simplified excerpt from the *SystemTemplate* that is needed for the physical and COM system designs of our minimalistic example is presented in Fig. 4.13.

Related to the physical system design, the excerpt shows the *EcuInstance* meta-class with the *diagnosticAddress* attribute which may contain a number of *CommunicationConnector*s that represent connections of *EcuIstance* to a *PhysicalChannel* (e.g., *CanCommunicationConnector* connects one *EcuInstance* to a *CanPhysicalChannel*). A number of *SwComponent*s (*CompositeSwComponent*s or *AtomicSwComponents*) created in the logical designs can be allocated to one *EcuInstace* by means of *SwcToEcuMapping*s.

Related to the physical COM design, the excerpt shows the *SenderReceiver-ToSignalMapping* of the *VariableData* created in the logical design of a *SystemSignal*. It also shows that one *SystemSignal* can be sent to multiple buses by creating different *ISignal*s and mapping them to *IPdu*s, which are in turn mapped to *Frame*s. *IPdu* is one type of *Pdu* (Protocol Data Unit) that is used for transporting signals, and there may be other types of *Pdu*s, e.g., *DcmPdu* for transporting diagnostic messages.

The model of our example of the physical system design presented in Fig. 4.4 that instantiates the *SystemTemplate* part of the meta-model is shown in Fig. 4.14.

The example shows the definition of the *BreakControl* ECU with diagnostic address 10 (lines 1–9) that owns a CAN communication connector (lines 5–7). It also shows the mapping of the *SpeedCalc* composite software component onto the *BreakControl* ECU (lines 10–14). Finally, it shows the definition of the *Can1* physical channel (lines 15–24) that points to the CAN communication connector of the *BreakControl* ECU (lines 19–21), thereby indicating that this ECU is connected to *Can1*.

The model of our example of the COM system design presented in Fig. 4.6 that instantiates the *SystemTemplate* part of the meta-model is shown in Fig. 4.15.

The example shows the definition of the *VehicleSpeed* system signal (lines 1–3) that is mapped to the *SpeedCalc* variable data element defined in the logical design phase (lines 4–12). The example also shows the creation of the *ISignal VehicleSpeedCan1* (lines) with initial value of 0 that is meant to transmit the vehicle speed on the *Can1* bus defined in the physical design phase. This *ISignal* is mapped to *Pdu1* (lines 20–22) using *ISignalToIPduMapping* (23–27) that in turn is mapped to *CanFrame1* (lines 28–30) using *IPduToFrameMapping* (lines 31–35).

```
1   <ECU-INSTANCE UUID="...">
2       <SHORT-NAME>BreakControl</SHORT-NAME>
3       <ECU-ADDRESS>10</ECU-ADDRESS>
4       <CONNECTORS>
5           <CAN-COMMUNICATION-CONNECTOR UUID="...">
6               <SHORT-NAME>Can1Connector</SHORT-NAME>
7           </CAN-COMMUNICATION-CONNECTOR>
8       </CONNECTORS>
9   </ECU-INSTANCE>
10  <SWC-TO-ECU-MAPPING UUID="...">
11      <SHORT-NAME>Mapping1</SHORT-NAME>
12      <COMPONENT-REF DEST="SW-COMPONENT">/.../SpeedCalc</SW-REF>
13      <ECU-REF DEST="ECU-INSTANCE">/.../BreakControl</ECU-REF>
14  </SWC-TO-ECU-MAPPING>
15  <CAN-PHYSICAL-CHANNEL UUID="...">
16      <SHORT-NAME>Can1</SHORT-NAME>
17      <COMM-CONNECTORS>
18          <COMMUNICATION-CONNECTOR-REF-CONDITIONAL>
19              <COMMUNICATION-CONNECTOR-REF DEST="CAN-COMMUNICATION-CONNECTOR">
20                  /.../BreakControl/Can1Connector
21              </COMMUNICATION-CONNECTOR-REF>
22          </COMMUNICATION-CONNECTOR-REF-CONDITIONAL>
23      </COMM-CONNECTORS>
24  </CAN-PHYSICAL-CHANNEL>
```

Fig. 4.14 AUTOSAR model example: physical design

```
<SYSTEM-SYGNAL UUID="...">
    <SHORT-NAME>VehicleSpeed</SHORT-NAME>
</SYSTEM-SYGNAL>
<SENDER-RECEIVER-TO-SIGNAL-MAPPING UUID="...">
    <SHORT-NAME>Mapping2</SHORT-NAME>
    <DATA-ELEMENT-REF DEST="VARIABLE-DATA">
        /.../VehicleSpeedInterface/SpeedCalc
    </DATA-ELEMENT-REF>
    <SYSTEM-SIGNAL-REF DEST="SYSTEM-SIGNAL">
        /.../VehicleSpeed
    </SYSTEM-SIGNAL-REF>
</SENDER-RECEIVER-TO-SIGNAL-MAPPING>
<I-SYGNAL UUID="...">
    <SHORT-NAME>VehicleSpeedCan1</SHORT-NAME>
    <INIT-VALUE>0</INIT-VALUE>
    <SYSTEM-SIGNAL-REF DEST="SYSTEM-SIGNAL">
        /.../VehicleSpeed
    </SYSTEM-SIGNAL-REF>
</I-SYGNAL>
<I-PDU UUID="...">
    <SHORT-NAME>IPdu1</SHORT-NAME>
</I-PDU>
<I-SIGNAL-TO-I-PDU-MAPPING UUID="...">
    <SHORT-NAME>Mapping3</SHORT-NAME>
    <I-PDU-REF DEST="I-PDU">/.../IPdu1</I-PDU-REF>
    <I-SIGNAL-REF DEST="I-SIGNAL">/.../VehicleSpeedCan1</I-SIGNAL-REF>
</I-SIGNAL-TO-I-PDU-MAPPING>
<CAN-FRAME UUID="...">
    <SHORT-NAME>CanFrame1</SHORT-NAME>
</CAN-FRAME>
<I-PDU-TO-FRAME-MAPPING UUID="...">
    <SHORT-NAME>Mapping4</SHORT-NAME>
    <PDU-REF DEST="I-PDU">/.../IPdu1</PDU-REF>
    <FRAME-REF DEST="CAN-FRAME">/.../CanFrame1</FRAME-REF>
</I-PDU-TO-FRAME-MAPPING>
```

Fig. 4.15 AUTOSAR model example: COM design

4.4.3 AUTOSAR Template Specifications

Like other language definitions, the AUTOSAR meta-model defines only syntax for different types of architectural models, without explaining how its meta-classes shall be used to achieve certain semantics. This is done in natural language specifications called templates [Gou10], e.g., *SwComponentTemplate* and *SystemTemplate*, that explain different parts of the AUTOSAR meta-model. These templates consist of the following main items:

- Design requirements that should be fulfilled by the models (specification items).
- Constraints that should be fulfilled by the models and checked by modeling tools.
- Figures explaining the use of a group of meta-classes.
- Class tables explaining meta-classes and their attributes/connectors.

As an example of a specification item related to our minimalistic example that calculates vehicle speed and presents its value to the driver, we present specification item no. 01009 from the *SystemTemplate* that describes the use of *CommunicationConnector*s:

[TPS_SYST_01009] Definition of CommunicationConnector ⌈An EcuInstance uses CommunicationConnector elements in order to describe its bus interfaces and to specify the sending/receiving behavior.⌋

As an example of a constraint, we present constraint no. 1032 from the *SwComponentTemplate* that describes the limitation in the use of *DelegationSwConnector*s.

[constr_1032] DelegationSwConnector can only connect ports of the same kind ⌈A DelegationSwConnector can only connect ports of the same kind, i.e. ProvidedPort to ProvidedPort and RequiredPort to RequiredPort.⌋

The majority of constraints including *constr_1032* could be specified directly in the AUTOSAR meta-model using OCL (Object Constraint Language). However, due to the complexity of OCL and thousands of automotive engineers in more than hundred OEM and supplier companies that develop automotive software components based on AUTOSAR, natural language specifications are considered a better approach for such a wide audience [NDWK99].

Meta-model figures show relationships between a number of meta-classes using UML notation and they are similar to Figs. 4.10 and 4.13 presented in the previous section. These figures are usually followed by class tables that describe the meta-classes in the figures in more detail, e.g. description of the meta-classes, their parent classes and attributes/connectors, so that the readers of the AUTOSAR specification do not need to look directly into the AUTOSAR meta-model which is maintained by the *Enterprise Architect* tool.

In addition to specification items, constraints, figures and class tables, AUTOSAR template specifications also contain a substantial amount of plain text that provides additional explanations, e.g., introductions to the topic and notes about specification items and constraints.

4.5 AUTOSAR ECU Middleware

AUTOSAR provides detailed functional specifications for the modules of its middleware layer (basic software modules). For example, the *COM* specification describes the functionality of the Communication Manager module that is mostly responsible for handling the communication between ECUs, i.e., transmitting signals received from the RTE onto electronic buses and vice versa. These specifications consist of the following main items:

- Functional requirements that should be fulfilled by the implementation of the BSW modules.
- Description of APIs of the BSW modules.

- Sequence diagrams explaining the interaction between BSW modules.
- Configuration parameters that are used for configuring the BSW modules.

The functional side of the AUTOSAR BSW module specifications (functional requirements, APIs and sequence diagrams) is outside of the scope of this chapter. However, we do describe here the general approach to configuration of the BSW modules as it is done based on the AUTOSAR meta-model and its templates.

Two of the AUTOSAR templates are responsible for specifying configuration of the AUTOSAR basic software—*ECUCParameterDefTemplate* and *ECUCDescriptionTemplate* on the *ARM2* layer. *ECUCParameterDefTemplate* specifies the general definition of configuration parameters, e.g., that parameters can be grouped into containers of parameters and that they can be configured at different configuration times (e.g., before or after building the complete ECU software). *ECUCDescriptionTemplate* specifies modeling of concrete parameter and container values that reference their corresponding definitions from the *ECUCParameterDefTemplate*.

The values of configuration parameters from the *ECUCDescriptionTemplate* models can be automatically derived from the models of other templates, e.g., *SoftwareComponentTemplate* and *SystemTemplate*. This process is called "upstream mapping" and it can be done automatically with support from the ECU configuration tools [LH09]. A simplified example of the *ECUCParameterDefTemplate* and *ECUCParameterDefTemplate* and their models, including the upstream mapping process, is shown in Fig. 4.16 in UML syntax.

The *ECUCParameterDefTemplate* on the *ARM2* layer (left blue box) specifies modeling of the definition of configuration parameters (*ECUCParameterDef*s) and containers (*ECUCContainerDef*s), with an example of the integer parameter definition (*ECUCIntegerParameterDef*). The *ECUCDescriptionTemplate* (left yellow box) specifies modeling of the values of containers (*ECUCContainerValue*s) and

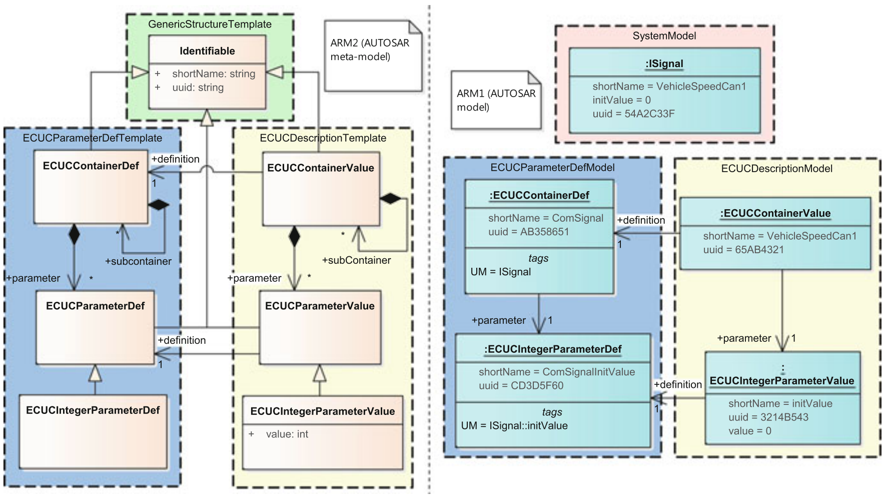

Fig. 4.16 Example of the AUTOSAR templates and their models

parameters (*ECUCParameterValue*s), with an example of the integer parameter value (*ECUCIntegerParameterValue*). As with the elements from the *SwComponentTemplate* and the *SystemTemplate*, the elements from these two templates are also inherited from the common element in the *GenericStructureTemplate* (green box) named *Identifiable*, which provides them with a short name and unique identifier (UUID).

The standardized model (i.e., provided by AUTOSAR) of the *ECUCParameterDefTemplate* can be seen on the *ARM1* layer (right blue box). It shows the *ECUCContainerDef* instance with *shortName* "ComSignal" that refers to the *ECUCParameterDef* instance with *shortName* "ComSignalInitValue". These two elements both have the tagged value named *UM*, denoting Upstream Mapping. The *UM* tagged value for the "ComSignal" container instance refers to the *ISignal* meta-class from the *SystemTemplate*. The *UM* tagged value for the "ComSignalInitValue" parameter instance refers to the *initValue* attribute of the *ISignal*. This implies that for every *ISignal* instance in the *SystemModel*, one *ECUCContainerValue* instance in the *ECUCDescriptionModel* shall be created with an *ECUCParameterValue* instance. The value of this parameter instance shall be equal to the *initValue* attribute of that *SystemSignal* instance.

Considering the "VehicleSpeedCan1" *ISignal* with "initValue" 0 (orange box) that we defined in our *SystemModel* shown in Fig. 4.15 (COM design phase), the *ECUCDescriptionModel* (right yellow box) can be generated. This model contains one instance of the *ECUCContainerValue* with *shortName* "VehicleSpeedCan1" that is defined by the "ComSignal" container definition and refers to one instance of the *ECUCParameterValue* with *shortName* "initValue" of value 0 that is defined by the "ComSignalInitValue" parameter definition.

AUTOSAR provides the standardized *ARM1* models of the *ECUCParameterDefTemplate* for all configuration parameters and containers of the ECU basic software. For example, the *ComSignal* container with *ComSignalInitValue* are standardized for the *COM* BSW module. On the smallest granularity, standardized models of the *ECUCParameterDefTemplate* are divided into a number of packages, where each package contains configuration parameters of one *Basic software* module. At the highest level of granularity, these models are divided into different logical packages, including ECU communication, diagnostics, memory access and IO access.

4.6 AUTOSAR Evolution

The development of the AUTOSAR standard started in 2003 and its first release in mass vehicle production was R3.0.1 from 2007. In this section, we show trends in the evolution of the AUTOSAR meta-model and its requirements (both template and basic software) from its first release until release 4.2.2 from 2016, with a focus on the newer releases (R4.0.1–R4.2.2).

4.6.1 AUTOSAR Meta-Model Evolution

From the architectural modeling point of view, the most important artifact to be analyzed through different releases of AUTOSAR is the AUTOSAR meta-model, as it defines several different types of AUTOSAR models such as SWC, COM and BSW configuration models. We start the analysis of the AUTOSAR meta-model evolution by showing its increase in size from the initial release until the latest one. Figure 4.17 shows the number of AUTOSAR meta-classes in all of its templates (left) and the number of standardized BSW configuration parameters, as instances of the *ECUCParameterDefTemplate* (right) [DSTH14].

The figure indicates relatively even evolution of the AUTOSAR application software and AUTOSAR basic software, except for the R1.0, where no BSW configuration parameters have been standardized. We can also ascertain that there is a significant increase in the number of meta-classes and configuration parameters starting from R4.0.1, and relatively small change between R3.0.1 and R3.1.5. This is because the development of AUTOSAR R4.0.1 started in parallel to that of R3.1.1 and continued to develop as independent branches, namely the 3.x branch and the 4.x branch, for a couple of years until all AUTOSAR OEMs switched to 4.x. The main development focus, such as the introduction of new AUTOSAR features, was on the 4.x branch, and the 3.x branch was considered to be in the maintenance mode, focusing mostly on fixing errors in the meta-model and specifications and implementing the most important features from the 4.x branch.

AUTOSAR R4.0.1 was made public in 2009 and brought significant changes to almost all specifications, including the AUTOSAR meta-model. These changes included a number of new features, referred to as concepts by AUTOSAR, such as for supporting the LIN 2.1 electronic bus and concept enabling the existence of different variants in AUTOASAR models related to different vehicle lines. However, it also included clean-up activities of the meta-classes and configuration parameters of unused or broken concepts. Because AUTOSAR 4.x releases are used today by the majority of AUTOSAR OEMs, we provide in the rest of this section a more detailed analysis of the evolution trends of the 4.x branch. Figure 4.18 shows the number of added, modified and removed meta-classes between different releases of 4.x.

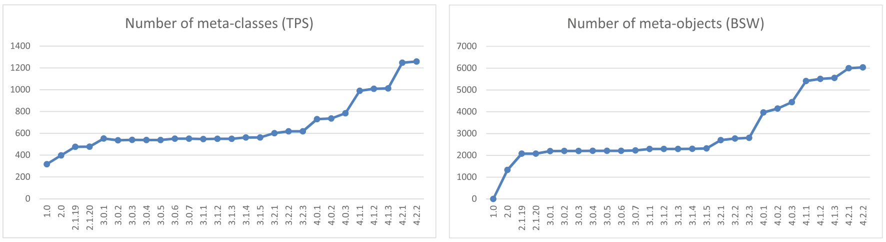

Fig. 4.17 Number of classes (TPS) and objects (BSW)

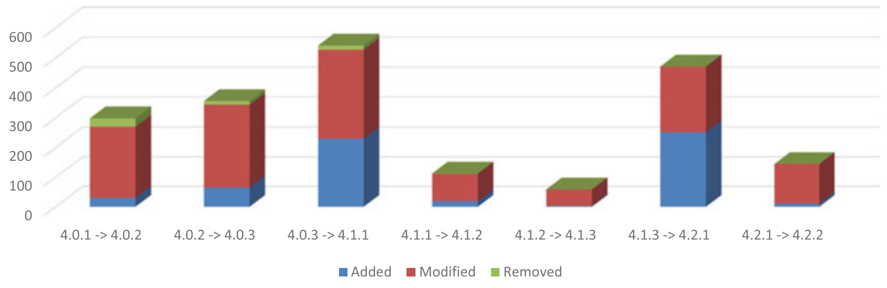

Fig. 4.18 Number of added, modified and removed classes (TPS)

At least three important conclusions about the evolution of the AUTOSAR meta-model can be derived from this figure. First, we can see that the evolution is mostly driven by modifications and additions of meta-classes, while removals are very seldom. The main reason behind this is the strong requirement for backwards compatibility of the AUTOSAR schema that is generated from the AUTOSAR meta-model, e.g., R4.0.2 models should be validated by the R4.0.3 schema or later. Second, we can see that the initial three releases, R4.0.1 to R4.1.1, all have increased the number of added, modified and removed meta-classes, indicating that it took a couple of releases to stabilize the 4.x branch. Finally, we can see that generally only minor AUTOSR releases (second digit changed, e.g., R4.1.1 vs. R4.2.1) bring a lot of new meta-classes. This is related to the AUTOSAR's policy that only major (first digit changed) and minor (second digit changed) releases may introduce new features while revisions (third digit changed) are mostly responsible for fixing errors in the meta-model related to the existing features.

In order for the automotive engineers to be able to make a preliminary assessment of the impact of adopting a new AUTOSAR release, or a subset of its new features, on the used modeling tools, a measure of meta-model change (*NoC*—Number of Changes) has been developed [DSTH14]. *NoC* considers all possible changes to the meta-classes, meta-attributes, and meta-connectors that need to be implemented by the vendors of the AUTOSAR tools used by OEMs and Tier1s. We use this measure to present the estimated effort needed to update the AUTOSAR modeling tool-chain in order to switch from one AUTOSAR release to another (e.g., 3.x to 4.x). The measurement results are presented in Fig. 4.19.

As expected, the highest effort is needed when switching from the older AUTOSAR releases (e.g., R1.0 and R2.0) to the newer ones (e.g., R4.2.1 and R4.2.2). However, the figure also shows that there is a significantly higher effort caused by the AUTOSAR releases in branch 4.x than in releases from the previous branches. This indicates that the functional "big bang" of AUTOSAR started with branch 4.x and it still continues to expand.

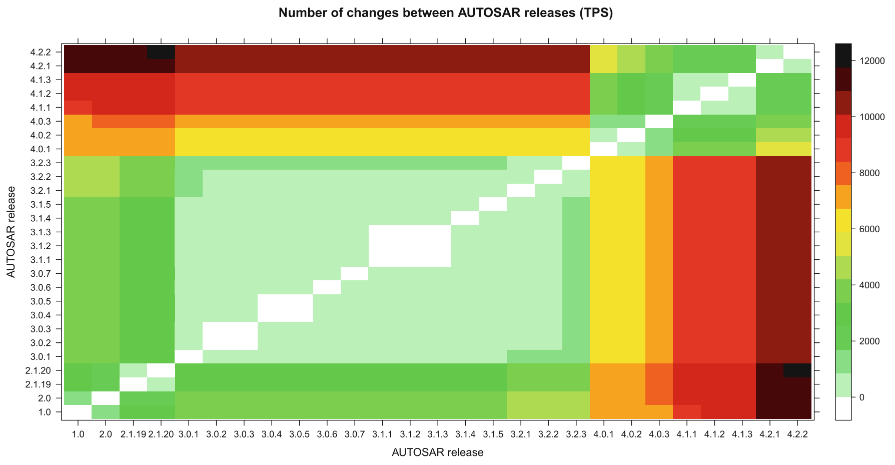

Fig. 4.19 Number of changes between releases (TPS)

Due to the fact that this expansion is mostly driven by new features incorporated into the minor releases of the AUTOSAR branch 4.x, we continue our analysis of the AUTOSAR evolution by presenting the impact of 14 new concepts of AUTOSAR R4.2.1. A brief description of each concept is presented below (more details can be found in [AUT16i]).

1. **SwitchConfiguration**: Enables full utilization of Ethernet and Ethernet switches as a communication medium between ECUs.
2. **SenderReceiverSerialization**: Enables mapping of complex data to single signal entities by means of byte array serialization. The goal is to reduce the number of signals and minimize the signal processing time.
3. **CANFD**: Introduces a new communication protocol for the CAN bus with higher bandwidth and payload for large signals.
4. **EfficientCOMforLargeData**: Enables faster transmission of large signals through the ECU by avoiding overhead of the *COM* module.
5. **E2E Extension**: Reworks the modeling of safety communication (e.g., indicator about modified or missing data during transport) between ECUs so that it does not require additional non-standardized code.
6. **GlobalTimeSynchronization**: Provides a common time base that is distributed across various buses for accurate ECU data correlation.
7. **SupportForPBLAndPBSECUConfiguration**: Enables simultaneous configuration of several ECU variants in a car and different car lines.
8. **SecureOnboardCommunication**: Provides mechanisms for securing the communication on in-vehicle networks (e.g., communication between the car and the outside world).
9. **SafetyExtensions**: Provides mechanisms to realizing and documenting functional safety of AUTOSAR systems (e.g. according to ISO 26262).

10. **DecentralizedConfiguration**: Extension of the AUTOSAR meta-model to support transfer of diagnostic needs of OEMs to suppliers by use of AUTOSAR-complaint models.
11. **IntegrationOfNonARSystems**: Enables integration of non-AUTOSAR systems, e.g., Genivi, into an AUTOSAR system during development.
12. **NVDataHandlingRTE**: Provides efficient mechanisms for the software components to handle non-volatile data.
13. **EcuMFixedMC**: Provides support for ECU state handling on ECUs with multiple cores.
14. **AsilQmProtection**: Provides means for protecting modules developed according to safety regulations from other potentially unsafe modules (i.e., it reduces the chance of error propagation to safety-critical modules).

Figure 4.20 shows the results of the *NoC* measure calculated for each of these 14 features for all AUTOSAR meta-model templates.

This figure shows two important aspects of the AUTOSAR meta-model evolution related to new concepts. First, they have very different impact on the AUTOSAR meta-model (see left part of the figure), i.e., some concepts have no impact at all, such as the concept of *IntegrationOfNonARSystems*, while some have a significant impact, such as the concept of *DecentralizedConfiguration*. Second, we can see that the vast majority of changes in R4.2.1 is related to new concepts (see right part of the figure) and only a small part to other changes, e.g., fixes of errors related to existing concepts.

Finally, in order to present the results of role-based assessment of impact of the AUTOSAR R4.2.1 concepts, we considered the following design roles in AUTOSAR-based development, whose work is described in Sect. 4.3 [DST15]:

- *Application Software Designer:* Responsible for the definition of software components and their data exchange points (involved in the phases of physical system and physical ECU design; see (2) and (3) in Fig. 4.2).

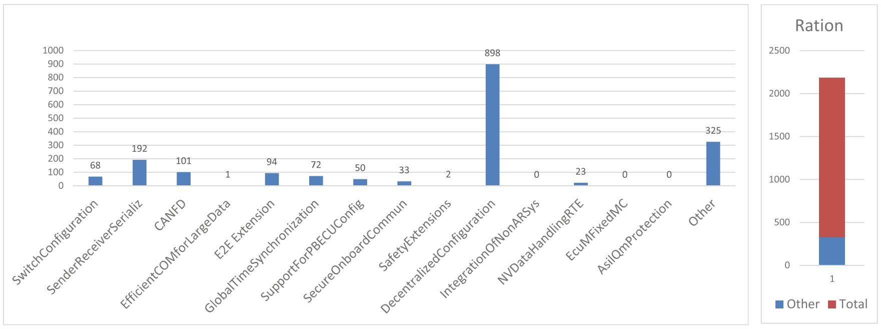

Fig. 4.20 Number of changes per concept (TPS)

- *Diagnostic Designer:* Responsible for the definition of diagnostic services required by software components (involved in the phases of physical system and physical ECU design; see (2) and (3) in Fig. 4.2).
- *ECU Communication Designer:* Responsible for the definition of signals and their transmission on electronic buses in different frames (involved in the phase of physical COM design; see (5) in Fig. 4.2).
- *Basic Software Designer:* Responsible for the design of BSW modules and their interfaces (involved in the phase of BSW development; see (7) in Fig. 4.2).
- *COM Configurator:* Responsible for the configuration of communication BSW modules (involved in the phase of BSW configuration; see (6) in Fig. 4.2).
- *Diagnostic configurator:* Responsible for the configuration of diagnostic BSW modules (involved in the phase of BSW configuration; see (6) in Fig. 4.2).

Figure 4.21 shows the impact of 13 new concepts of AUTOSAR R4.2.1 (all except the concept of *SupportForPBLAndPBSECUConfiguration*, as it causes a significantly higher number of changes in comparison to other concepts, thereby obscuring the results) on these six roles.

This figure shows the following interesting points. First, the *COM Configurator* role followed by the *ECU Communication Designer* role are mostly affected by the concepts, i.e., roles related to the communication between different ECUs. Then, we can see that the majority of concepts do not have impact on all roles, except for the concept of *DecentralizedConfiguration*. Finally, we can see that some concepts, e.g., *IntegrationOfNonARSystems* and *AsilQmProtection*, do not have impact on any of the major roles. The concept of *IntegrationOfNonARSystems* represents a methodological guideline without the actual impact on the models while the concept of *AsilQmProtection* has impact on other safety-related basic software modules that are not explicitly related to ECU communication and diagnostics.

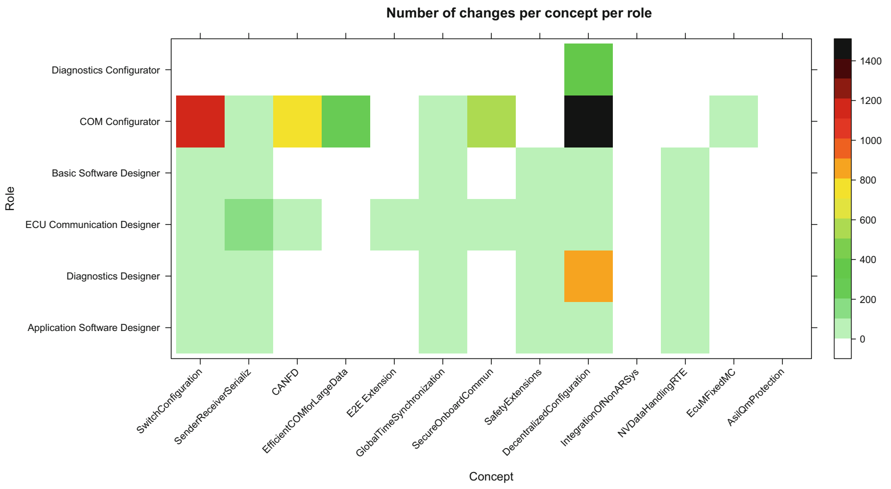

Fig. 4.21 Number of changes per concept affecting different roles

4.6.2 AUTOSAR Requirements Evolution

As already explained, AUTOSAR defines two main types of requirements:

- Design requirements in the AUTOSAR template specifications (*TPS requirements*) that define semantics for the AUTOSAR meta-model elements. They include both specification items and constraints checked by the modeling tools.
- Functional middleware requirements in the AUTOSAR basic software specifications (*BSW requirements*) that define functionality of the AUTOSAR BSW modules, e.g., *Com*, *Dem* and *Dcm*.

The evolution of the AUTOSAR TPS requirements is tightly related to the evolution of the AUTOSAR meta-model, as the introduction, modification and removal of meta-classes also require introduction, modification and removal of the supporting requirements for their use. This evolution mostly affects the work of OEMs and Tier1s in the system, ECU and COM design phases of the development. The evolution of the AUTOSAR BSW requirements indicates the functional changes in the ECU middleware that is developed by Tier2s in the basic software development phase and to some extent Tier3s related to the development of drivers for the chosen hardware. In this subsection, we present the analysis of evolution of both types of AUTOSAR requirements for different AUTOSAR releases in branch 4.x [MDS16]. We first show in Fig. 4.22 the increase in the number of requirements in the AUTOSAR templates (left) and BSW specifications (right).

There are two interesting points that can be observed from this figure. First, we can see a constant increase in the number of both TPS and BSW requirements in the new AUTOSAR releases. This indicates that the standard is still growing, i.e., new features are being incorporated into the standard. Second, we can see a relatively steady increase in the number of BSW requirements without disruptive changes in their number between consecutive releases. On the other hand, we can see a big increase (almost three times) in the number of TPS requirements between AUTOSAR R4.0.2 and R4.0.3 and then again almost a double increase between R4.0.3 and R4.1.1 that indicates that the evolution of the TPS and BSW requirements does not follow the same trend. This large increase in the

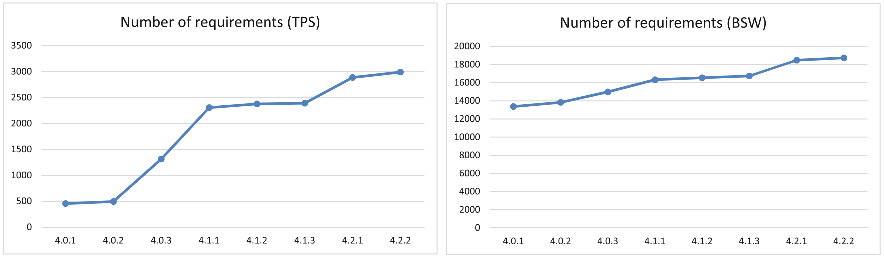

Fig. 4.22 Number of TPS and BSW requirements

number of TPS requirements is partially related to the immaturity of the older AUTOSAR template specifications, where a lot of plain text had to be converted into specification items and constraints.

We already showed that AUTOSAR continues to grow by standardizing new features. In order to assess the influence of new features on the evolution of both template and BSW specifications, we analyzed the number of added, modified and removed TPS and BSW requirements. Figure 4.23 shows the results for the TPS requirements for different releases of the AUTOSAR 4.x branch.

We can see that the evolution of the AUTOSAR TPS specifications is mostly driven by introduction of new requirements (specification items and constraints). This is especially the case with minor releases of AUTOSAR, but also in R4.0.3. Removal of TPS requirements is not common, similarly to removal of meta-classes from the AUTOSAR meta-model, as it may affect backwards compatibility of the AUTOSOAR models that is kept high within one major release.

Figure 4.24 shows the number of added, modified and removed BSW requirements for different releases of the AUTOSAR 4.x branch.

The results confirm that the evolution of the AUTOSAR BSW specifications is also mostly driven by the introduction of new BSW requirements (i.e., new basic software features). However, we can see that there are generally more modifications

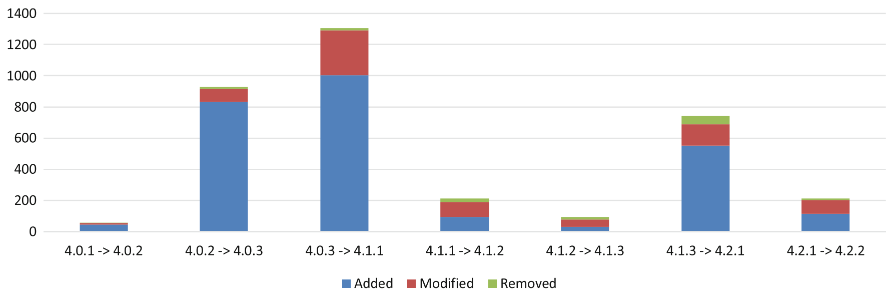

Fig. 4.23 Number of added, modified and removed requirements (TPS)

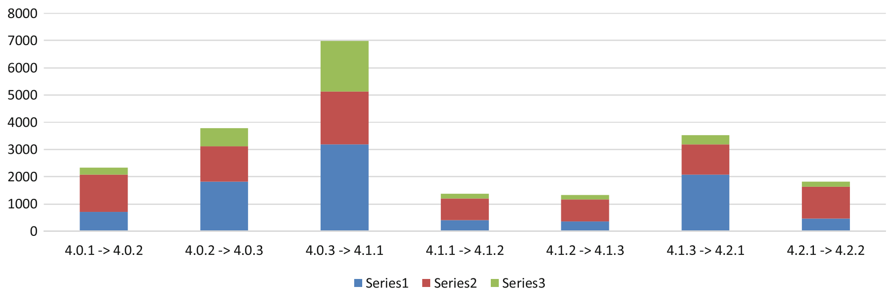

Fig. 4.24 Number of added, modified and removed requirements (BSW)

and removals of BSW requirements than was the case with TPS requirements. High modification to the BSW requirements could indicate lower stability of the AUTOSAR basic software. High removals of the BSW requirements could indicate that certain features become obsolete, probably due to the introduction of newer features that provide the same or similar functionality. Another reason behind more removals of BSW requirements in comparison to TPS requirements is related to the relaxed backwards-compatibility requirements of the AUTOSAR basic software in comparison to the AUTOSAR models, as AUTOSAR models are exchanged between several roles in the development process (e.g., OEM and Tier1) while basic software modules are usually developed by the role of Tier2 only.

4.7 Future of AUTOSAR

The results of the AUTOSAR evolution analysis presented in the previous section show that it is strongly driven by innovations in the form of new features incorporated into the standard. This trend is expected to continue in the future and will be expanded, considering the plans of AUTOSAR for maintaining two platforms in parallel—a classic platform that represents the continuation of the work on the branch 4.x and an adaptive platform that represents a new platform with the goal to satisfy future needs of the automotive industry. The classic platform aims to stabilize and improve the existing AUTOSAR features while the adaptive platform aims to anticipate the future by identifying technological trends and key features for AUTOSAR.

The subsequent release of the classic platform was R4.3.0, planned for the last quarter of 2016. Examples of new concepts that will be supported in this release are presented below:

1. *MacroEncapsulationOfLibraryCalls:* Simplifies handling of interpolation routines provided by libraries (automated selection and parametrization).
2. *CryptoInterface:* Develops the strategy for security SW-/HW-Interface to support technology trends such as Car-2-X communication and autonomous drive.
3. *V2XSupport:* Enables implementation of Intelligent Transportation Systems (ITS) applications [ETS16] as AUTOSAR software components and their integration into an AUTOSAR ECU (Ethernet-based V2X-stack).
4. *ProfilesForDataExchangePoints:* Improves the interoperability between AUTOSAR tools by providing means for describing which data is expected for a given data exchange point.
5. *DecentralizedConfigurationExt01:* Extends the concept of "Decentralized Configuration", which provides a top-down configuration of diagnostics via diagnostic extract, with features such as On-Board Diagnostics (OBD).
6. *ExtendedBufferAccess:* Extends the existing rapid prototyping functionality (quick validation of a software algorithm in an ECU context without the need for a production build) with support for bypassing the RTE.

7. *PolicyManager:* Allows specification of security policies in AUTOSAR ARXML (e.g., insurance company can read data but not modify it).
8. *DLTRework:* Improves the Development Error Tracer (DLT) module that provides generic logging and tracing functionality for the software components, RTE, DEM module, etc.
9. *SOMEIPTransportProtocol:* Defines a protocol for segmentation of Scalable Service-Oriented Middleware over IP (SOME/IP) [Völ13] packets that are larger than 128 kBytes.

Together with these new features, the release will bring improvements to the existing features by fixing a number of issues in the related specifications.

The first release of the adaptive platform is planned for the first quarter of 2017 and will be following a different naming schema that consist of the release year and month, e.g., *R17-03*. Its main goal is to ensure the fulfillment of the 2020 expectations from the automotive industry which state that all major vehicle innovations will be driven by electrical systems. The list of selected main functional drivers for the AUTOSAR's adaptive platform is presented below [AUT16b]:

1. *Highly automated driving:* Support driving automation levels 3–4 according to the NHSTA (National Highway Safety Traffic Administration) [AUT13], i.e., limited driving automation where the driver is occasionally expected to take the control and full driving automation where the vehicle is responsible for performing the entire trip. This includes support for cross-domain computing platforms, high-performance micro-controllers, distributed and remote diagnostics, etc. The levels of autonomous functionality are described further in Sect. 9.2.
2. *Car-2-X applications:* Support interaction of vehicles with other vehicles and off-board systems. This includes support for designing automotive systems with non-AUTOSAR ECUs based on Genivi, Android, etc.
3. *Vehicle in the cloud:* Support vehicle to cloud communication. This includes the development of secured on-board communication, security architecture and secure cloud interaction.
4. *Increased connectivity:* Support increased connectivity of the automotive software systems and other non-AUTOSAR and off-board systems. This includes support for dynamic deployment of software components and common methodology of work regardless of whether the system is on-board or off-board.

The idea behind adaptive cars is depicted in Fig. 4.25 [AUT16b]. The figure shows several classic AUTOSAR ECUs ("C") that are responsible for common vehicle functionalities, e.g., engine or brake control units. The figure also shows several non-AUTOSAR ECUs ("N") that are responsible for infotainment functionalities or communication with the outside world (e.g., Genivi or Android ECUs). Finally, the figure shows certain adaptive AUTOSAR ECUs ("A") that are responsible for realization of advanced car functionalities that usually require inputs or provide outputs to both classic and non-AUTOSAR ECUs, such as car-2-X applications. These ECUs are commonly developed following agile development methodologies and require more frequent updates and run-time configuration.

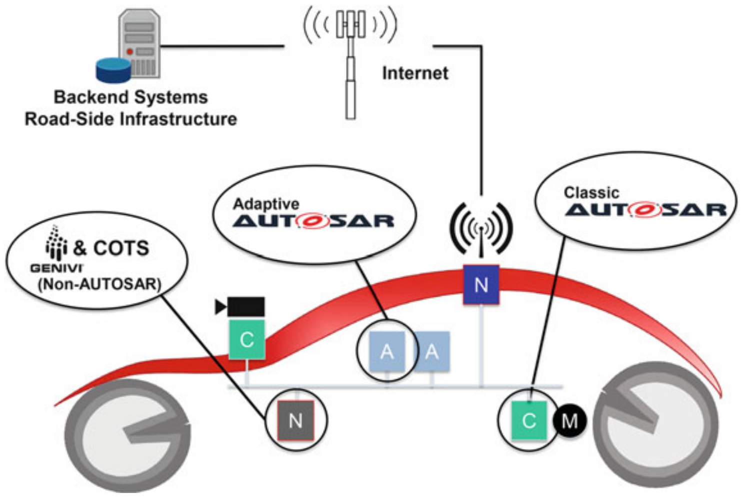

Fig. 4.25 Adaptive AUTOSAR vehicle architecture

Considering the functional drivers for the adaptive platform and the idea behind the adaptive vehicle architecture explained above, adaptive ECUs are expected to be designed using the following principles and technologies (the list is not exhaustive):

- Agile software development methodology based on dynamic software updates. This enables a continuous functional development mode that starts with a minimum viable product.
- Fast addition of new features (application software components) deployed in different packages. This enables fast software innovation cycles.
- Secured service-oriented point-to-point communication. This enables dynamic updates of application software, where new software components subscribe to existing services via a service discovery protocol.
- Wireless updates of the application software. This enables "on the road" software updates without the need for taking the car to a workshop.
- Support for run-time configuration. This enables dynamic adaption of the system based on available functionality.
- High bandwidth for inter-ECU communication (Ethernet). This enables faster transmission of large data.
- Switched networks (Ethernet switches). This enables smart data exchange between different Ethernet buses.
- Micro-processors with external memory instead of micro-controllers. This enables higher amounts of memory and peripherals that can be extended.
- Multi-core processors, parallel computing and hardware acceleration. This enables faster execution of vehicle functions.

- Integration with classic AUTOSAR ECUs or other non-AUTOSAR ECUs (e.g., Genivi, Android). This enables unanimous design of heterogeneous automotive software systems.
- Execution models of access freedom, e.g. full access or sandboxing. This enables security mechanism for separating running programs from each other, e.g., safety- and security-critical programs from the rest.

AUTOSAR plans to achieve this using the adaptive ECU architecture presented in Fig. 4.26 [AUT16b]:

As was the case with classic AUTOSAR platform, AUTOSAR standardizes the middleware layer of the adaptive platform that is referred to as the *Adaptive AUTOSAR Services* layer. However, this layer is organized in functional clusters rather than as a detailed description of the modules (internal structure of the clusters), which enables platform-independent design of the software architectures. Orange clusters represent parts of the ECU architecture that will be standardized in the first release of the adaptive platform, while gray clusters represent parts of the ECU architecture that will be standardized in the later releases.

We can see that, apart from the *Operating system* that is based on POSIX and standardized *Bootloader* for downloading software to the ECU, AUTOSAR plans to deliver *Execution Management*, *Logging and Tracing*, *Diagnostics* and *Communications* functional clusters. The *Execution Management* cluster is responsible for starting/stopping applications related to different car modes and it is based on threads rather than runnables. The *Logging and Tracing* cluster is responsible for collecting information about different events such as the ones related to safety or

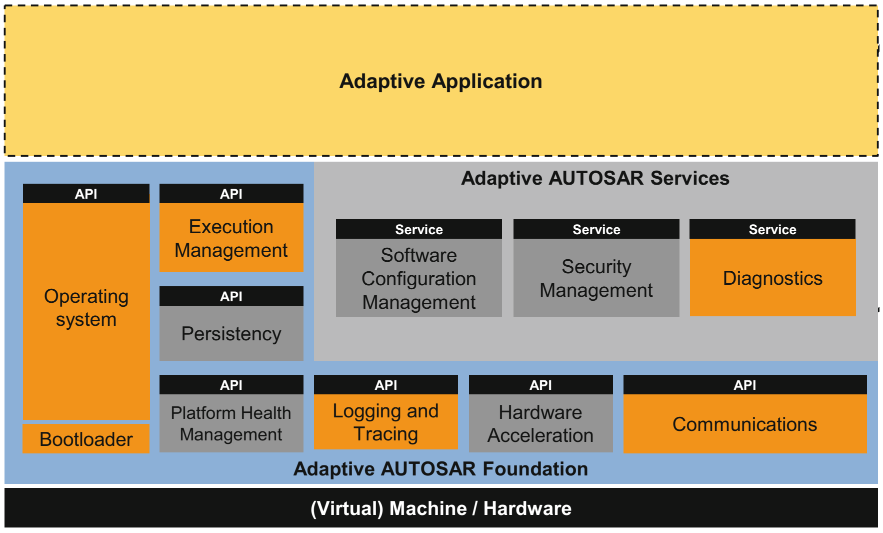

Fig. 4.26 AUTOSAR adaptive ECU architecture

security. As in the classic AUTOSAR platform, the *Diagnostics* cluster is responsible for collection of diagnostic event data which is now possible to exchange with the diagnostic backend. Finally, the *Communications* cluster is responsible for service-oriented communication between ECUs connected via Ethernet (SOME/IP protocol).

The big improvement in the standardization process of the AUTOSAR adaptive platform is the validation of new features before their standardization. This means that AUTOSAR will form a group of engineers that will create prototypes of the new features based on specifications that are planned to be released and provide feedback to AUTOSAR about their feasibility. This ensures agility of the development process within the AUTOSAR consortium as well.

4.8 Further Reading

For those who would like details about AUTOSAR, it is important to understand that AUTOSAR is a huge standard with over 200 specifications and more than 20,000 requirements, so it is nearly impossible to be an expert in all of its features. AUTOSAR's specifications are divided into *standard* and *auxiliary* specifications, where only the standardized ones are required to be followed for achieving full AUTOSAR compliance. Nevertheless, both standardized and auxiliary specifications could be of interest to the readers who would like to learn specifics about the AUTOSAR standard.

We recommend all AUTOSAR beginners to start reading the *Layered Software Architecture* document [AUT16g], as it defines high-level features of AUTOSAR that should be known before diving deeper into other specifications. The AUTOSAR's *Methodology* specification [AUT16h] could be a natural continuation as it contains descriptions of the most important artifacts that are created by different roles in the AUTOSAR development process. However, it also contains many details that may not be understandable at this point, so it should be skimmed through, with us focusing on the familiar topics.

The rest of the readings are specific to the interest topic of the reader. Readers interested in the architectural design of automotive software systems should look into AUTOSAR's template specifications (TPSs). For example, if they are interested in the logical system/ECU design, they should take a look at the AUTOSAR *Software Component* template [AUT16j] in order to understand how to define application software components and their data exchange points. Some general concept used in all templates could be found in the *Generic Structure* template [AUT16f], but it is probably best to follow references from the template that is being read to the concrete section in the *Generic Structure* templates. This is because understanding the entire document at once could be challenging. There is no real need to look at the actual AUTOSAR meta-model specified in UML, as all relevant information and diagrams are exported to the AUTOSAR template specifications.

Readers interested in the functionalities of the AUTOSAR basic software should read the software specifications (SWS) of the relevant basic software modules. For example, if they are interested in the ECU diagnostic functionality, they should take a look at the AUTOSAR *Diagnostic Event Manager* [AUT16d] and *Diagnostic Configuration Manager* [AUT16c] specifications. Requirements applicable to all basic software modules can be found in the *General Requirements on Basic Software Modules* specification [AUT16e].

On a higher granularity level, design requirements from the TPS specifications can be traced to the more formalized requirements from the requirements specifications (RS) documents. Similarly, functional basic software requirements from the SWS specifications can be traced to the more formalized requirements from the software requirements specifications (SRS) documents [MDS16]. RS and SRS requirements can be traced to even higher-level specifications such as the ones describing general AUTOSAR features and AUTOSAR's objectives. However, we advise AUTOSAR beginners to stick to the TPS and SWS specifications, at least at the beginning, as they are the ones that contain explanations and diagrams needed for understanding the AUTOSAR features in detail.

There are two additional general recommendations that we could give to readers who want to learn more about AUTOSAR. First, AUTOSAR specifications are not meant to be read from the beginning until the end. It is therefore recommended to switch between different specifications in a search for explanations related to a particular topic. Second, the readers should always read the latest AUTOSAR specifications as they contain up-to-date information about the current features of the AUTOSAR standard.

Apart from the specifications released by AUTOSAR, readers interested in knowing more about the AUTOSAR standard could find useful information in a few scientific papers. Related to the AUTOSAR methodology, Briciu et al. [BFH13] and Sung et al. [SH13] show an example of how AUTOSAR software components shall be designed according to AUTOSAR and Boss et al. [Bos12] explain in more detail the exchange of artifacts between different roles in the AUTOSAR development process, e.g., OEMs and Tier1s.

Related to the AUTOSAR meta-model, Durisic et al. [DSTH16] analyze the organization of the AUTOSAR meta-model and show possible ways in which it could be re worked in order to be compliant with the theoretical meta-modeling concept of strict meta-modeling. Additionally, Pagel et al. [PB06] provide more details about the generation of the AUTOSAR's XML schema from the AUTOSAR meta-model, and Brorkens et al. [BK07] show the benefits of using XML as an AUTOSAR exchange format.

Related to the configuration of AUTOSAR basic software, Lee et al. [LH09] explain further the use of the AUTOSAR meta-model for the configuration of AUTOSAR basic software modules. Finally, Mjeda et al. [MLW07] connect the phases of automotive architectural design based on AUTOSAR and functional implementation of the AUTOSAR software component in Simulink.

4.9 Summary

Since its beginning in 2003, AUTOSAR soon became a world-wide standard in the development of automotive software architectures, accepted by most major car manufacturers in the world. In this chapter, we explained the reference layered system architecture defined by AUTOSAR that is instantiated in dozens of car ECUs, and how different architectural components are usually developed according to the AUTOSAR methodology. We showed the role of the AUTOSAR meta-model in the design of the architectural components and the exchange of architectural models between different parties in the automotive development process. We also described major components of the AUTOSAR middleware layer (basic software) and how they could be configured.

Towards the end of the chapter, we visualized the evolution of the AUTOSAR standard by analyzing its meta-model and requirements changes between the latest AUTOSAR releases, and showed that the standard is still growing by standardizing new features. We also showed how AUTOSAR plans to support future cars functionalities, such as autonomous drive and car-to-car communication, by presenting ideas behind the AUTOSAR adaptive platform.

In our future work we plan to further analyze the differences between AUTOSAR classic and adaptive platforms on the meta-model and requirements levels, and the impact of using both platforms in the design of the automotive software systems.

References

AK03. C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *Journal of IEEE Software*, 20(5):36–41, 2003.

AUT13. AUTOSAR, http://www.nhtsa.gov. *National Highway Traffic Safety Administration*, 2013.

AUT16a. AUTOSAR, www.autosar.org. *Automotive Open System Architecture*, 2016.

AUT16b. AUTOSAR, www.autosar.org. *AUTOSAR Adaptive Platform for Connected and Autonomous Vehicles*, 2016.

AUT16c. AUTOSAR, www.autosar.org. *Diagnostic Communication Manager v4.2.2*, 2016.

AUT16d. AUTOSAR, www.autosar.org. *Diagnostic Event Manager v4.2.2*, 2016.

AUT16e. AUTOSAR, www.autosar.org. *General Requirements on Basic Software Modules v4.2.2*, 2016.

AUT16f. AUTOSAR, www.autosar.org. *Generic Structure Template v4.2.2*, 2016.

AUT16g. AUTOSAR, www.autosar.org. *Layered Software Architecture v4.2.1*, 2016.

AUT16h. AUTOSAR, www.autosar.org. *Methodology Template v4.2.2*, 2016.

AUT16i. AUTOSAR, www.autosar.org. *Release Overview and Revision History v4.2.2*, 2016.

AUT16j. AUTOSAR, www.autosar.org. *Software Component Template v4.2.2*, 2016.

BFH13. C. Briciu, I. Filip, and F. Heininger. A New Trend in Automotive Software: AUTOSAR Concept. In *Proceedings of the International Symposium on Applied Computational Intelligence and Informatics*, pages 251–256, 2013.

BG01. Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *International Conference on Automated Software Engineering*, pages 273–280, 2001.

BK07. M. Brörkens and M. Köster. Improving the Interoperability of Automotive Tools by Raising the Abstraction from Legacy XML Formats to Standardized Metamodels. In *Proceedings of the European Conference on Model Driven Architecture-Foundations and Applications*, pages 59–67, 2007.

BKPS07. M. Broy, I. Kruger, A. Pretschner, and C. Salzmann. Engineering Automotive Software. In *Proceedings of the IEEE*, volume 95 of *2*, 2007.

Bos12. B. Boss. Architectural Aspects of Software Sharing and Standardization: AUTOSAR for Automotive Domain. In *Proceedings of the International Workshop on Software Engineering for Embedded Systems*, pages 9–15, 2012.

DST15. D. Durisic, M. Staron, and M. Tichy. ARCA - Automated Analysis of AUTOSAR Meta-Model Changes. In *International Workshop on Modelling in Software Engineering*, pages 30–35, 2015.

DSTH14. D. Durisic, M. Staron, M. Tichy, and J. Hansson. Evolution of Long-Term Industrial Meta-Models - A Case Study of AUTOSAR. In *Euromicro Conference on Software Engineering and Advanced Applications*, pages 141–148, 2014.

DSTH16. D. Durisic, M. Staron, M. Tichy, and J. Hansson. Addressing the Need for Strict Meta-Modeling in Practice - A Case Study of AUTOSAR. In *International Conference on Model-Driven Engineering and Software Development*, 2016.

ETS16. ETSI, www.etsi.org. *Intelligent Transport Systems*, 2016.

Gou10. P. Gouriet. Involving AUTOSAR Rules for Mechatronic System Design. In *International Conference on Complex Systems Design & Management*, pages 305–316, 2010.

Kru95. P. Kruchten. Architectural Blueprints - The “4+1” View Model of Software Architecture. *IEEE Softwar*, 12(6):42–50, 1995.

Küh06. T. Kühne. Matters of (Meta-) Modeling. *Journal of Software and Systems Modeling*, 5(4):369–385, 2006.

LH09. J. C. Lee and T. M. Han. ECU Configuration Framework Based on AUTOSAR ECU Configuration Metamodel. In *International Conference on Convergence and Hybrid Information Technology*, pages 260–263, 2009.

LLZ13. Y. Liu, Y. Q. Li, and R. K. Zhuang. The Application of Automatic Code Generation Technology in the Development of the Automotive Electronics Software. In *International Conference on Mechatronics and Industrial Informatics Conference*, volume 321–324, pages 1574–1577, 2013.

MDS16. C. Motta, D. Durisic, and M. Staron. Should We Adopt a New Version of a Standard? - A Method and its Evaluation on AUTOSAR. In *International Conference on Product Software Development and Process Improvement*, 2016.

MLW07. A. Mjeda, G. Leen, and E. Walsh. The AUTOSAR Standard - The Experience of Applying Simulink According to its Requirements. *SAE Technical Paper*, 2007.

NDWK99. G. Nordstrom, B. Dawant, D. M. Wilkes, and G. Karsai. Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments. In *IEEE Conference on Engineering of Computer Based Systems*, pages 68–74, 1999.

Obj04. Object Management Group, www.omg.org. *MOF 2.0 Core Specification*, 2004.

Obj14. Object Management Group, http://www.omg.org/mda/. *MDA guide 2.0*, 2014.

PB06. M. Pagel and M. Brörkens. Definition and Generation of Data Exchange Formats in AUTOSAR. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 52–65, 2006.

SH13. K. Sung and T. Han. Development Process for AUTOSAR-based Embedded System. *Journal of Control and Automation*, 6(4):29–37, 2013.

Völ13. L. Völker. SOME/IP - Die Middleware für Ethernet-basierte Kommunikation. *Hanser automotive networks*, 2013.