Document Title	Application Design Patterns Catalogue
Document Category	Technical report (TR)
Document UID	672
Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R26-11

Document Change History

Date	Release	Changed by	Description
2026-11-26	R26-11	AUTOSAR Release Management	• No content changes
2025-11-27	R25-11	AUTOSAR Release Management	• Editorial changes
2024-11-27	R24-11	AUTOSAR Release Management	• Editorial changes
2023-11-23	R23-11	AUTOSAR Release Management	• Editorial changes
2022-11-24	R22-11	AUTOSAR Release Management	• Table for interface definitions introduced • Usage options for SensorActuator-Pattern • Editorial changes
2021-11-25	R21-11	AUTOSAR Release Management	• Editorial changes
2020-11-30	R20-11	AUTOSAR Release Management	• Signal quality states introduction • Extension of definition of electrical sensor interface

△

2019-11-28	R19-11	AUTOSAR Release Management	• Subfunctions per layer defined • Capability information introduced • FAQ and known issues section implemented • Separation of Sensor and Actuator in namespace • Changed Document Status from Final to published
2018-10-31	4.4.0	AUTOSAR Release Management	• Editorial changes
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes
2016-11-30	4.3.0	AUTOSAR Release Management	• generalization of arbitration pattern, three examples: several setpoint requesters, several providers of estimated values, several providers of consolidated values • minor changes
2015-07-31	4.2.2	AUTOSAR Release Management	• reconsideration of signal definitions and tailored pattern for smart actuators and actuators with no feedback loop • specification items added • minor changes
2014-10-31	4.2.1	AUTOSAR Release Management	• First Release of document. Patterns covered: – Sensor and Actuator Pattern – Arbitration of Several Set-point Requester Pattern • Previously published as part of EXP_AIPowertrain

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction 8
1.1 Document conventions . . . 8
2 About Patterns 10
2.1 Types of Pattern . . . 10
2.2 Describing Patterns . . . 10
3 Sensor and Actuator Pattern 12
3.1 Motivation . . . 12
3.2 Also Known As . . . 12
3.3 Applicability . . . 12
3.4 Solution . . . 13
3.5 Naming . . . 17
3.6 Interface definitions . . . 22
3.7 Sensor Actuator types . . . 23
3.7.1 Sensor . . . 24
3.7.2 Actuator without Feedback Loop . . . 24
3.7.3 Actuator with Feedback Loop . . . 25
3.8 Usage of pattern . . . 26
3.9 Examples . . . 27
3.9.1 Throttle Valve . . . 27
3.9.2 Turbo Charger . . . 28
3.9.3 Turbo Charger with Stages and Banks . . . 29
3.9.4 Sensor for Environment Temperature . . . 30
3.9.5 Standard Sensor . . . 31
3.9.6 Standard Sensor for Environment Temperature . . . 32
3.9.7 Distributing Device Abstraction . . . 33
3.10 Sample Code and Model . . . 35
3.11 Typical location of some common function within the specified layers . . . 37
3.11.1 Virtual Device Coordinator (DevCoorrVirt) . . . 37
3.11.1.1 Conversion and linearization of physical requested value . . . 38
3.11.1.2 DCM service / Diagnostic tester interface for basic function test 38
3.11.1.3 Cleaning / Ice breaking . . . 39
3.11.1.4 Dither of setpoint . . . 39
3.11.1.5 Release function of setpoint . . . 39
3.11.1.6 Coordination of activation and deactivation of the actuator . . . 39
3.11.2 Actuator Device Driver (DevDrvrActr) . . . 40
3.11.2.1 Dither of output value . . . 40
3.11.2.2 Release function of output value . . . 40
3.11.2.3 Limitation . . . 41
3.11.2.4 Feed forward controller . . . 41
3.11.2.5 Closed loop controller . . . 41

3.11.2.6 Set point limitation . . . 41
3.11.2.7 Set point gradient limitation . . . 41
3.11.2.8 Control deviation monitoring . . . 41
3.11.2.9 Capability . . . 42
3.11.3 Electrical Actuator Driver (DrvrActrElec) . . . 43
3.11.3.1 Power stage monitoring . . . 44
3.11.4 Virtual Device Driver (DevSnsrVirt) . . . 44
3.11.4.1 Substitution . . . 44
3.11.4.2 Inertia compensation . . . 45
3.11.4.3 Signal qualifier evaluation . . . 45
3.11.4.4 DCM service / Diagnostic tester interface for basic function test 45
3.11.4.5 Plausibilization . . . 45
3.11.5 Sensor Device Driver (DevDrvrSnsr) . . . 46
3.11.5.1 High level filtering . . . 46
3.11.5.2 Offset adaption . . . 46
3.11.5.3 Zero point adaption . . . 47
3.11.5.4 Drift detection . . . 47
3.11.5.5 Conversion . . . 47
3.11.5.6 Physical signal gradient calculation . . . 47
3.11.5.7 Physical signal gradient check . . . 47
3.11.5.8 Stuck check diagnosis . . . 47
3.11.5.9 Physical signal range check . . . 48
3.11.6 Electrical Sensor Driver (DrvrSnsrElec) . . . 48
3.11.6.1 Basic filter . . . 48
3.11.6.2 Voltage compensation . . . 49
3.11.6.3 Electrical diagnosis . . . 49
3.12 Known Issues . . . 49
3.13 FAQ . . . 49
3.14 Known Uses . . . 49
3.15 Related Patterns . . . 50
3.16 Anti-Patterns One Should be Aware of . . . 50
3.17 Further Readings . . . 50
4 Arbitration between several requesters or providers 51
4.1 Problem . . . 51
4.2 Applicability . . . 51
4.3 Solution . . . 51
4.4 Examples . . . 54
4.4.1 Several Setpoint Requesters . . . 54
4.4.2 Several Providers of Consolidated Values . . . 55
4.4.3 Several Providers of Estimated Values . . . 57
4.5 Sample Code and Model . . . 59
4.6 Known Uses . . . 59
4.7 Related Patterns . . . 59

5 Signal Quality States 60
5.1 Problem . . . 60
5.2 Applicability . . . 60
5.3 Solution . . . 60
5.3.1 Implementation proposal . . . 61
A Change history of AUTOSAR traceable items 62
A.1 Traceable item history of this document according to AUTOSAR Release R24-11 . . . 62
A.1.1 Added Specification Items in R24-11 . . . 62
A.1.2 Changed Specification Items in R24-11 . . . 62
A.1.3 Deleted Specification Items in R24-11 . . . 62
A.2 Traceable item history of this document according to AUTOSAR Release R23-11 . . . 62
A.2.1 Added Specification Items in R23-11 . . . 62
A.2.2 Changed Specification Items in R23-11 . . . 62
A.2.3 Deleted Specification Items in R23-11 . . . 62
A.3 Traceable item history of this document according to AUTOSAR Release R22-11 . . . 62
A.3.1 Added Specification Items in R22-11 . . . 62
A.3.2 Changed Specification Items in R22-11 . . . 63
A.3.3 Deleted Specification Items in R22-11 . . . 63
A.3.4 Added Constraints in R22-11 . . . 63
A.3.5 Changed Constraints in R22-11 . . . 63
A.3.6 Deleted Constraints in R22-11 . . . 63
B Mentioned Class Tables 64

References

[1] ANTLR parser generator V3
http://www.antlr.org

[2] Standardization Template
AUTOSAR_FO_TPS_StandardizationTemplate

[3] SW-C and System Modeling Guide
AUTOSAR_CP_TR_SWCModelingGuide

[4] XML Specification of Application Interfaces
AUTOSAR_CP_MOD_AISpecification

[5] Architectural Pattern
http://en.wikipedia.org/wiki/Architectural_pattern

[6] Software Design Pattern
http://en.wikipedia.org/wiki/Software_design_pattern

[7] Design Pattern
http://en.wikipedia.org/wiki/Design_Pattern

[8] Anti Pattern
http://en.wikipedia.org/wiki/Anti-pattern

[9] Software Design Pattern Template
http://c2.com/cgi/wiki?DesignPatternTemplate

[10] Secure Design Patterns
http://www.sei.cmu.edu/reports/09tr010.pdf

[11] Software Component Template
AUTOSAR_CP_TPS_SoftwareComponentTemplate

[12] Layered Software Architecture
AUTOSAR_CP_EXP_LayeredSoftwareArchitecture

1 Introduction

1.1 Document conventions

Technical terms (Class Names) are typeset in mono spaced font, e.g. `FrameTriggering`.

When defining name patterns the syntax defined according to ANTLR is used [1]. The grammar for name patterns as defined in [2], [TPS_STDT_00055], is used. In the following we just list the most important placeholders that are used throughout the document:

anyName This represents a string which is valid `shortName` according to `Identifier`

anyNamePart This represents a string (([a-zA-Z0-9]|_[a-zA-Z0-9])*_?) which is valid part of a `shortName`.

Hint: The place holder "anyNamePart" shall not be used at the beginning of a `shortName` pattern to avoid invalid `shortName`s.

blueprintName This represents the `shortName` / `shortLabel` / `symbol` of the applied blueprint

componentName This represents the `shortName` of the BSW module resp. ASW SwComponentType / ASW component prototype related to the derived object. "Related" mainly could be both, aggregating or referencing.

The placeholder `componentName` in particular supports multiple derivation of a `PortPrototypeBlueprint` in the context of different software component types resp. modules [TPS_STDT_00036].

componentTypeName This represents the `shortName` of the dedicated `SwComponentType`.

componentPrototypeName This represents the `shortName` of the dedicated `SwComponentPrototype`.

index This represents a numerical index applicable for example to arrays.

keyword This represents the `abbrName` of a keyword acting as a name part of the short name [TPS_STDT_00004].

For a complete description see [2], [TPS_STDT_00055]. Additionally we assume that the naming rules as defined in [3] are fulfilled. If applicable and available the keywords used in names are those standardized in [4].

Additionally we extend the grammar using the following place holders:

anyLongName This represents a string which is a valid `longName`.

Additionally we assume that AUTOSAR name pattern is fulfilled. This means that the long name starts with a capital letter and that all words except articles (e.g.

”a”, ”the”), prepositions (e.g. ”at”, ”by”, ”to”) and conjunctions (e.g. ”and”, ”or”) start with a capital letter as well.

anyLongNamePart This represents a string which is a valid part of a `longName`.

2 About Patterns

This document gives an overview of the patterns defined in AUTOSAR for ease the usage of AUTOSAR architecture, AUTOSAR application interfaces and the AUTOSAR meta-model. The focus is on application software (ASW).

2.1 Types of Pattern

The following categories/classifications of patterns are distinguished:

Architectural Pattern An architectural pattern is a standard design in the field of software architecture. The concept of an architectural pattern has a broader scope than the concept of design pattern. The architectural patterns address various issues in software engineering, such as computer hardware performance limitations, high availability and minimization of a business risk [5].

Design Pattern In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer must implement themselves in the application [6].

Solution Pattern A solution pattern describes a generic solution for a specific problem like for example error handling or job scheduling [5].

An orthogonal classification of patterns is the following:

Design Patterns A design pattern in architecture and computer science is a formal way of documenting a solution to a design problem in a particular field of expertise [7].

Anti-Patterns In software engineering, an anti-pattern (or anti-pattern) is a pattern used in social or business operations or software engineering that may be commonly used but is ineffective and/or counterproductive in practice [8].

2.2 Describing Patterns

The description of the patterns in this document follow a predefined structure. This structure was created based on the contents of the documents [6], [9], [10], [1], and [2].

A pattern is described in a separate section and the header of the particular pattern contains the name of the pattern and the pattern identification (standardized name): {pattern name} ({pattern identification})

At the very beginning of the section describing a specific pattern the classification is given as shown below:

Classification {type of pattern} Pattern

The type of the pattern is one of the categories described in section 2.1.

Section	Mandatory	Instruction	Additional Information
Problem	Yes	The problem solved by the design pattern and its general rationale and purpose.	None
Also Known As	No	Other names for the pattern, if any are known.	None
Applicability	Yes	A general description of the characteristics a system must have for the pattern to be useful in the design or implementation of the program.	Indications: something you notice, hinting that this pattern may be applicable Contraindications: something that would indicate that this pattern would not be applicable
Solution	Yes	A textual or graphical description of the pattern. This provides a detailed specification of the structural aspects of the pattern, using appropriate notations.	Also think about *Overdose Effect*: what undesirable thing happens if you keep applying the suggested action over and over and over and over. Also think about *Side Effects*: new problems that you might expect to crop up upon applying the solution, or new issues that come to the fore.
Naming	No	Describes naming pattern that are usable or should be used in the context of the pattern.	Name pattern follow syntax defined according to ANTLR like it was decided to use in [2], e.g. in [TPS_STDT_00055].
Example	Yes	Example how to apply the pattern.	None
Sample Code and Model	No	Code or model providing an example of how to implement the pattern.	None
Known Uses	No	Examples of the use of the pattern, taken from existing systems or literature.	None
Related Patterns	No	Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.	Other patterns that relate, either superordinate, subordinate, competitor, or neighboring patterns, with references to where they can be found.
Anti-Patterns	No	Anti-Patterns you should be aware of.	None
Reading	No	Further material worthwhile to know.	None

Table 2.1: Pattern Description Template

3 Sensor and Actuator Pattern

Classification Design Pattern

3.1 Motivation

The Sensor/Actuator Design Pattern describes how to handle sensors or actuators that are connected to an ECU in the context of an overall architecture.

The main intention of this pattern is standardizing application interfaces for SWC controlling sensors and actuators, it focuses on aspects of:

- Independence of application software from concrete sensors and actuators connected to a specific ECU.
- Reusable code between different sensors and actuators.
- Different code sharing cooperation models (software sharing), thus supporting different business models.
- Deployment of functionality to different ECUs.

For standardizing interfaces it is useful to have an architectural design overview of a sensor/actuator composition. Therefore it was decided to create an architectural design pattern first and define the interface inside next. In a first step a layer model containing the main interfaces between those layers is created. Then the most common functions within the layers are defined and described for a common understanding in a second step. In the third step it is planned to describe also the interfaces in these functions from step 2.

The pattern in general is a strong recommendation but is not mandatory to be followed. The interfaces which are standardized as a result from the pattern will be reserved exactly for the described usecase and shall not be used for other purpose even if the pattern is not followed.

3.2 Also Known As

This pattern is also known as *Device Abstraction*.

3.3 Applicability

The *Device Abstraction* is located above the RTE. It is a set of software components that abstracts from the sensors and actuators connected to a specific ECU. It uses sensor actuator software components, the only components above RTE that are allowed to access the ECU abstraction interface.

In case direct access to the Micro controller is required because specific interrupts and/or complex Micro controller peripherals to fulfill the special functional and timing requirements of the sensor evaluation or actuator control have to be implemented this pattern cannot be applied. Instead a complex driver implementation shall be used.

The Sensor/Actuator Design Pattern supports software sharing (=collaboration between various partners) on different levels: Development partner one might deliver the sensors together with the basic electrical driver software (DrvrSnsrElec), development partner two might deliver the sensor device driver software (DevDrvrSnsr) and the third partner might develop the substitute models together with the virtual device drivers (DevSnsrVirt). There might be different suppliers for the same Sensor/Actuator or there might be sensors/actuators from different vendors used within one and the same system.

In case software sharing shall not be supported it is also possible to just implement the interfaces of the composition of a single sensor or actuator but not following the internal three-level-architecture.

The Sensor/Actuator-Pattern also supports different deployment scenarios to ECUs. One ECU might provide the measured value of a sensor whereas another ECU is implementing the model that calculates the estimated value that may substitute the measured sensor value.

Note: In general a pattern is not applied without any changes but with extension by combining several patterns to one solution. For example:

- The composition pattern (splitting of component if they are getting too large and are not maintainable any longer) is combined with this pattern.
- The diagnosis pattern is combined with this pattern.

3.4 Solution

In Figure 3.1 that was taken from [11] an example of the signal flow for a lamp (actuator) and a velocity sensor is shown. This signal flow pattern is refined by this sensor/actuator pattern.

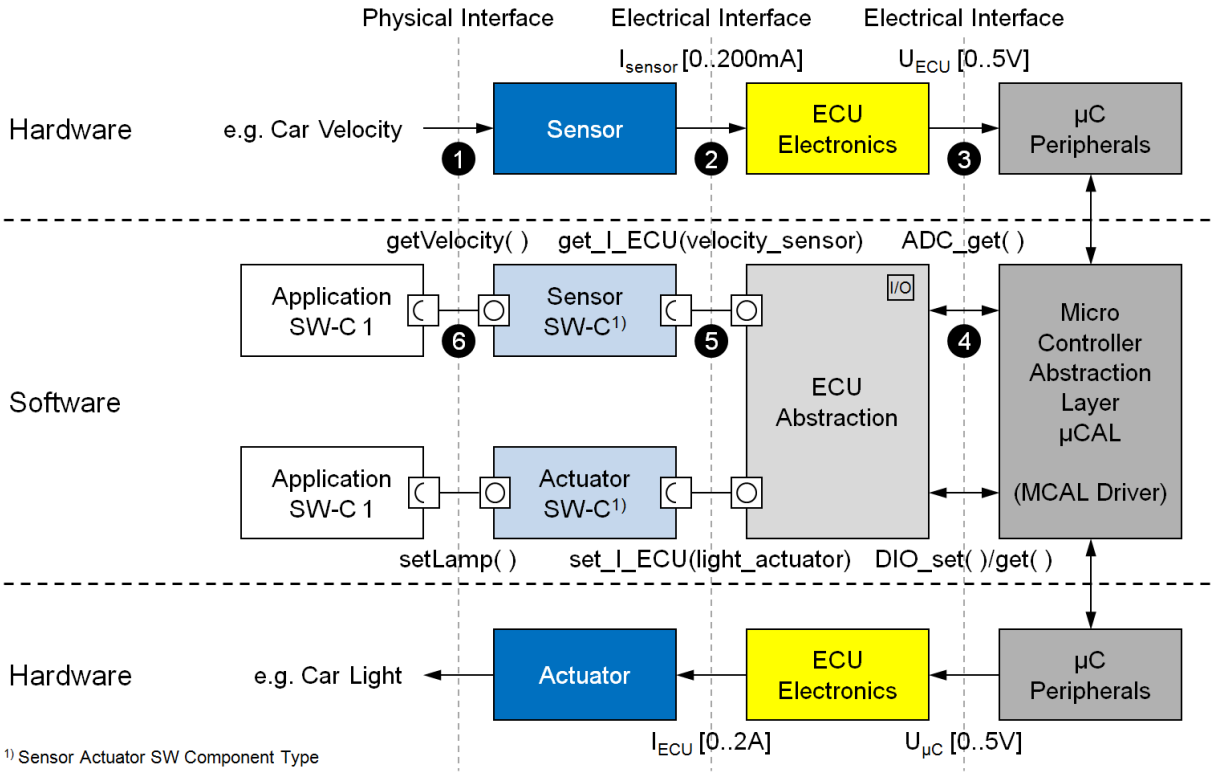


Figure 3.1: Sensor Actuator Signal Flow [11]

The solution is proposing a three-level layering within a composition representing a sensor or actuator:

- electrical device driver layer,
- sensor/actuator device driver layer,
- virtual device driver layer.

Each layers can be represented by a single `SwComponentType` or also by a `CompositionSwComponentType` containing one or more `SwComponentType`s. The electrical device driver layer in addition must contain at least one `SensorActuatorSwComponentType`.

In Figure 3.2 the overall structure of the pattern is shown. Recursive elements are optional. Closed loop controlled actuator and position feedback is included. The naming is simplified and will be explained in more detail later.

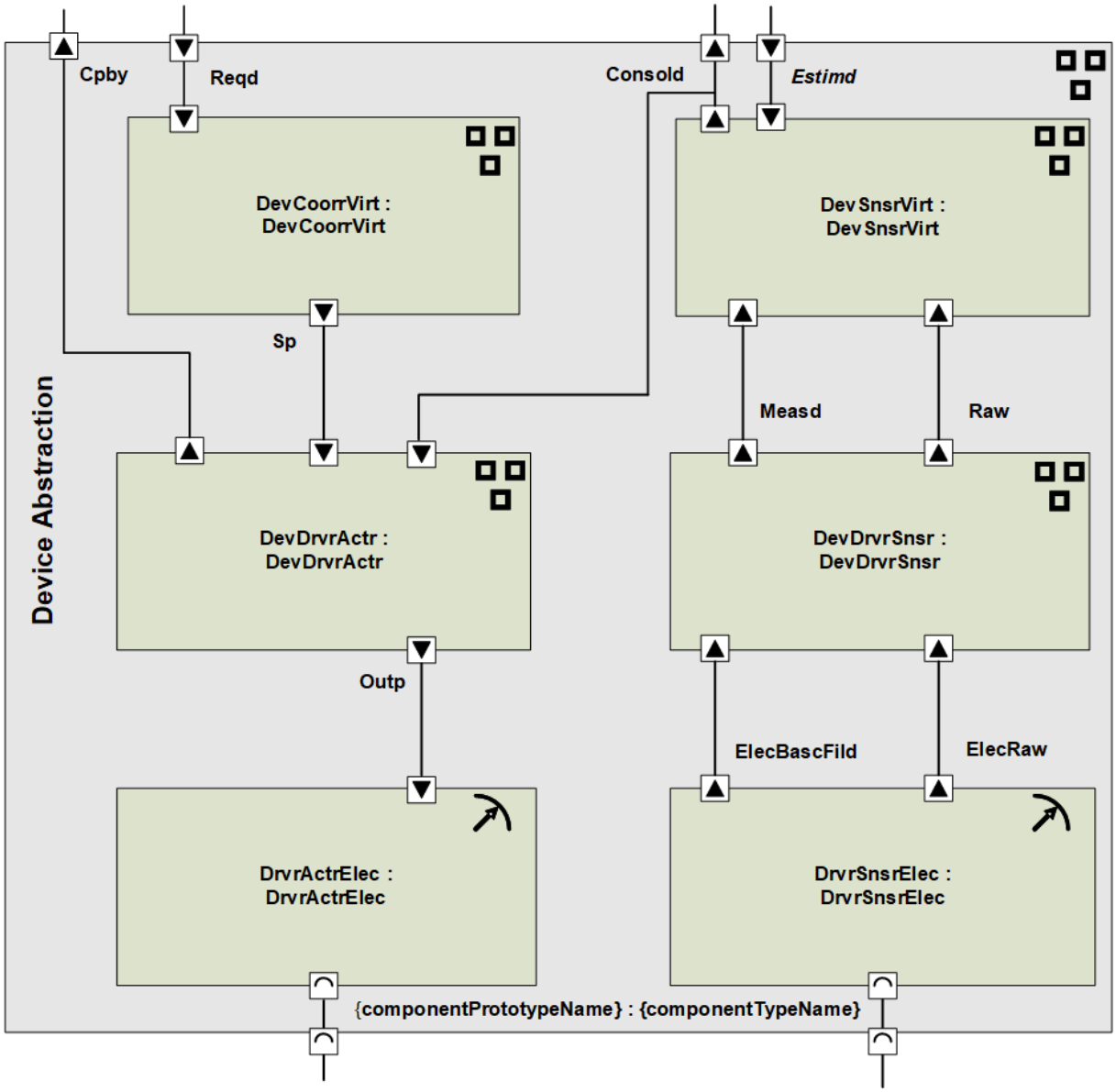


Figure 3.2: Sensor Actuator Pattern for Closed Loop

The application software can rely on the existence of the consolidated value. The consolidated value can be calculated from the

- estimated value,
- setpoint value,
- measured and/or raw value.

The calculation of the consolidated value via the setpoint or estimated value is used in case of actuators without feedback loop. In Figure 3.6 an example of an actuator without feedback loop calculating the consolidated value from the setpoint value is shown. Besides actuators with open loop control there are also smart actuators that can directly deal with the setpoint value itself. In this case the device driver actuator SW-C and the electrical driver actuator SW-C are only routing the setpoint value since

the controlling of the actuator and thus the calculating of the output value etc. is realized within the smart actuator itself. However, the two layers, electrical device layer and device driver layer, are additionally needed because of diagnosis etc.

The pattern can be tailored for a standard sensor. In this case the consolidated value (Consold) is provided and the estimated value (Estimd) is requested, see Figure 3.5.

The signal flow is shown in Figure 3.3: The electrical raw value is requested from the ECU Abstraction. After basic filtering the signal is converted to a physical value representing the measured value. If the measured value is not suitable for the application the estimated value might be chosen to be the consolidated value, i.e. the value that can be used by the rest of the application software. Some applications request to explicitly know about the physical raw value. This is why this signal is also made available.

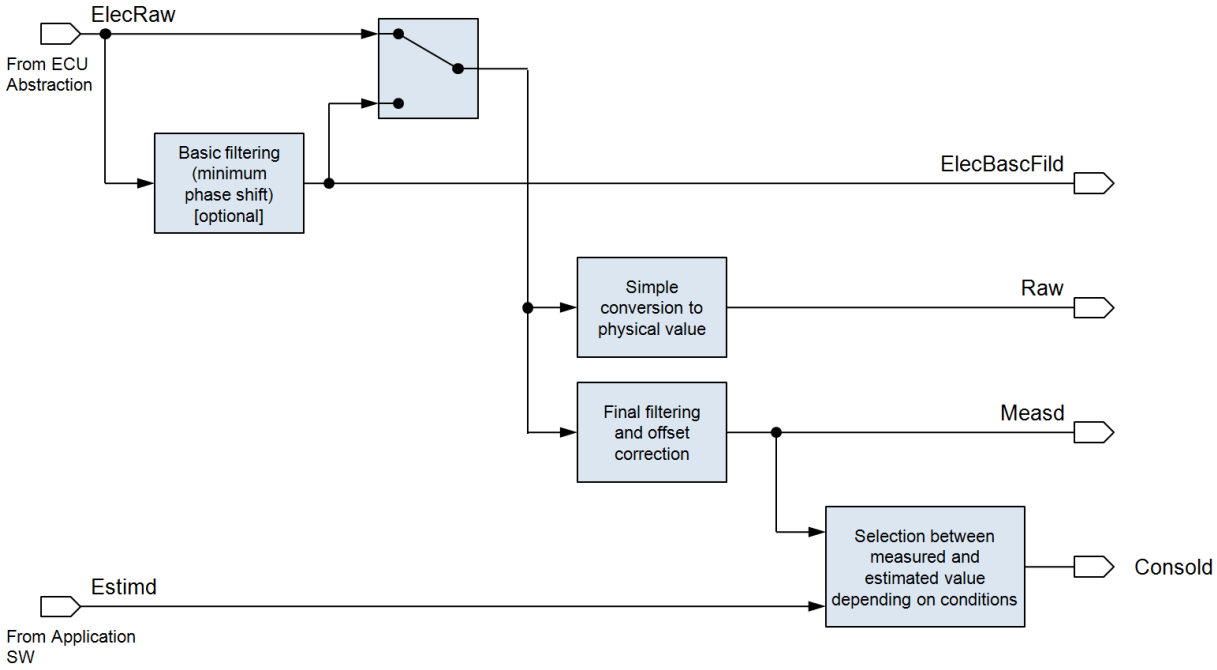


Figure 3.3: Signal Flow within Sensor and Actuator Pattern

Please be aware: `SensorActuatorSwComponentType`s are the only components that are allowed to access ECU Abstraction Software, namely `EcuAbstractionSwComponentType`. This is shown in Figure 3.4 taken from [12]. Access is denoted by ”IO”.

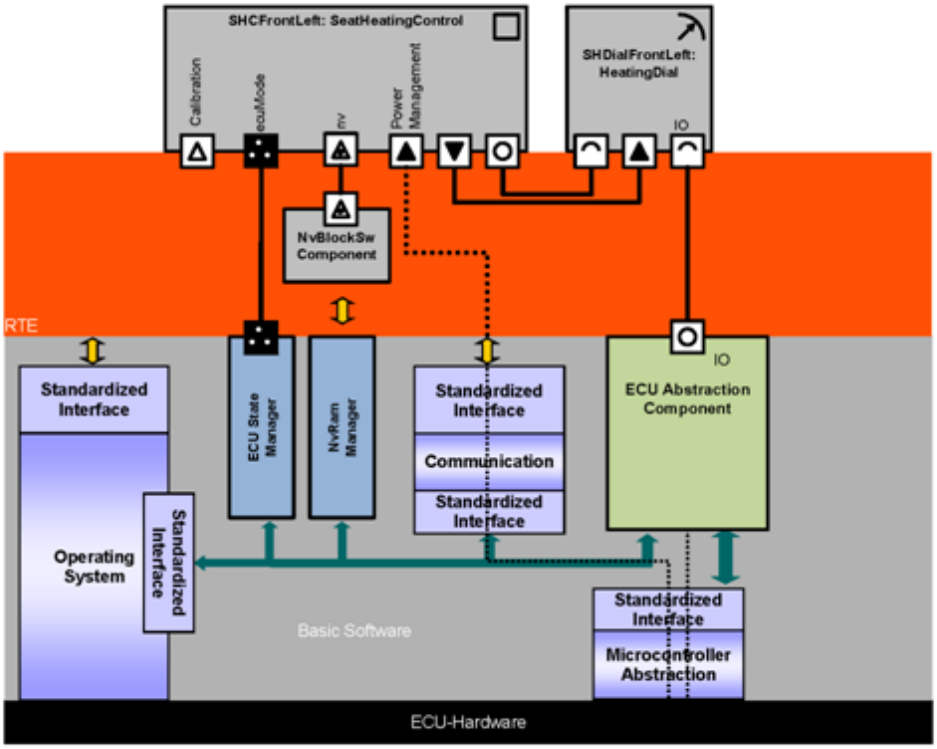


Figure 3.4: Access to ECU Abstraction

3.5 Naming

In the following the semantic port prototype (blueprint) definition together with the name patterns are described.

The overall name pattern for port short names is described in grammar 3.1. In the following these port (prototype blueprint) names are also referred to as signal names. In Table 3.1 additionally the pattern for the corresponding long names is given.

```
grammar PSnsrActrPortNames;

portName
   :   {'sensorActuatorSignal'} ;

sensorActuatorSignal
   :   {anyName}{'sensorActuatorSignalType'} ;

sensorActuatorSignalType
   :   ( ElecRaw | ElecBascFild | Raw | Measd | Consold | Estimd | Outp |
       Sp | Reqd ) ;

anyName
```

```
:    ('keyword')* ;
```

Listing 3.1: Name Pattern for Ports in Device Abstraction

In addition to port prototypes, interface names are extended with sequence number. Usage and handling of sequence numbers is described in [3] in [TR_SWNR_00044].

```
grammar PSnsrActrPortInterfaceNames;

interfaceName
:    {'sensorActuatorSignal'} ;

sensorActuatorSignal
:    {anyName}{'sensorActuatorSignalType'}{sequenceNumber} ;

sensorActuatorSignalType
:    ( ElecRaw | ElecBascFild | Raw | Measd | Consold | Estimd | Outp | Sp |
     Reqd ) ;

anyName
:    ('keyword')* ;
```

Listing 3.2: Name Pattern for PortInterfaces in Device Abstraction

In case of a generic long name {anyLongNamePart} or {anyLongName}, resp., is empty.

Generic Signal Name	Long Name Pattern of Concrete Sensor/Actuator Signal (EN)	Generic Long Name of Signal (EN)	AUTOSAR Definition
`ElecRaw`	Electrical Raw Value of {anyLongNamePart}	Electrical Raw Value	Electrical raw sensor value as provided by the ECU Abstraction. Typically this value is unfiltered. However, there are for example smart components doing some filtering themselves. This signal can only be represented in voltage, current, (period) time, binary value, frequency, dutycycle [11].
`ElecBascFild`	Electrical Basic Filtered Value of {anyLongNamePart}	Electrical Basic Filtered Value	Basic filtered electrical raw sensor value (e.g. maximum allowed phase shift is one scheduling raster or maximum 360 degree crankshaft rotation if exhaust gas pulsation dependent). Electrical representation of a technical signal [11]. This signal can only be represented in voltage, current, (period) time, binary value, frequency, dutycycle.
`Raw`	Raw Value of {anyLongNamePart}	Raw Value	Physical raw/base sensor value. Simple conversion of basic filtered electrical (`ElecBascFild`) to physical value.
`Measd`	{anyLongName} (Measured)	Measured Value	Final filtered and offset corrected physical sensor value. Physical sensor value/standard sensor value. The physical sensor value is the linearized/filtered physical raw/base sensor value including offset. At this step a (significant) phase-shift could be possible.

Consold	{anyLongName}	Value	Consolidated physical value, either a measured value (Measd) or a modeled value (Estimd). Final filtered and offset corrected consolidated actuator value/physical sensor value. Virtual physical sensor value/fused sensor value that comes as close as possible to the technical signal. In case of inability to provide a physical sensor value (e.g. failure, implausibility or other reasons) a substitute value/default value or a frozen value is provided.
Estimd	{anyLongName} (Estimated)	Estimated Value	Modelled value physical sensor value/standard sensor value. Can be used as a replacement for final filtered and offset corrected physical sensor value. The interface is optional.
Outp	Output of {anyLongNamePart}	Output Value	Final controller output (closed loop or open loop). It includes the necessary control actions to reach the requested setpoint in the given system conditions. For example for realizing the requested actuator position a precontrol impulse to overcome the static friction is needed. In case of a smart actuator the output value might add a dedicated initialization duty cycle to wakeup the actuator. Typically expressed as percentage.
Sp	Setpoint {anyLongNamePart}	Setpoint Value	Final actuator setpoint. Typically expressed as percentage.
Reqd	Requested Setpoint {anyLongNamePart}	Requested Setpoint	Final requested physical setpoint. Typically expressed as percentage but could also be expressed e.g. as factor.
Cpby	Capability {anyLongNamePart}	Capability	Provides the dynamic instant capability typically based on output limitation but could also contain the limitation on rate of change of the consolidated value. It is expressed as percentage.

Table 3.1: Signal Names and Semantics

Some examples of short and long names for sensor/actuator signals or ports, resp., are given in Table 3.2.

Short Name	Class	Long Name (EN)
TrboChrgrReqd	PortPrototype	Requested Setpoint for Turbo Charger
Consold	PortPrototype	Consolidated Value
TrboChrgr	PortPrototype	Value of Turbo Charger

Table 3.2: Port Names Examples

In grammar 3.3 the pattern for component types and component prototypes for the atomic components within a composition representing a sensor or an actuator is described.

In some cases there might be parts of the implementation that can be reused for different sensors/actuators. Therefore the name pattern for the component type name is more generic and does not necessarily contain the Sensor/Actuator name. In other cases the Sensor/Actuator names are not sufficient to make the component type names unique so an additional identifier can be added to the component type name.

```
grammar PSnsrActrAtomicSwcShortName;

sensorActuatorComponentTypeName
   :   sensorActuatorComponentName ;

sensorActuatorComponentPrototypeName
   :   sensorActuatorComponentName ;

sensorActuatorComponentName
   :   (Drvr{Device}Elec | DevDrvr{Device} | Dev{Device}Virt | DevCoorrVirt
       )('anyNamePart') ;

Device
   :   ( Snsr | Actr ) ;

anyNamePart
   :   ('keyword')* ;
```

Listing 3.3: Name Pattern for Atomic Software Component Types in Device Abstraction

In grammar 3.4 the pattern is more refined but still conforming to grammar 3.3 because ”For” is a standardized keyword. Note: the refined grammar requests that field blocks are concatenated by adding an appropriate preposition.

```
grammar PSnsrActrAtomicSwcShortNameRefined;

sensorActuatorComponentTypeName
   :   sensorActuatorComponentName ;

sensorActuatorComponentPrototypeName
   :   sensorActuatorComponentName ;

sensorActuatorComponentName
   :   (Drvr{deviceType}Elec | DevDrvr{deviceType} | Dev{deviceType}Virt |
      DevCoorrVirt) ({device}) ;

deviceType
   :   ( Snsr | Actr ) ;

device
   :   ( For{sensor}('anyNamePart') | For{actuator}('anyNamePart') ) ;

sensor
   :   'anyName' ;

actuator
   :   'anyName' ;

anyName
   :   ('keyword')* ;
```

```
anyNamePart
   :   ('keyword')* ;
```

Listing 3.4: Refined Name Pattern for Atomic Software Component Types in Device Abstraction

In grammar 3.5 the pattern for the corresponding English long names of the components is described.

```
grammar PSnsrActrAtomicSwcLongName;

sensorActuatorComponentLongName
   :   sensorActuatorComponentName ;

sensorActuatorComponentLongName
   :   ('anyLongName') ( Electrical Sensor Driver | Sensor Device Driver |
       Virtual Device Driver | Electrical Actuator Driver | Actuator Device
       Driver | Virtual Device Coordinator) ('anyLongNamePart') ;

anyLongName
   :   ('keyword')* ;

anyLongNamePart
   :   ('keyword')* ;
```

Listing 3.5: Pattern for English Long Names Atomic Software Component Types in Device Abstraction

In Table 3.3 the generic sensor and actuator component short and long names are shown as pairs.

Generic Short Name Pattern	Generic Long Name (EN)
DrvrSnsrElec	Electrical Sensor Driver
DevDrvrSnsr	Sensor Device Driver
DevSnsrVirt	Virtual Device Driver
DrvrActrElec	Electrical Actuator Driver
DevDrvrActr	Actuator Device Driver
DevCoorrVirt	Virtual Device Coordinator

Table 3.3: Sensor and Actuator Component Name Patterns

Short Name	Class	Long Name (EN)
DrvrActrElecForTle8209	SensorActuatorSw-ComponentType	TLE8209: Electrical Sensor Driver
DrvrActrElecForTrboChrgr	SwComponentPrototype	Turbo Charger: Electrical Sensor Driver
DevSnsrVirtForAnyTSnsr	ApplicationSwComponent-Type	Virtual Device Driver for Any Temperature Sensor
DevSnsrVirtForTrboChrgr	SwComponentPrototype	Turbo Charger: Virtual Device Driver
TrboChrgrAcmeT064	CompositionSwComponent-Type	Turbo Charger: ACME T064

Table 3.4: Examples for Sensor and Actuator Names

In grammar 3.6 a pattern is described how to refine 'anyNamePart' as defined in grammar 3.4 in case of a system with several banks and stages. In Table 3.5 corresponding name examples are shown using this grammar part.

```
grammar PSnsrActrStgBnkShortNames;

stageBank
   :   (Stg{'indexStg'}(AtBnk{'indexBnk'}) ;

indexStg
   :   ( 1st | 2nd | 3rd ) ;

indexBnk
   :   ( 1st | 2nd | 3rd ) ;
```

Listing 3.6: Name Pattern for Signals in Device Abstraction in Case of a System with Several Banks

Short Name	Class	Long Name (EN)
TrboChrgrStg3rdAtBnk1st	PortPrototype	Value of Turbo Charger at Third Stage at First Bank
TrboChrgrStg3rdAtBnk2nd	SwComponentPrototype	Turbo Charger at Third Stage at Second Bank

Table 3.5: Sensor, Actuator and Port Names examples in Case of a System with Several Banks

3.6 Interface definitions

The following table defines the default attribute for all previously described PortInterfaces. SI units shall be used for SensorActuator PortInterfaces whenever possible.

Interface name	ImplDatatype	CompuMethod	Unit	PhysicalDimension
Reqd	sint16	$100/(2^{13})$	Perc (%)	Ratio
Cpby	sint16	$100/(2^{13})$	Perc (%)	Ratio
Sp	sint16	$100/(2^{13})$	Perc (%)	Ratio
Outp	sint16	$100/(2^{13})$	Perc (%)	Ratio
Measd, Consold, Estimd, Raw (binary)	n.a.	1	-	Dimensionless
Measd, Consold, Estimd, Raw (T)	float32	1	Kelvin (K)	Absolute Temperature / Relative Temperature
Measd, Consold, Estimd, Raw (P)	float32	1	Pa (Pa)	Pressure
Measd, Consold, Estimd, Raw (Posn)	float32	1	Perc (%)	Ratio
Measd, Consold, Estimd, Raw (I)	float32	1	Ampr (A)	Electric Current
Measd, Consold, Estimd, Raw (N)	float32	1	PerSec (1/s)	Rotational Speed
Measd, Consold, Estimd, Raw (U)	float32	1	Volt (V)	Voltage

Measd, Consold, Estimd, Raw (R)	float32	1	Ohm (Ohm)	Electric Resistance
Measd, Consold, Estimd, Raw (Pwr)	float32	1	Watt (W)	Power
Measd, Consold, Estimd, Raw (RelHum)	float32	1	Perc (%)	Ratio
Measd, Consold, Estimd, Raw (AbsHum)	float32	1	KiloGrPerMtrCubd (kg/m^3)	Mass Density
Measd, Consold, Estimd, Raw (Tq)	float32	1	NwtMtr (Nm)	Torque
Measd, Consold, Estimd, Raw (Conc)	float32	1	MolPerMtrCubd (mol/m^3)	Concentration
ElecRaw, ElecBascFild (binary)	n.a.	1	-	Dimensionless
ElecRaw, ElecBascFild (I)	float32	1	Ampr (A)	Electric Current
ElecRaw, ElecBascFild (Ti)	float32	1	s	Time
ElecRaw, ElecBascFild (Frq)	float32	1	Hz	Frequency
ElecRaw, ElecBascFild (DutyCyc)	float32	1	Perc (%)	Ratio
ElecRaw, ElecBascFild (U)	float32	1	Volt (V)	Voltage

Table 3.6: Interface Properties

3.7 Sensor Actuator types

The following S/A types are available as SwComponentTypes:

- Sensor (Snsr)
- Actuator without feedback (Actr)
- Actuator with feedback (ActrWithFb)

3.7.1 Sensor

In Figure 3.5 a design pattern of blueprint components for a sensor is shown.

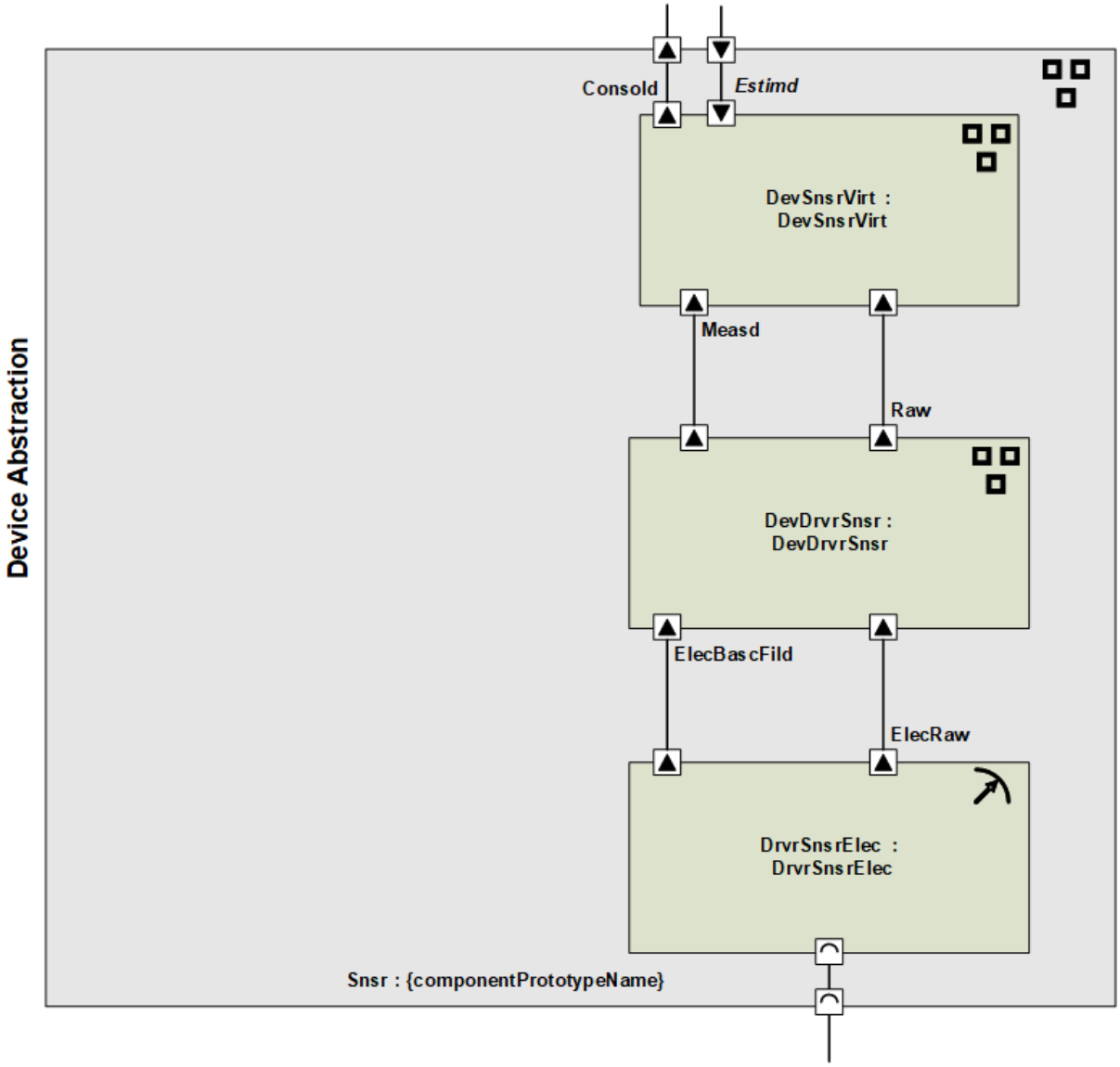


Figure 3.5: Device Abstraction for Sensor

3.7.2 Actuator without Feedback Loop

In Figure 3.6 an open loop controlled actuator is shown. In addition to the actuator chain it is possible to also provide and calculate the consolidated value using

the setpoint input, but there are alternatives how to calculate the consolidated value.

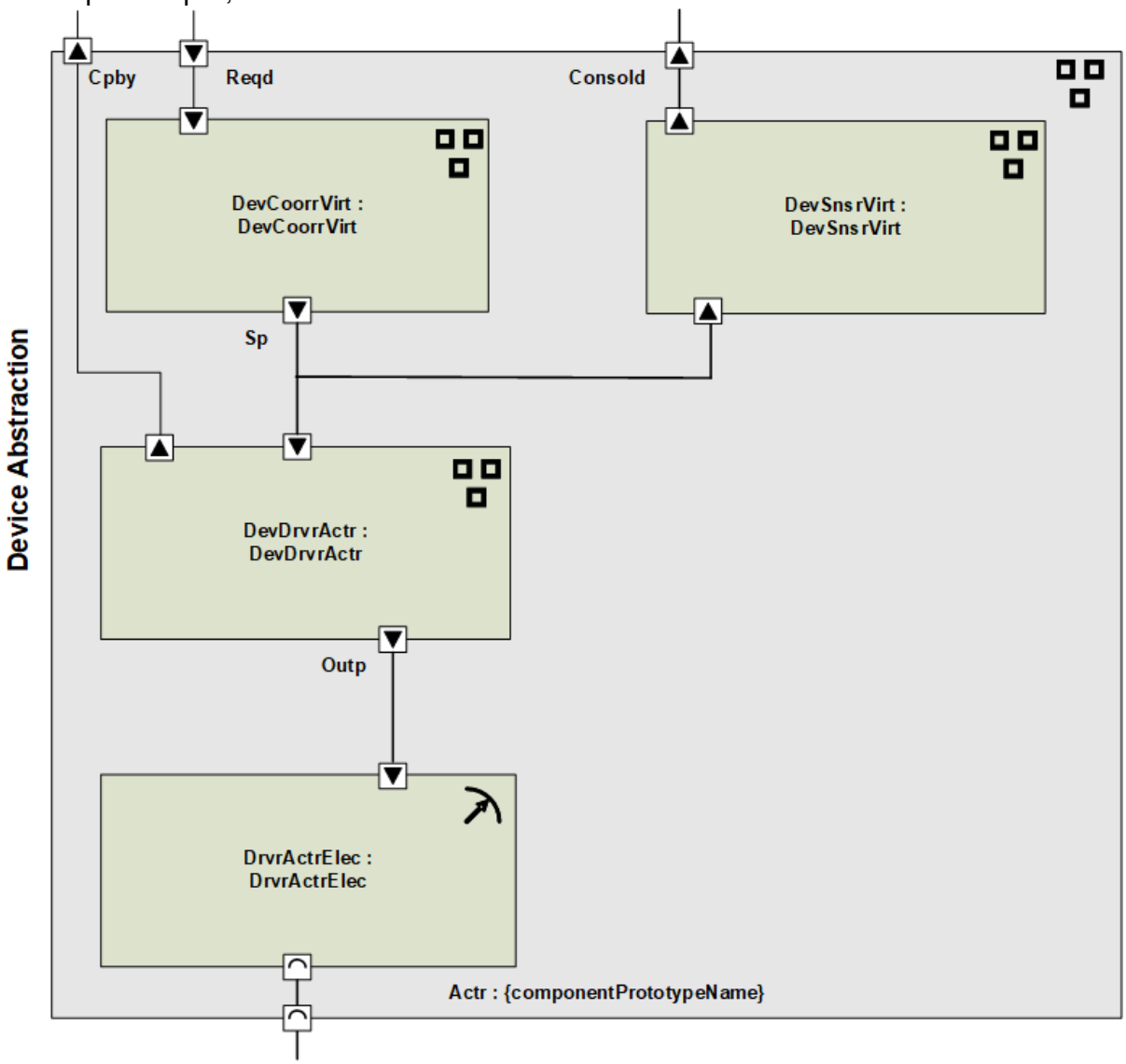


Figure 3.6: Example Actuator without Feedback Loop (Setpoint Alternative)

3.7.3 Actuator with Feedback Loop

In Figure 3.7 a design pattern of blueprint components for a typical closed loop controlled device with (position) feedback is shown.

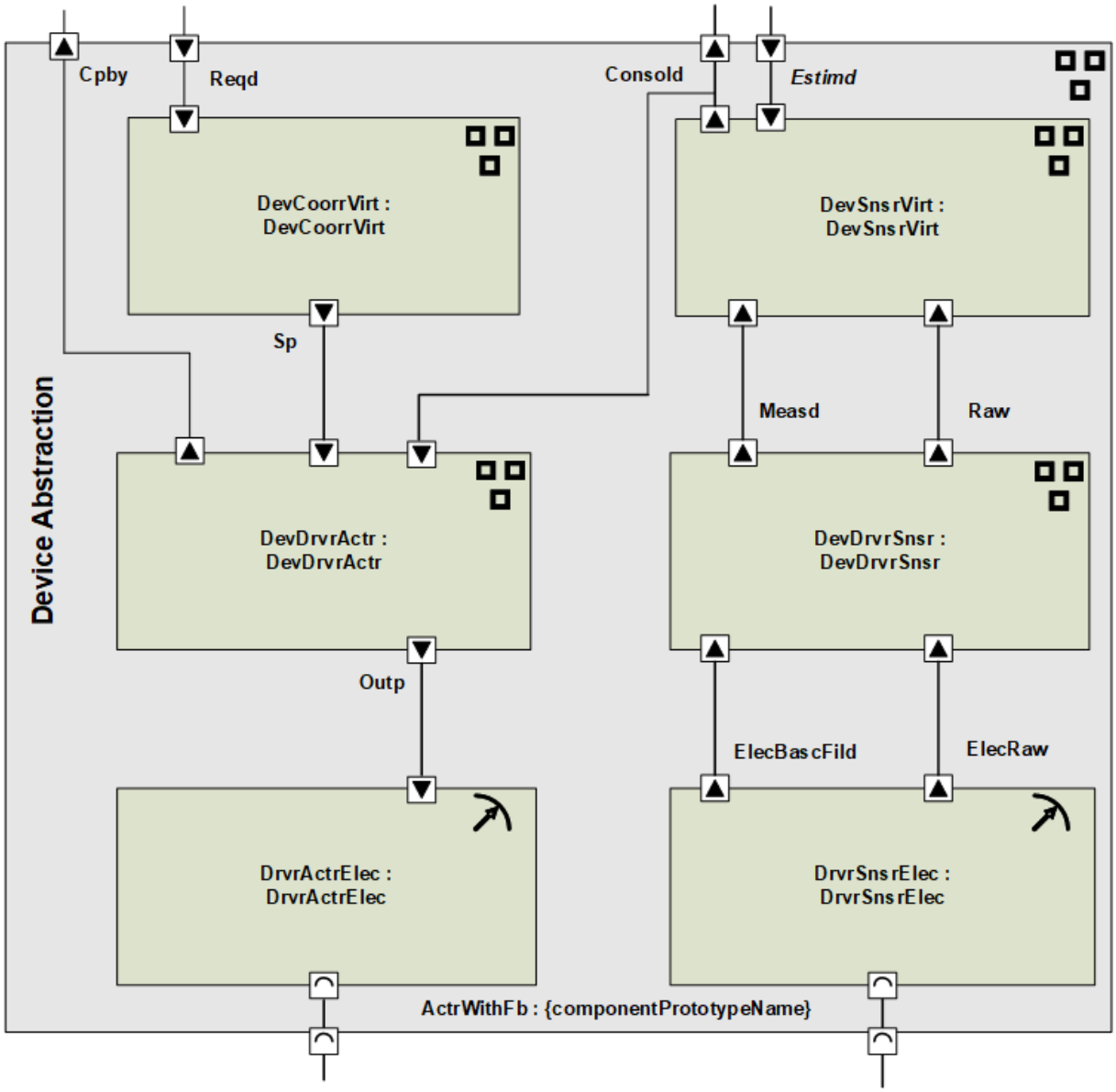


Figure 3.7: Device Abstraction for Actuator with Feedback

3.8 Usage of pattern

The SensorActuator-Pattern can be used in different ways, see figure 3.8.

S/A types defined in 3.7 are stored and available via AI tool. User can either take SwComponentTypes directly and create one or multiple instances, or he/she can take them as a blueprint with the need to extend and adjust them afterwards.
During single or multi instantiation SwComponentPrototypes with different names, but same content are generated. The blueprint mechanism offers the possibility to flexible change name, content and even rename ports. From point of creation, it has to be maintained separately.

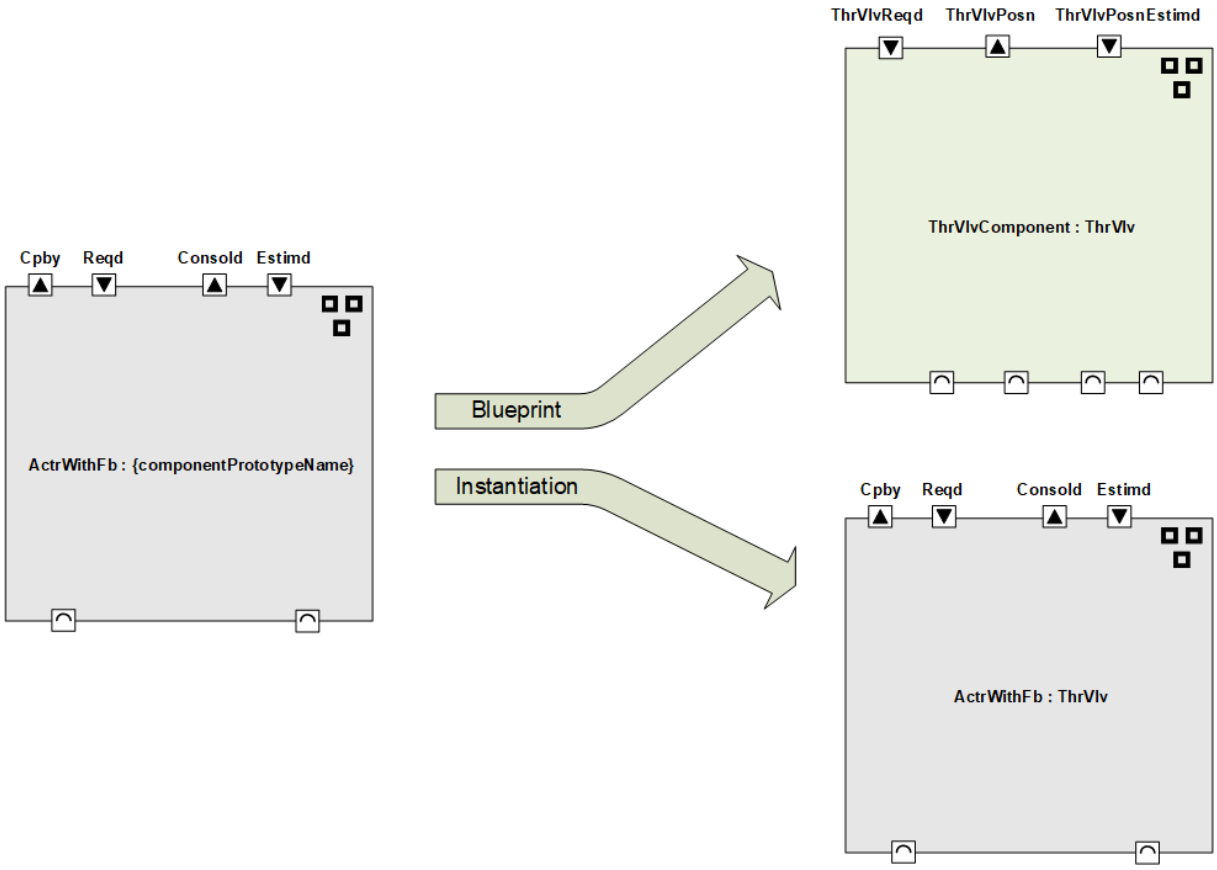


Figure 3.8: Usage options for SensorActuator-Pattern

3.9 Examples

3.9.1 Throttle Valve

Figure 3.9 shows an example device abstraction for a throttle valve.

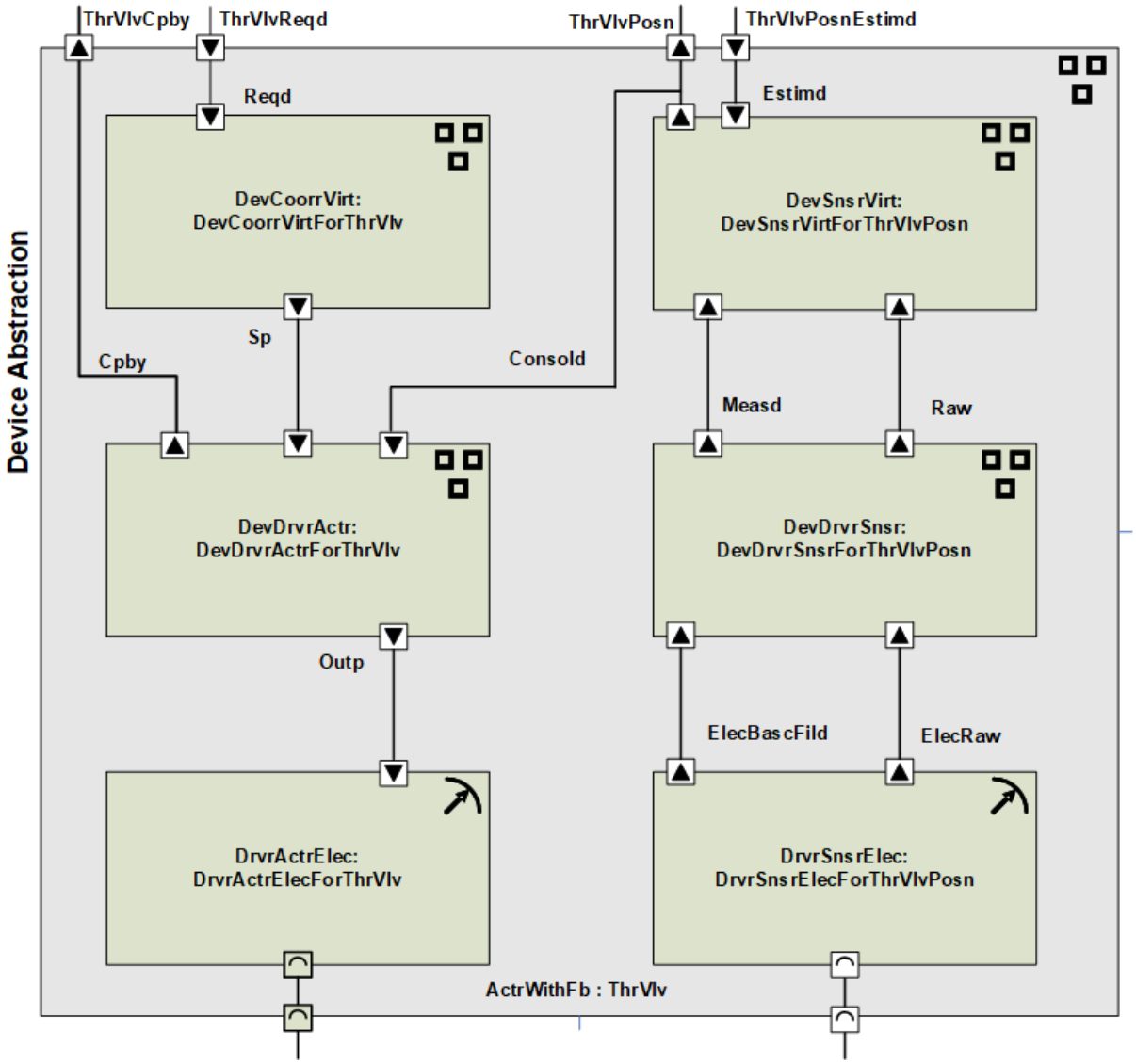


Figure 3.9: Device Abstraction for a Throttle Valve

3.9.2 Turbo Charger

In Figure 3.10 an example of a closed looped controlled device with position feedback — a turbo charger — is shown.

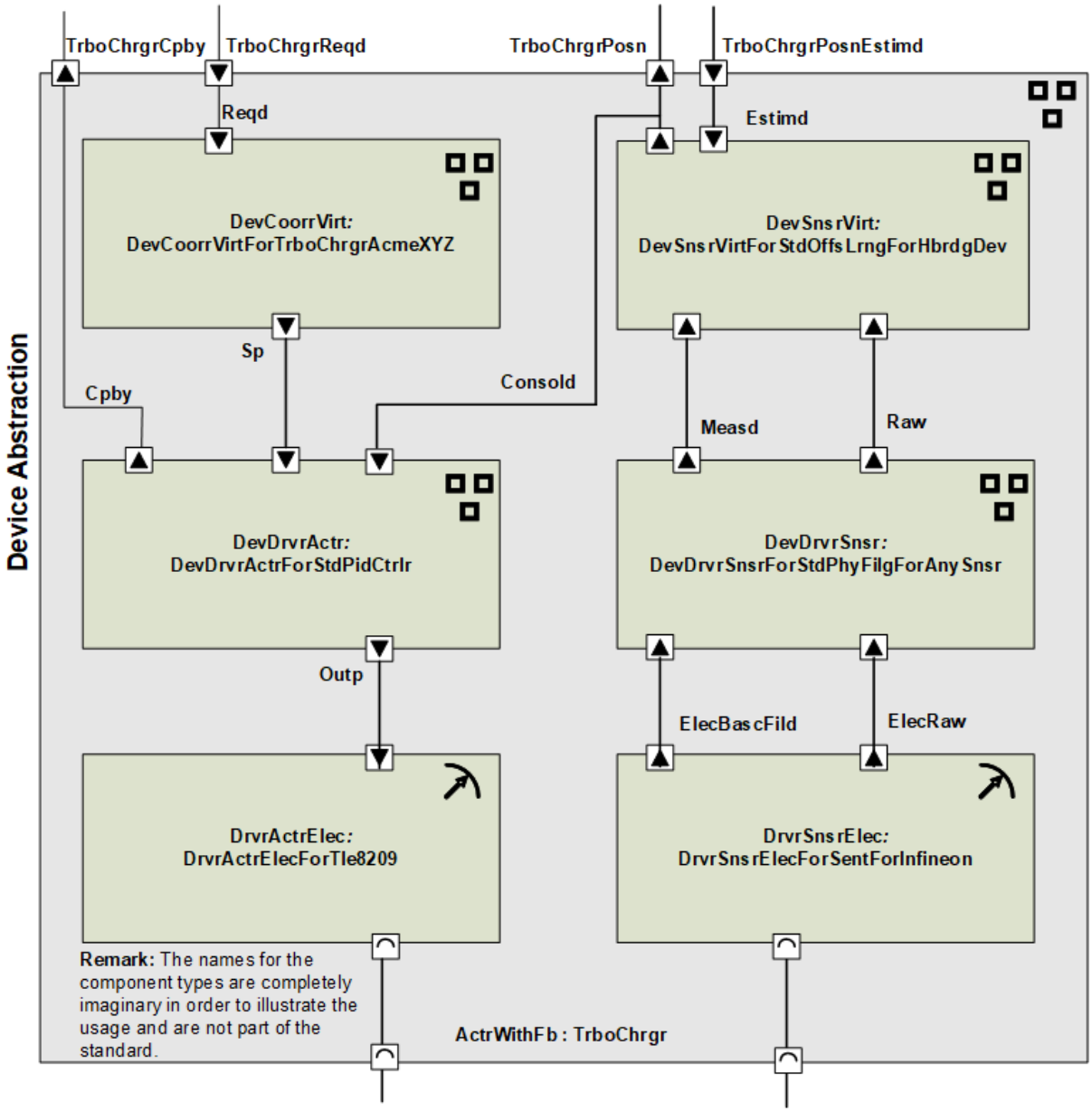


Figure 3.10: Device Abstraction for a Turbo Charger

Hint: In most cases it is not recommended to use company names in model names (like "AcmeXYZ" used in the Figures). Company names etc. are only used in the examples to show the difference between type and prototype and what is the reason for the difference. For general rules and recommendations how to deal with variants in models, as for example expressed by the company names in the examples, please refer to the modeling guides and templates.

3.9.3 Turbo Charger with Stages and Banks

In Figure 3.11 a project system configuration for turbo charger with several stages and banks is shown.

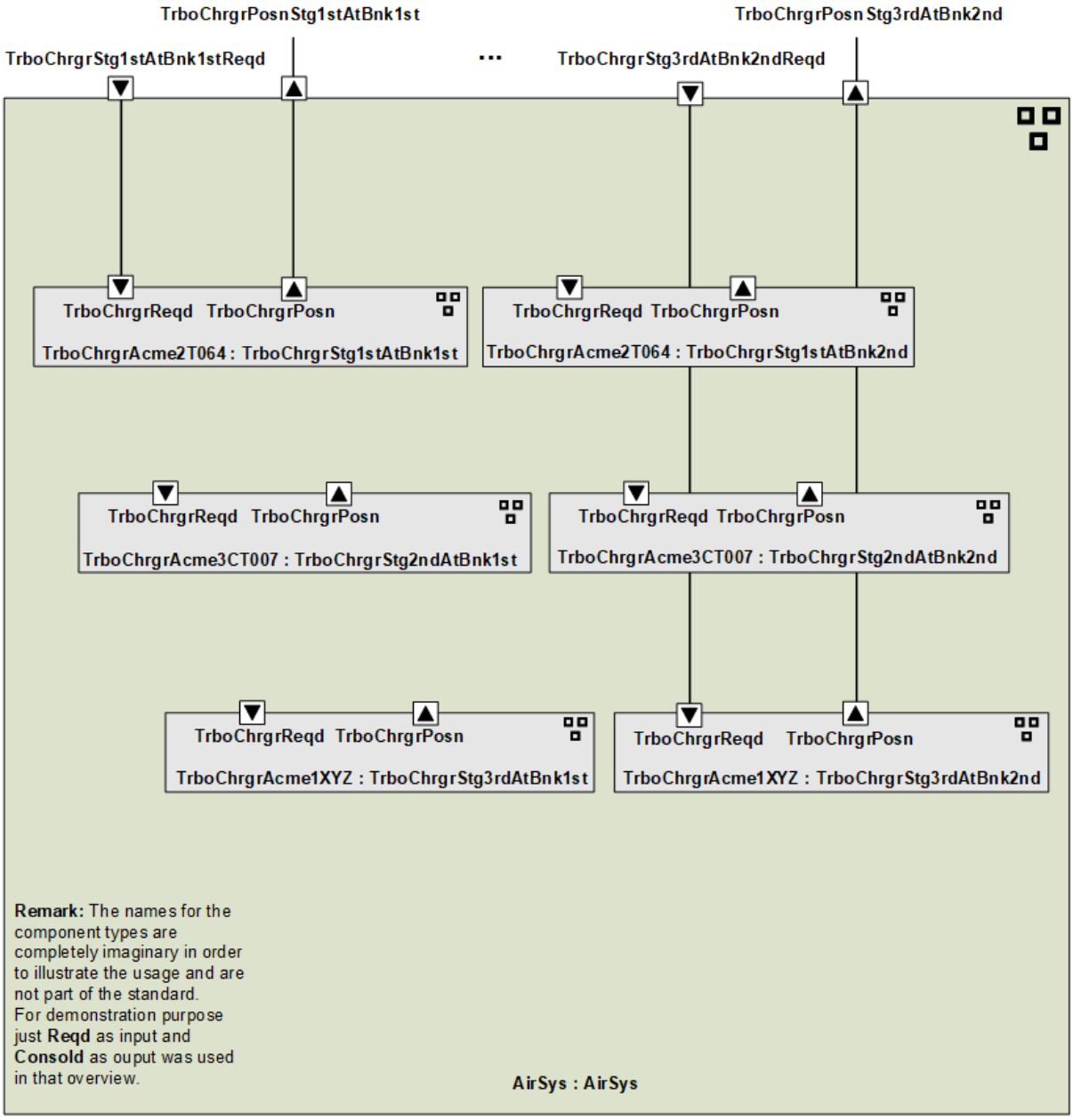


Figure 3.11: Device Abstraction for a Turbo Charger with Banks and Stages

3.9.4 Sensor for Environment Temperature

In Figure 3.12 a typical sensor for environment temperature is shown.

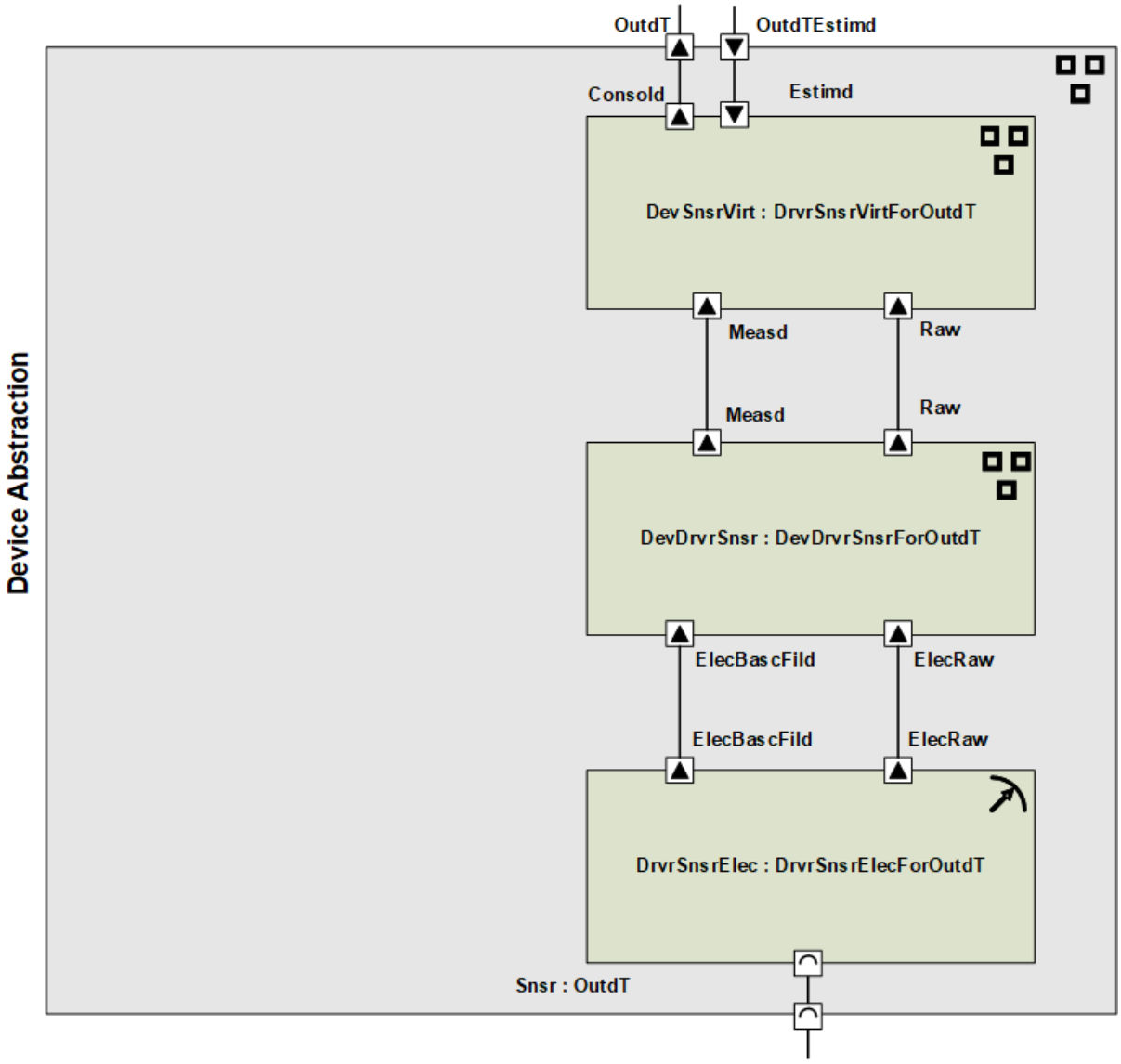


Figure 3.12: Device Abstraction for a Sensor measuring the Environment Temperature

3.9.5 Standard Sensor

In Figure 3.13 a design pattern of blueprint components for a standard sensor is shown.

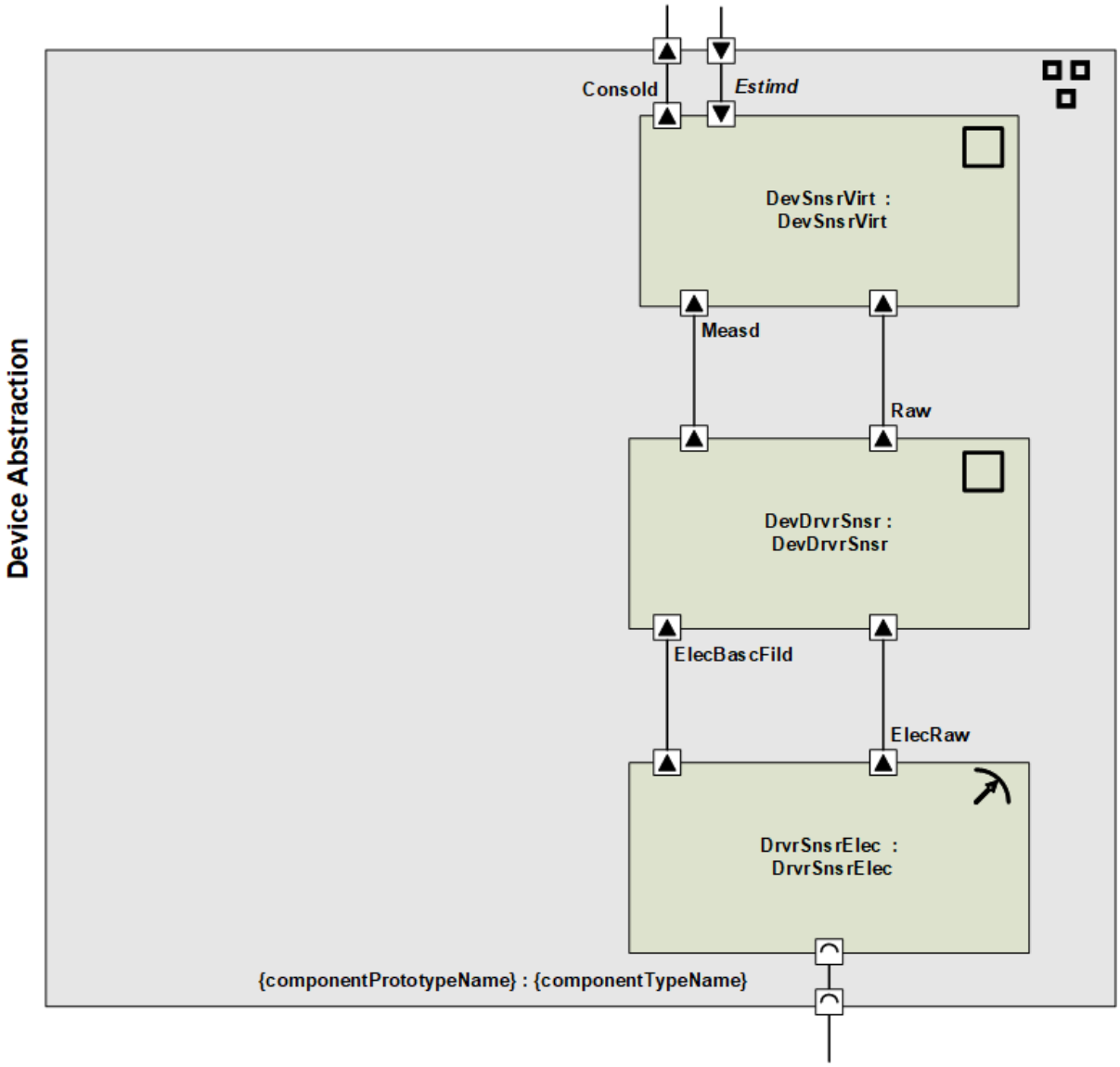


Figure 3.13: Device Abstraction for Standard Sensor

3.9.6 Standard Sensor for Environment Temperature

In Figure 3.14 a standard sensor for environment temperature is shown.

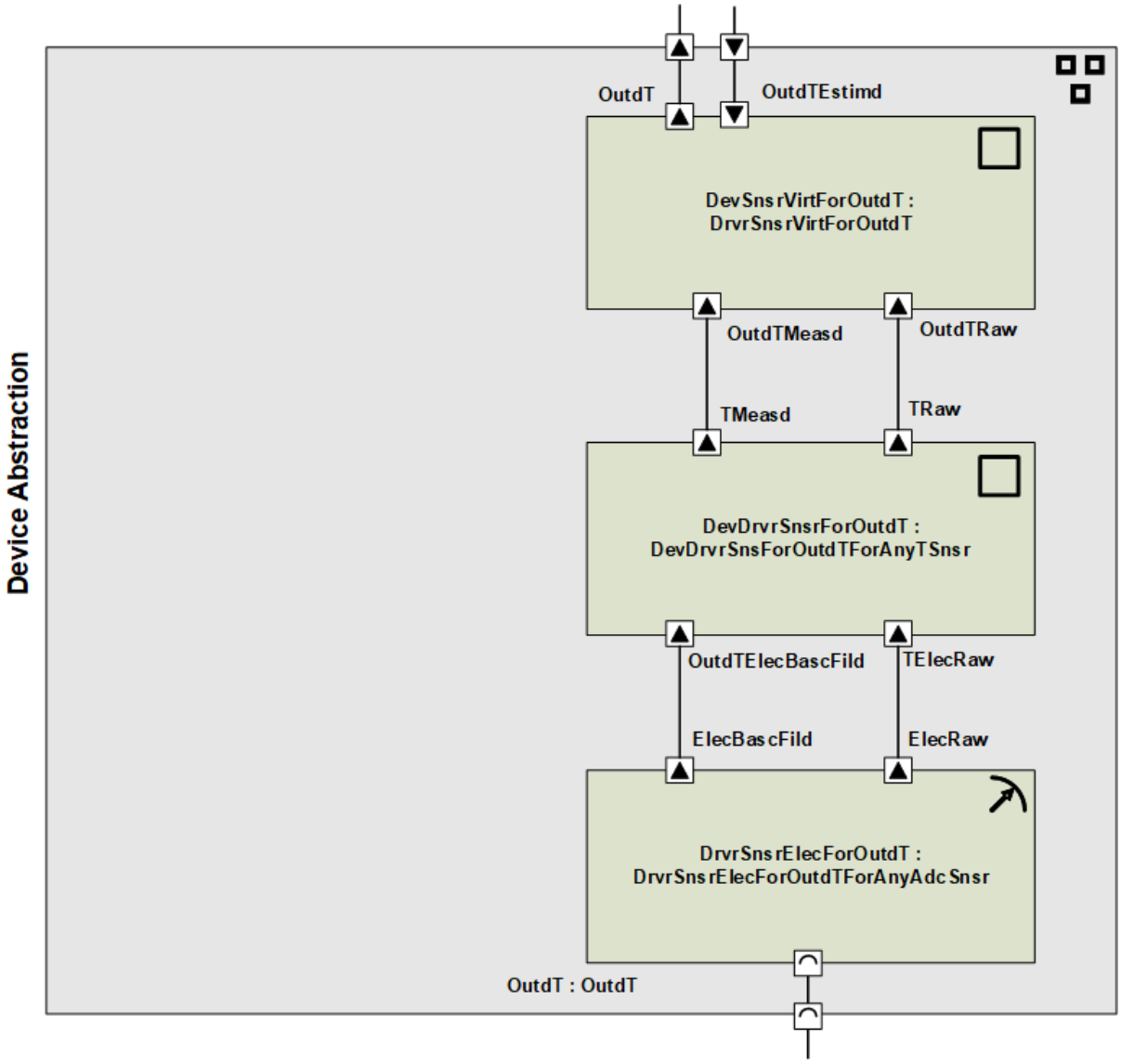


Figure 3.14: Device Abstraction for a Sensor measuring the Environment Temperature

3.9.7 Distributing Device Abstraction

In Figure 3.16 the ECU view derived from the VFB view of a temperature sensor as shown in Figure 3.15 is shown. Finally it is shown that it is possible to also deploy the different SW-C to different ECUs. Of course timing constraints have to be considered before distributing components to different ECUs.

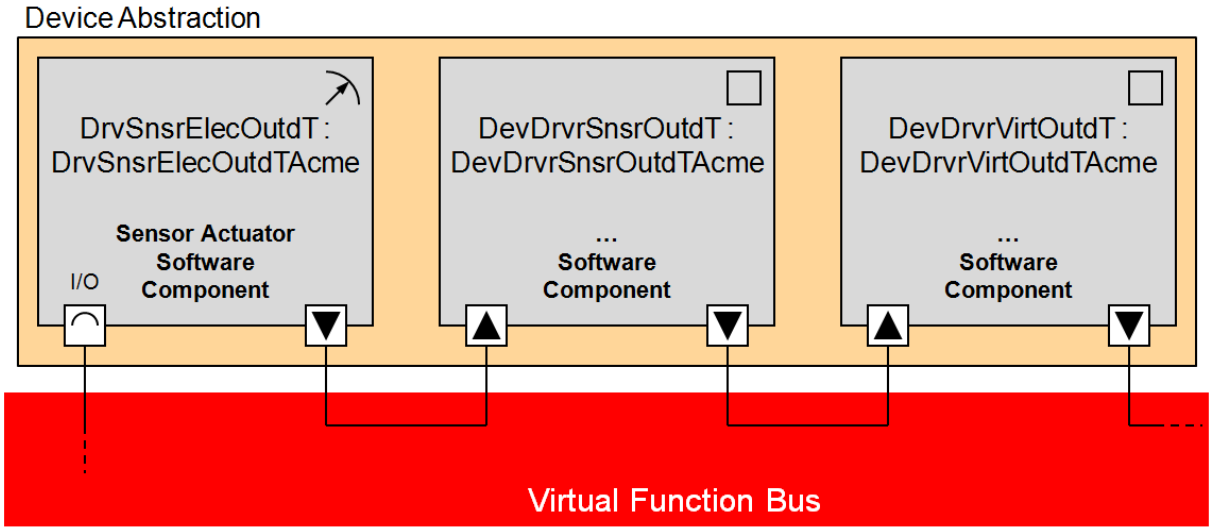


Figure 3.15: VFB View of Temperature Sensor Example

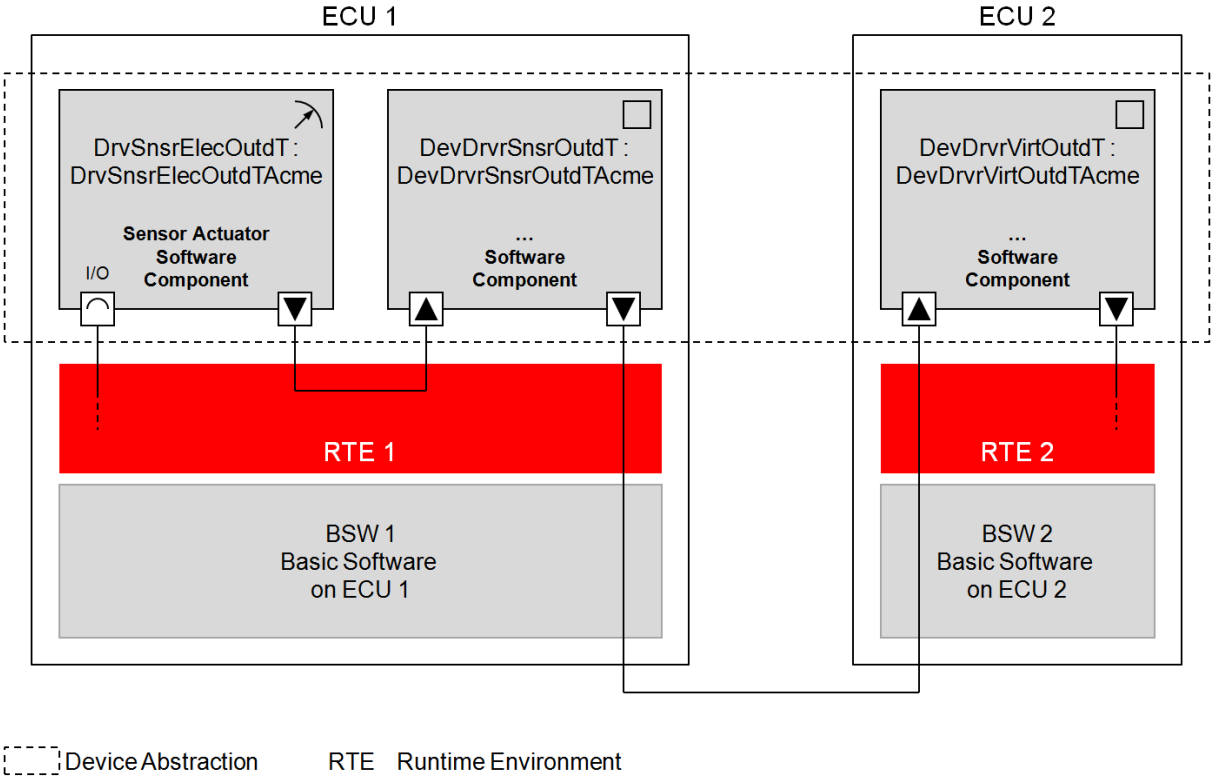


Figure 3.16: ECU Views after Distribution of SW-Cs of Temperature Sensor to two ECUs

3.10 Sample Code and Model

In Listing 3.7 a blueprint for the components used in the Sensor/Actuator pattern is provided. The blueprint code is not complete but just gives an idea how it is realized. The composition component is not shown.

Please note that the AUTOSAR meta model requests that a sensor actuator component type references a corresponding sensor or actuator, resp., using a HwDescriptionEntity, [11]. In this case a HwElement is needed to be used. Since there is a standardized HwCategory for sensors and actuators also a HwType is defined that is referenced by the HwElement.

```
<AR-PACKAGE>
  <SHORT-NAME>SwComponentTypes_Blueprint</SHORT-NAME>
  <CATEGORY>BLUEPRINT</CATEGORY>
  <REFERENCE-BASES>
    <REFERENCE-BASE>
      <SHORT-LABEL>HwDescriptionEntitys</SHORT-LABEL>
      <PACKAGE-REF DEST="AR-PACKAGE">/HwElements_Blueprint</PACKAGE-REF>
    </REFERENCE-BASE>
    <REFERENCE-BASE>
      <SHORT-LABEL>PortInterfaces_Blueprint</SHORT-LABEL>
      <PACKAGE-REF DEST="AR-PACKAGE">/AUTOSAR/AISpecification/
          PortInterfaces_Blueprint</PACKAGE-REF>
    </REFERENCE-BASE>
<REFERENCE-BASE>
        <SHORT-LABEL>SwComponentTypes_Example</SHORT-LABEL>
        <PACKAGE-REF DEST="AR-PACKAGE">/AUTOSAR/AISpecification/
          SwComponentTypes_Example</PACKAGE-REF>
    </REFERENCE-BASE>
<REFERENCE-BASE>
  <SHORT-LABEL>PortInterfaces_Example</SHORT-LABEL>
        <PACKAGE-REF BASE="SwComponentTypes_Example" DEST="AR-PACKAGE">
          SwDesignPatterns/SnsrActrAbstraction/PortInterfaces_Example</
          PACKAGE-REF>
    </REFERENCE-BASE>
  </REFERENCE-BASES>
  <ELEMENTS>
    <SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
      <SHORT-NAME NAME-PATTERN="{anyName}DrvrSnsrElec{anyNamePart}">
        DrvrSnsrElec</SHORT-NAME>
      <LONG-NAME>
        <L-4 L="EN">Driver for Electrical Signals of Sensor</L-4>
      </LONG-NAME>
      <INTRODUCTION><!-- optional: add documentation -->
      </INTRODUCTION>
      <PORTS>
        <P-PORT-PROTOTYPE>
          <SHORT-NAME NAME-PATTERN="{anyName}ElecRaw{anyNamePart}">ElecRaw
            </SHORT-NAME>
          <LONG-NAME>
            <L-4 L="EN">Electrical Raw Value</L-4>
          </LONG-NAME>
          <PROVIDED-INTERFACE-TREF DEST="SENDER-RECEIVER-INTERFACE" BASE="
            PortInterfaces_Example">ElecRaw1</PROVIDED-INTERFACE-TREF>
```

```
        </P-PORT-PROTOTYPE>
        <P-PORT-PROTOTYPE>
          <SHORT-NAME NAME-PATTERN="{anyName}ElecBascFild{anyNamePart}">
              ElecBascFild</SHORT-NAME>
          <LONG-NAME>
            <L-4 L="EN">Electrical Basic Filtered Value</L-4>
          </LONG-NAME>
          <PROVIDED-INTERFACE-TREF DEST="SENDER-RECEIVER-INTERFACE" BASE="
              PortInterfaces_Example">ElecBascFild1</PROVIDED-INTERFACE-
              TREF>
        </P-PORT-PROTOTYPE>
      </PORTS>
      <!-- add correct reference to sensor actuator type -->
      <SENSOR-ACTUATOR-REF DEST="HW-DESCRIPTION-ENTITY" BASE="
          HwDescriptionEntitys">mySensorActuatorElement</SENSOR-ACTUATOR-
          REF>
    </SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
    <APPLICATION-SW-COMPONENT-TYPE>
      <SHORT-NAME NAME-PATTERN="DevDrvrSnsr{anyNamePart}">DevDrvrSnsr</
          SHORT-NAME>
      <LONG-NAME>
        <L-4 L="EN">Device Driver for Sensor</L-4>
      </LONG-NAME>
      <!-- Ports to be added -->
    </APPLICATION-SW-COMPONENT-TYPE>
    <APPLICATION-SW-COMPONENT-TYPE>
      <SHORT-NAME NAME-PATTERN="DevSnsrVirt{anyNamePart}">DevSnsrVirt</
          SHORT-NAME>
      <LONG-NAME>
        <L-4 L="EN">Virtual Device Driver for Sensor</L-4>
      </LONG-NAME>
      <!-- Ports to be added -->
    </APPLICATION-SW-COMPONENT-TYPE>
  </ELEMENTS>
</AR-PACKAGE>
<!-- AR-PACKAGE: HwTypes_Blueprint -->
<AR-PACKAGE>
  <SHORT-NAME>HwTypes_Blueprint</SHORT-NAME>
  <CATEGORY>BLUEPRINT</CATEGORY>
  <ELEMENTS>
    <HW-TYPE>
      <SHORT-NAME NAME-PATTERN="{anyName}">SensorActuatorType</SHORT-NAME>
      <HW-CATEGORY-REFS>
        <HW-CATEGORY-REF DEST="HW-CATEGORY" BASE="HwCategorys">/
            HwCategorys_Blueprint/SensorActuator</HW-CATEGORY-REF>
      </HW-CATEGORY-REFS>
    </HW-TYPE>
  </ELEMENTS>
</AR-PACKAGE>
<!-- AR-PACKAGE: HwElements_Blueprint -->
<AR-PACKAGE>
  <SHORT-NAME>HwElements_Blueprint</SHORT-NAME>
  <CATEGORY>BLUEPRINT</CATEGORY>
  <ELEMENTS>
    <HW-ELEMENT>
```

```
      <SHORT-NAME NAME-PATTERN="{anyName}">mySensorActuatorElement</SHORT-
          NAME>
      <HW-TYPE-REF DEST="HW-TYPE" BASE="HwTypes">/HwTypes_Blueprint/
          SensorActuatorType</HW-TYPE-REF>
    </HW-ELEMENT>
  </ELEMENTS>
</AR-PACKAGE>
```

Listing 3.7: Sensor/Actuator Pattern

The HwCategorys should be provided centrally because they are standardized. Definition of HwCategory "SensorActuator" is shown in Listing 3.8.

```
<!-- AR-PACKAGE: HwCategorys_Blueprint -->
<AR-PACKAGE>
  <SHORT-NAME>HwCategorys_Blueprint</SHORT-NAME>
  <CATEGORY>BLUEPRINT</CATEGORY>
  <ELEMENTS>
    <HW-CATEGORY>
      <SHORT-NAME NAME-PATTERN="blueprintName">SensorActuator</SHORT-NAME>
    </HW-CATEGORY>
    <!--  add additional HW categories  -->
  </ELEMENTS>
</AR-PACKAGE>
```

Listing 3.8: HW Categories as used in Sensor/Actuator Pattern

3.11 Typical location of some common function within the specified layers

This chapter is for detailed description of the distribution of features across the device abstraction layers. It provides some examples of some typical and common features and their recommended location within the specified layers of the S/A-Pattern. Scope for this chapter is to make interface standardization easier.

3.11.1 Virtual Device Coordinator (DevCoorrVirt)

Virtual device is an abstraction of the physical representation of the actuator.

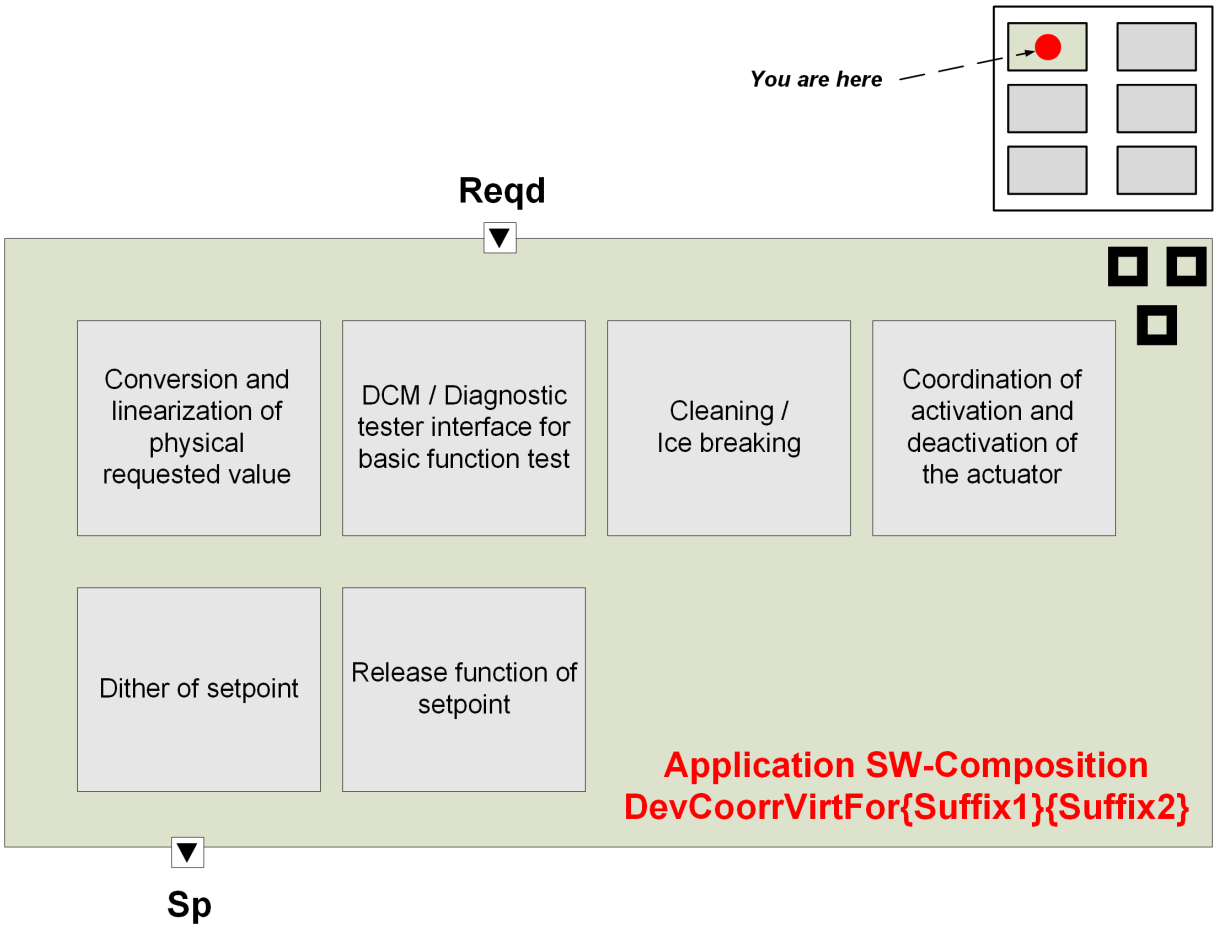


Figure 3.17: Typical functions in layer DevCoorrVirt

3.11.1.1 Conversion and linearization of physical requested value

Typically there is a delta between the mechanical endstops and the position where the physical effects are influenced due to the movement of the actuator. This gap could be compensated via offset compensation algorithm of the position sensor or via linearization of the requested setpoint value. The transfer function is used to compensate the actuator HW design/physics.

3.11.1.2 DCM service / Diagnostic tester interface for basic function test

The DCM service interface is typically used as a tester interface and can overwrite the requested value to perform a basic function test of the actuator.

3.11.1.3 Cleaning / Ice breaking

Overwrite/Ignore the requested value, in order to prepare the actuator for proper actuation. The function switches between two different setpoint values for a specific time to either

1. condition the actuator for offset learning
2. clean particles/compounds from actuator
3. break up from ice

3.11.1.4 Dither of setpoint

Continous overlayed/modulated signal on setpoint value to overcome static friction of actuator.

3.11.1.5 Release function of setpoint

The release function is manipulating the requested setpoint value. This could be needed in case of a blocked actuator, i.e. the actuator got stuck at its position.

3.11.1.6 Coordination of activation and deactivation of the actuator

Activation: The actuator shall be activated as soon as actuation is requested.
Deactivation: To ensure safe operation, the actuator shall be shut off under certain conditions (incl. monitoring e.g. open hood) and shall be shut off to fail safe before voltage supply is switched off.

3.11.2 Actuator Device Driver (DevDrvrActr)

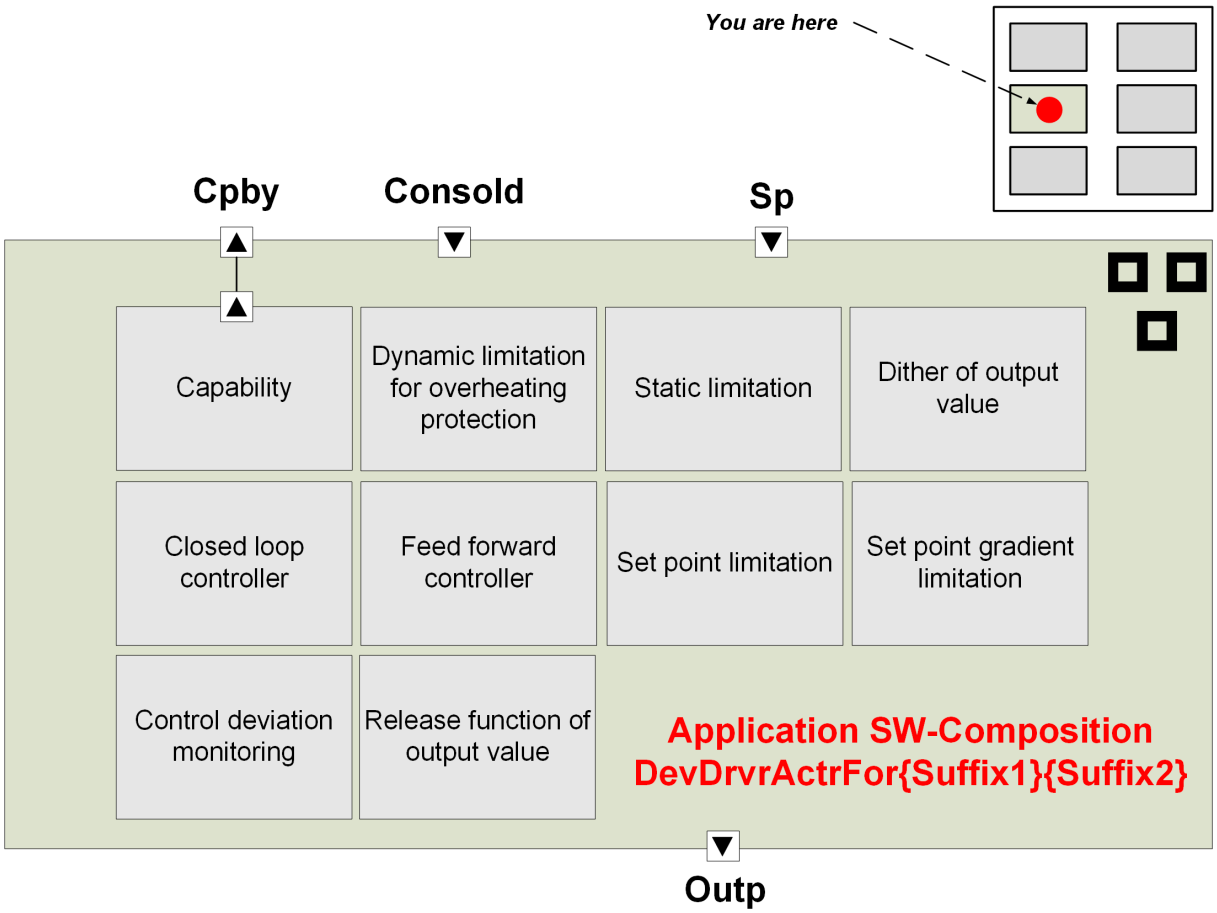


Figure 3.18: Typical functions in layer DevDrvrActr

3.11.2.1 Dither of output value

Continous overlayed/modulated signal on output value to overcome static friction of actuator.

3.11.2.2 Release function of output value

The release function is manipulating the output value. This could be needed in case of a blocked actuator, i.e. the actuator got stuck at its position.

3.11.2.3 Limitation

3.11.2.3.1 Static limitation

The output value is limited to protect the actuator from any mechanical or thermal damage at a static position. It is a static limitation of the output value.
Example: Limitation of dutycycle at the mechanical endstops, e.g. to avoid overheating.

3.11.2.3.2 Dynamic limitation for overheating protection

Effective current monitoring + housing/motor temperature monitoring is used as overheating protection. To protect the actuator of overheating, the energy input to the actuator or the temperature inside the actuator is observed. It is a dynamic limitation of the output value.
Hint: The temperature information could also come as a consolidated value from an abstracted sensor SW component.

3.11.2.4 Feed forward controller

The Feed Forward Controller compensatex the influcence of the known disturbances in the controlled system. It calculates the pre-controlled output value.

3.11.2.5 Closed loop controller

The Closed Loop Controller uses feedback to control output of a dynamic system, i.e. the output value is adapted according to the consolidated value.

3.11.2.6 Set point limitation

Set point limitation given by plant used as closed loop controller input.

3.11.2.7 Set point gradient limitation

Protection of the actuator by limiting the set point gradient, e.g. in position close to the endstops.

3.11.2.8 Control deviation monitoring

Monitoring of the permanent deviation between setpoint and consolidated value.

3.11.2.9 Capability

Providing a Capability is a way of summarizing all active limitations on an actuator. The Capability is related to the requested set point, providing the dynamic boundaries of possible usage.
For example, an electric machine actuator SW composition will report its capability to the coordinator functionality in the application software. If the capability is reduced, the coordinator functionality in the application software may use this capability information to redistribute the requested set points differently between the actuators of the system to obtain the overall system control objective.

Generic Signal Name	Long Name Pattern of Concrete Sensor/Actuator Signal (EN)	Generic Long Name of Signal (EN)	AUTOSAR Definition
`Cpby`	Capability {anyLongNamePart}	Capability	Provides the dynamic instant capability typically based on output limitation but could also contain the limitation on rate of change of the consolidated value. It is expressed as percentage.

Table 3.7: Signal Names and Semantics of function Capability

This following section presents examples of capability.
The capability can be described as the temporary dynamic bounds of actuation. These bounds could depend on current working point of operation or some consolidated value. The capability is provided as percentage of maximum defined actuator limitations.
For example, if the capability is provided as neutral (see figure 3.19), the capability is set to 100%. Consequently, neutral capability does not reflect the current effectiveness of the actuator.



Figure 3.19: Example for providing neutral Capability information

In another example (see figure 3.20), the capability is provided as a function of the set point and output limitations. The dynamic set point and output limitations may then also be a function of the consolidated value.

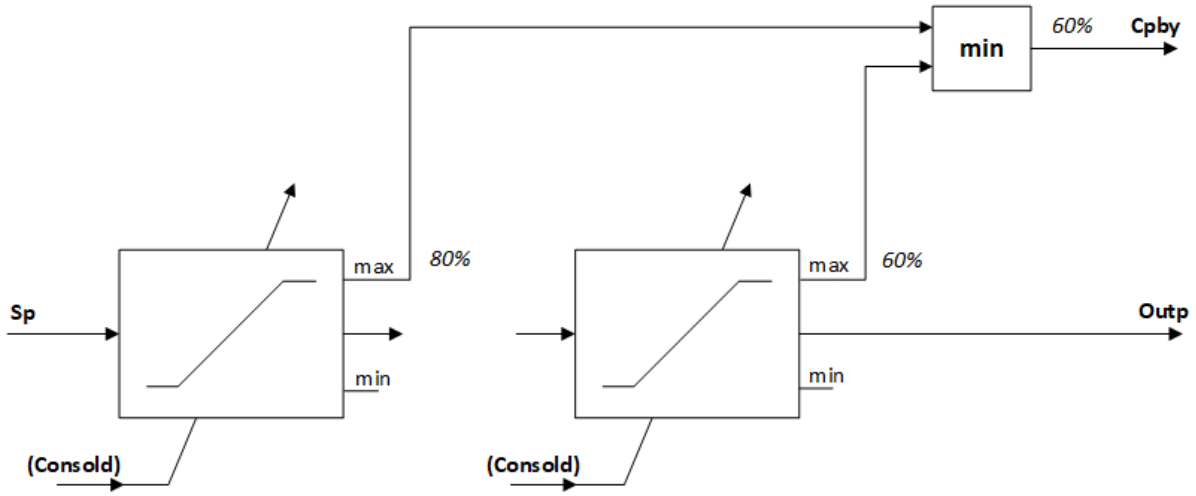


Figure 3.20: Example for simple Capability calculation

3.11.3 Electrical Actuator Driver (DrvrActrElec)

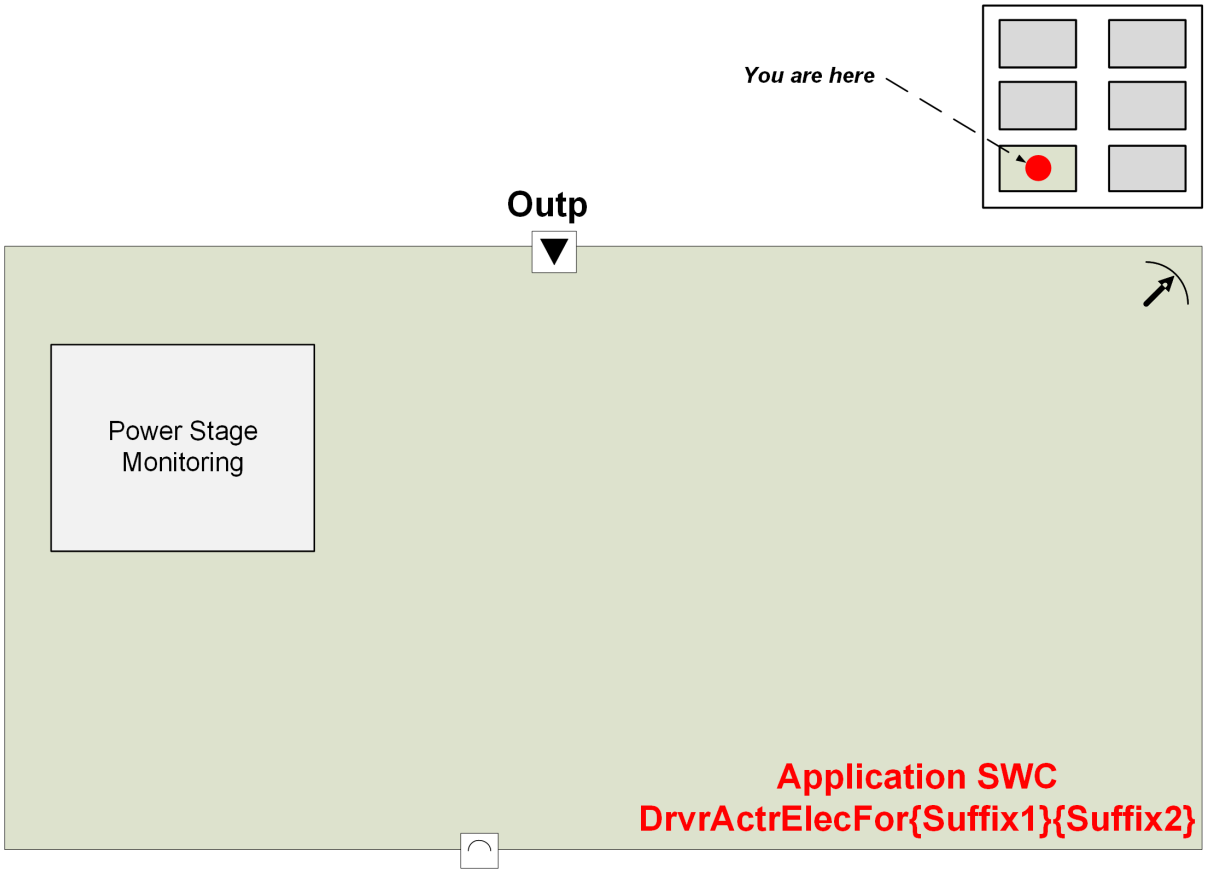


Figure 3.21: Typical functions in layer DrvrActrElec

3.11.3.1 Power stage monitoring

An ECU might contain various power stages for driving different electrical loads. Common electrical faults at power stages are Short Circuit to Battery (SCB), Short Circuit to Ground (SCG), and Open Load (OL). These faults can occur during either on-state or off-state of the power stage output.

3.11.4 Virtual Device Driver (DevSnsrVirt)

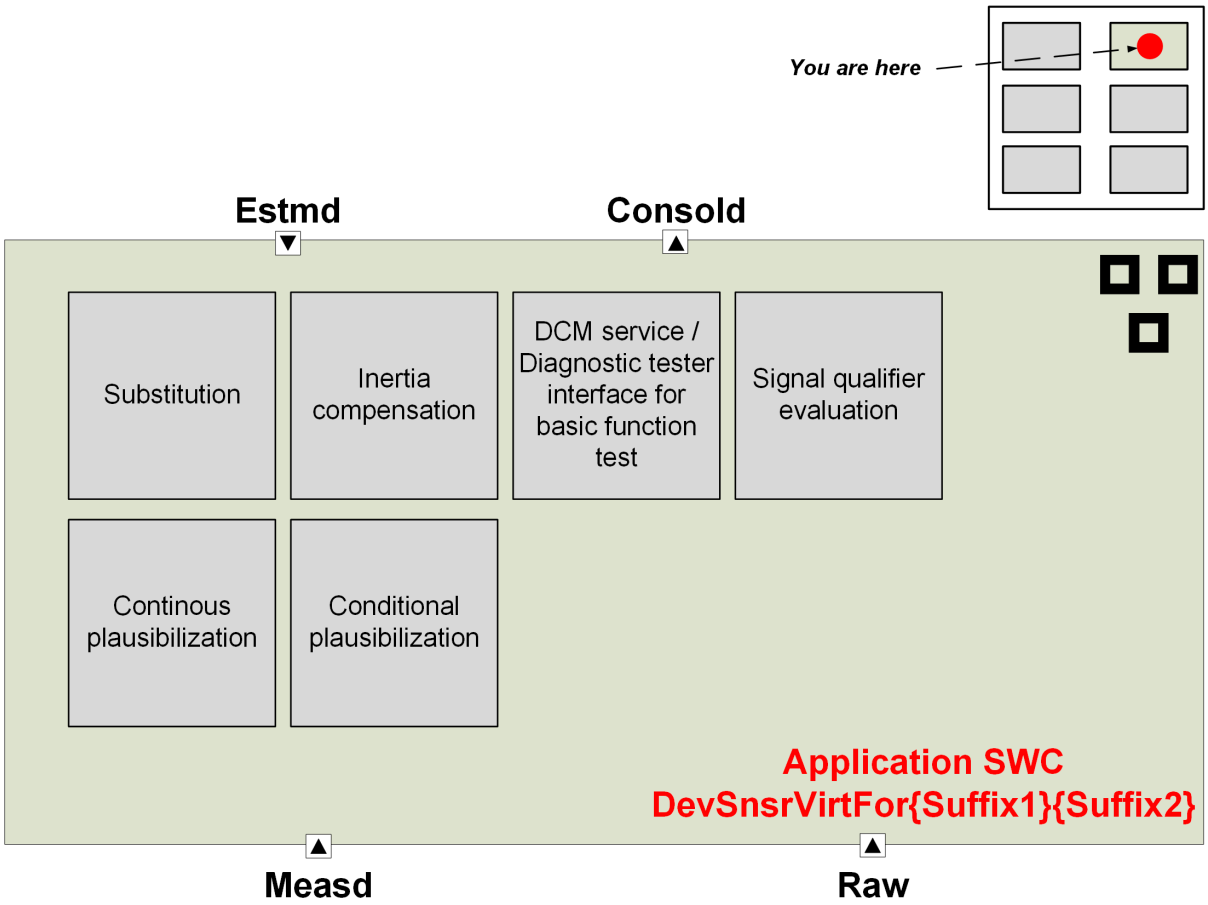


Figure 3.22: Typical functions in layer DevSnsrVirt

3.11.4.1 Substitution

The function switches between the measured and a replacement value. The replacement value could be the estimated value.
Example: The switching can happen based on:

1. Sensor diagnostic information

2. Sensor signal quality
3. Sensor availabilty

3.11.4.2 Inertia compensation

The function provides a predicted sensor value (forecast) to compensate the inertia of the sensor.
Examples: thermal inertia, mechanical inertia

3.11.4.3 Signal qualifier evaluation

The quality of the consolidated value is provided by that function. It is determined by checking consolidated value and all sensor related diagnosis information.

3.11.4.4 DCM service / Diagnostic tester interface for basic function test

The DCM service interface is typically used to overwrite and stimulate the consolidated sensor value.

3.11.4.5 Plausibilization

3.11.4.5.1 Continous plausibilization

The measured value is checked continously against another redundant sensor information. This redundant sensor information can be provided by any other sensor or by the estimated value.
Example: Offset diagnosis, in case difference (measured value vs. redundant value) exceeds certain threshold, e.g. tolerance threshold.

3.11.4.5.2 Conditional plausibilization

The measured value is checked at specific points in time (e.g. once in a driving cycle or at specific driving modes) against another redundant sensor information. This redundant sensor information can be provided by any other sensor or by the estimated value.
Hint: The conditional plausibilization can be used to compensate or just identify sensor individual tolerances.

3.11.5 Sensor Device Driver (DevDrvrSnsr)

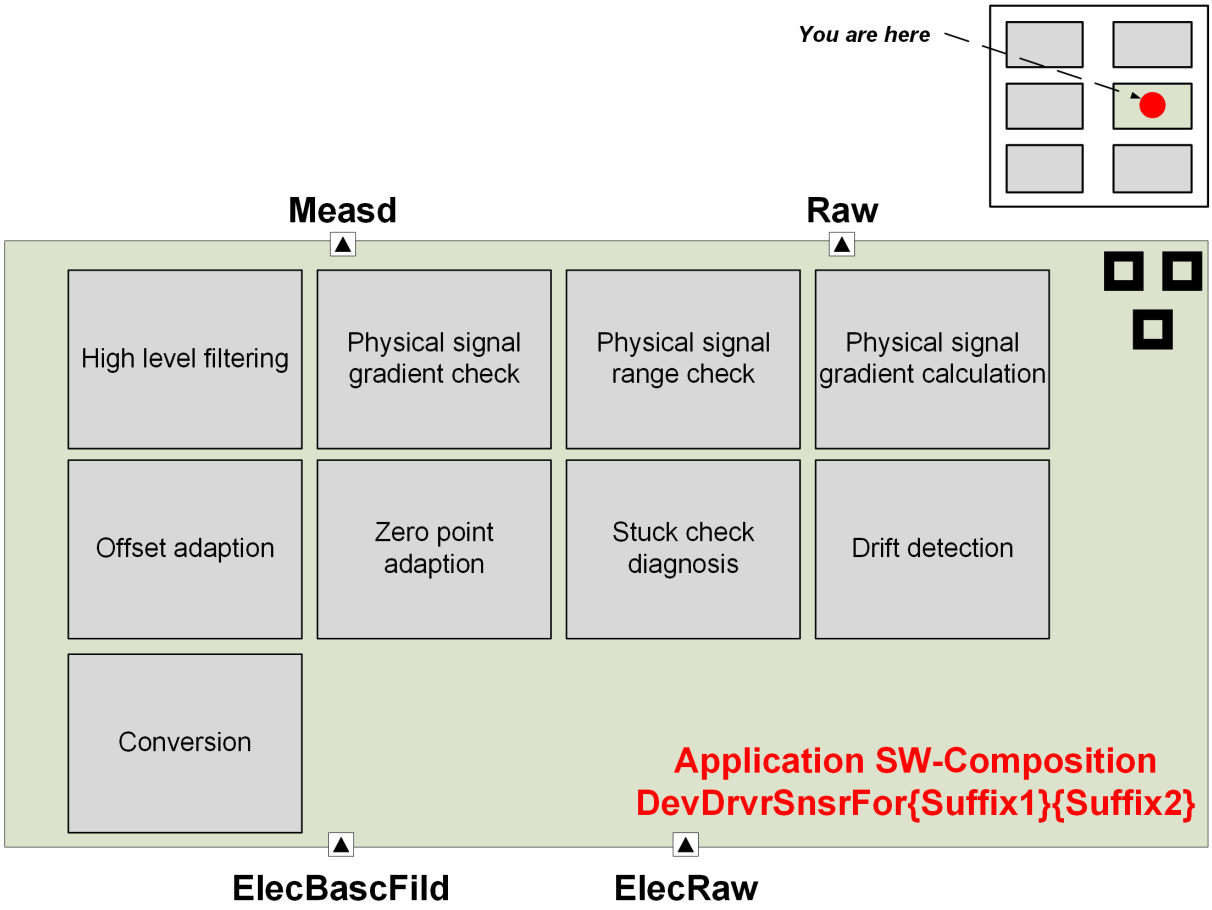


Figure 3.23: Typical functions in layer DevDrvrSnsr

3.11.5.1 High level filtering

This function block contains every kind of filter which might lead to a significant phase shift of the sensor value in order to provide a physical sensor value, fitting to requirements from user functions (regarding timing, accuracy).
Hint: Therefore a good trade-off between phase shift and accuracy has to be found.

3.11.5.2 Offset adaption

The result of conditional plausibilization can be used to do an offset adaption of measured value to compensate individual tolerances of the sensor. The determined offset information is used to adapt the sensor signal to show values closer to the actual physical signal.
Hint: The conditional plausibilization can be used to compensate or just identify sensor individual tolerances.

3.11.5.3 Zero point adaption

The zero point adaption is used to adjust the transfer function in the conversion to the physical zero point.
Hint: The adaption of this zero point is done within the conversion block.
Example 1: Sensors measuring relative values (differential pressure) shall show 0 if there is equalized pressure.
Example 2: The sensor value is adapted to the mechanical endstop position of an closed loop operated actuator.

3.11.5.4 Drift detection

Sensor values are monitored throughout the driving cycle and used to derive a sensor deviation compared to the first and last learned value.
Hint: Can be used for offset adaption, to improve sensor information or it can be used for diagnosis purpose only.

3.11.5.5 Conversion

The electrical signal is converted into physical representation by transfer function. In case of nonlinear signal, linearization will be part of transfer function as well.

3.11.5.6 Physical signal gradient calculation

In order to get information about the current dynamic of the sensed system, a gradient is calculated based on current and previous sensor information.

3.11.5.7 Physical signal gradient check

The gradient of the physical signal is checked against a maximum. For certain sensors a maximum gradient should not be exceeded. In case the sensor shows a higher gradient, it could be indicated as defect.

3.11.5.8 Stuck check diagnosis

Identify a "frozen" sensor information, in case the sensor signal does not change. A permanent "frozen" sensor information could be indicated as a defect.

3.11.5.9 Physical signal range check

Comparison of physical sensor signal against minimum and maximum thresholds for continuous diagnosis of physical limits.

3.11.6 Electrical Sensor Driver (DrvrSnsrElec)

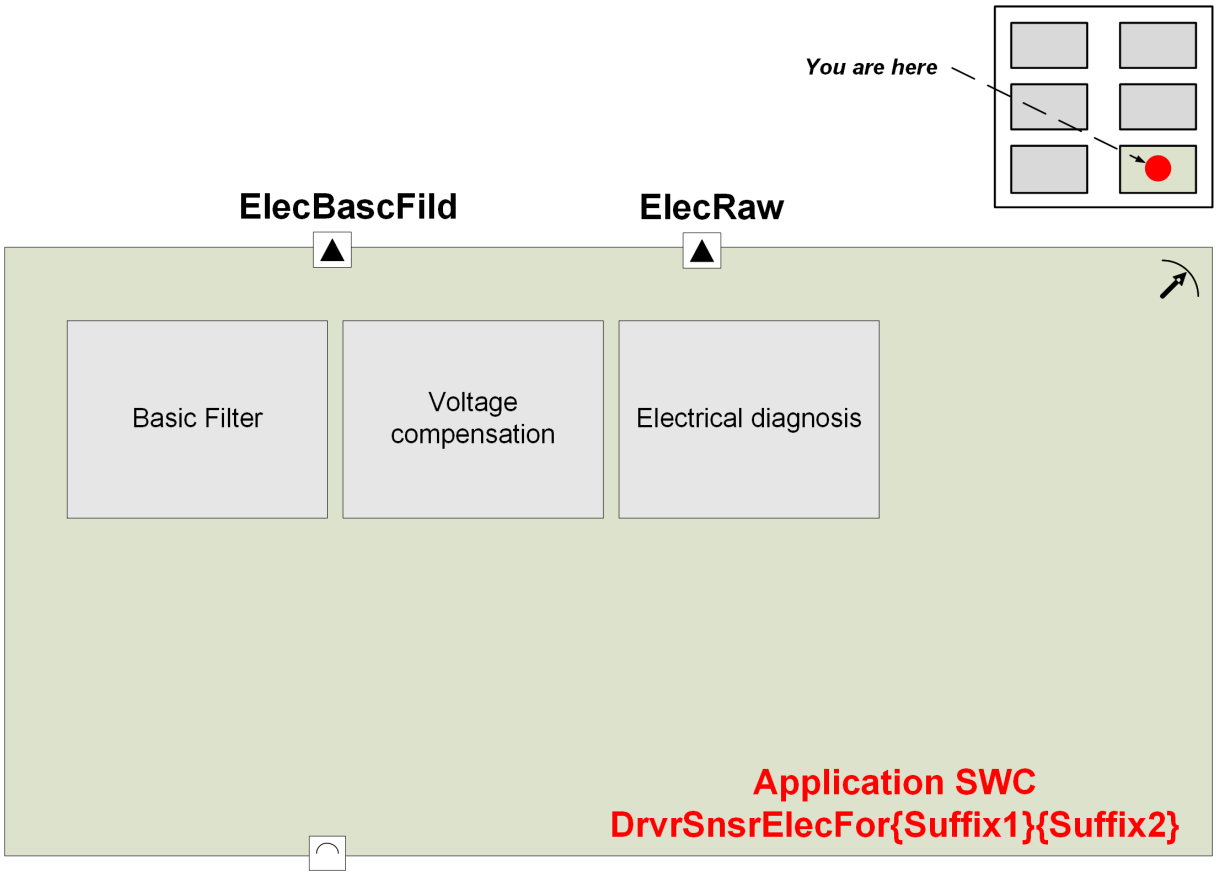


Figure 3.24: Typical functions in layer DrvrSnsrElec

3.11.6.1 Basic filter

A basic filter is needed to mitigate electric noise. The timing behavior shall not give any significant phase shift to signal.
Example: The definition of a significant phase shift is that it does not have any impact on the physical behaviour of the system. For signals influenced by the combustion the phase shift should not exceed the time given by a $360\mathrm{deg}$ camshaft rotation.
Hint: Possible filter types for this use case could be FIR (finite impulse response) filter or PLL (phase locked loop).
Reason: The DevDrvrSnsr transfers electrical value to physical value. In case the signal already has a phase shift, the timing within the upper layers cannot be compensated anymore.

3.11.6.2 Voltage compensation

Required for sensors with power supply from outside ECU. The seperate power supply creates a potential difference in reference voltage which needs to be compensated in SW.
Hint: This functionality can be realized in hardware alternatively.

3.11.6.3 Electrical diagnosis

It is needed to diagnose electrical faults on the sensor.
Examples: Short Circuit to Battery (SCB), Short Circuit to Ground (SCG), Open Circuit, Loose Contact.

3.12 Known Issues

Sensor abstraction of sensors with typical digital interfaces (e.g. SENT, FAS) or which are connected via bus (e.g. CAN, LIN) is part of this pattern as well. Description of required extensions is in progress.

3.13 FAQ

- Why is the estimated value in Example "Actuator without Feedback Loop (Setpoint Alternative)" not used?
  An estimated value does not exist for every sensor. So there is no need for it to be used. In this example, the consolidated value is calculated based on the setpoint.
- Is there a signal quality considered in the pattern?
  The topic "signal qualifier" is not yet considered. At the moment (R19-11) there is no activity known for standardizing such a signal quality.
- How are the names for the layers derived (e.g. DevCoorrVirt)? Can they be changed?
  The AUTOSAR abbreviations are given by strict rules [3]. Even the concatenation of the abbreviations is defined. The names should not be changed due to backward compatibility reasons.

3.14 Known Uses

None.

3.15 Related Patterns

Pattern	Description
Arbitration Pattern (see Chapter 4)	The sensor/actuator pattern is typically combined with the arbitration pattern to allow several set point requesters, several providers of consolidated values or several providers of estimated values. This is, arbitration is not done within the sensor/actuator pattern but outside the device abstraction.

Table 3.8: Related Patterns

3.16 Anti-Patterns One Should be Aware of

None.

3.17 Further Readings

More information could be found in [11] and [12].

4 Arbitration between several requesters or providers

Classification Design Pattern

4.1 Problem

Arbitration between several different providers or requesters.

4.2 Applicability

The number of requesters or providers, resp., has to be known at pre-compile time. The number of requesters or providers, resp., has to be known at implementation or generation time of the arbiter component.

This pattern can be applied in the context of Sensor/Actuator Design Pattern, e.g. for modeling several setpoint requesters, several providers of consolidated values or several providers of estimated values.

4.3 Solution

A new component for managing all requests from different requesters or providers, resp., is introduced. In Figure 4.1 the overall pattern for requesters is shown in case sender receiver interfaces are used. In Figure 4.2 the overall pattern for providers is shown in case sender receiver interfaces are used.

When using sender/receiver interfaces the arbitration component, also called ”arbiter”, needs to have unique names for the different requests or providers. This is realized by different request or provide ports, one per requester or provider, resp. The port interface or at least the application data type is typically the same for all of these requesters or providers, resp., and the resulting request or arbitrated value.

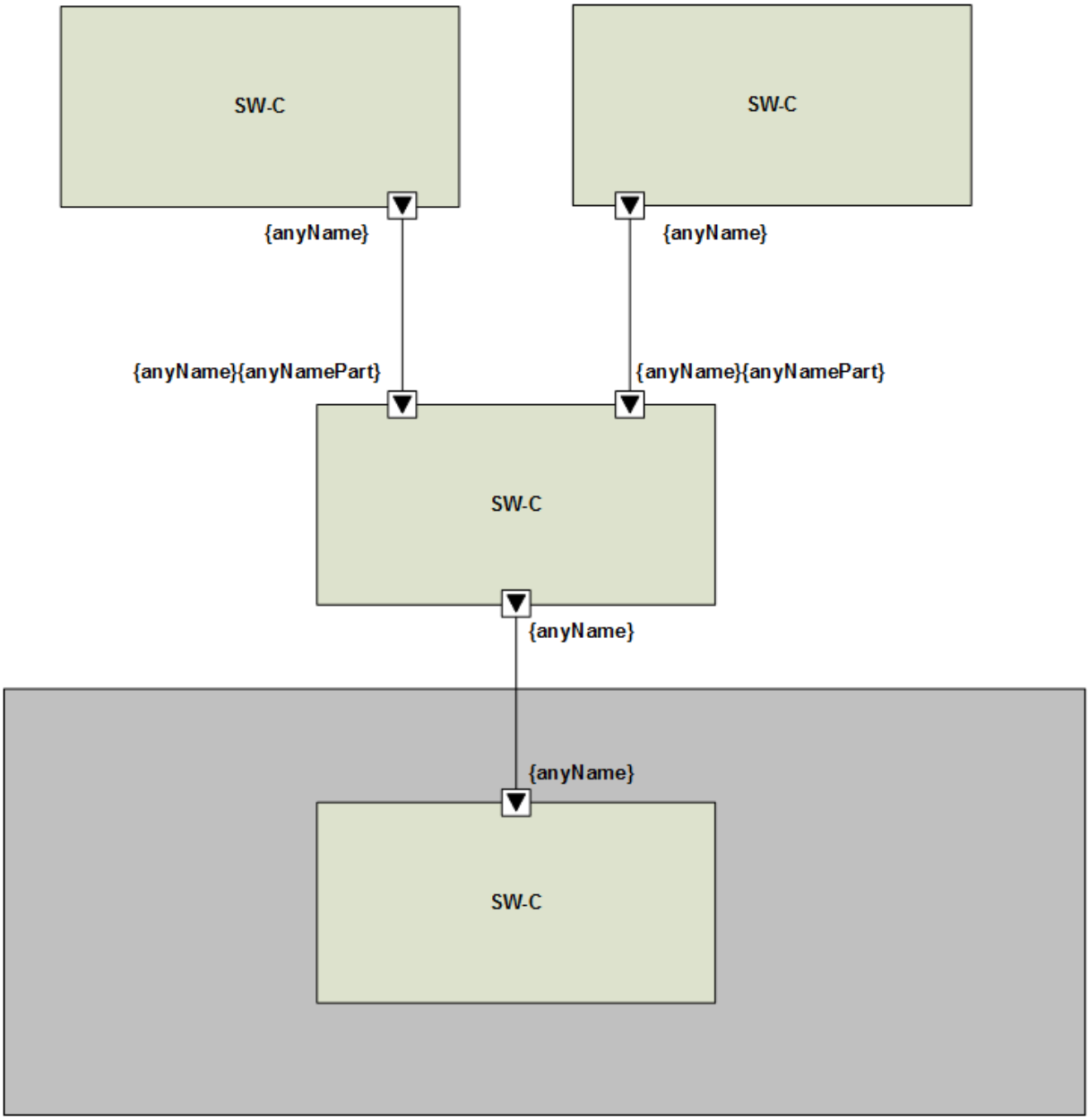


Figure 4.1: Pattern ”Arbitration between Several Requesters”

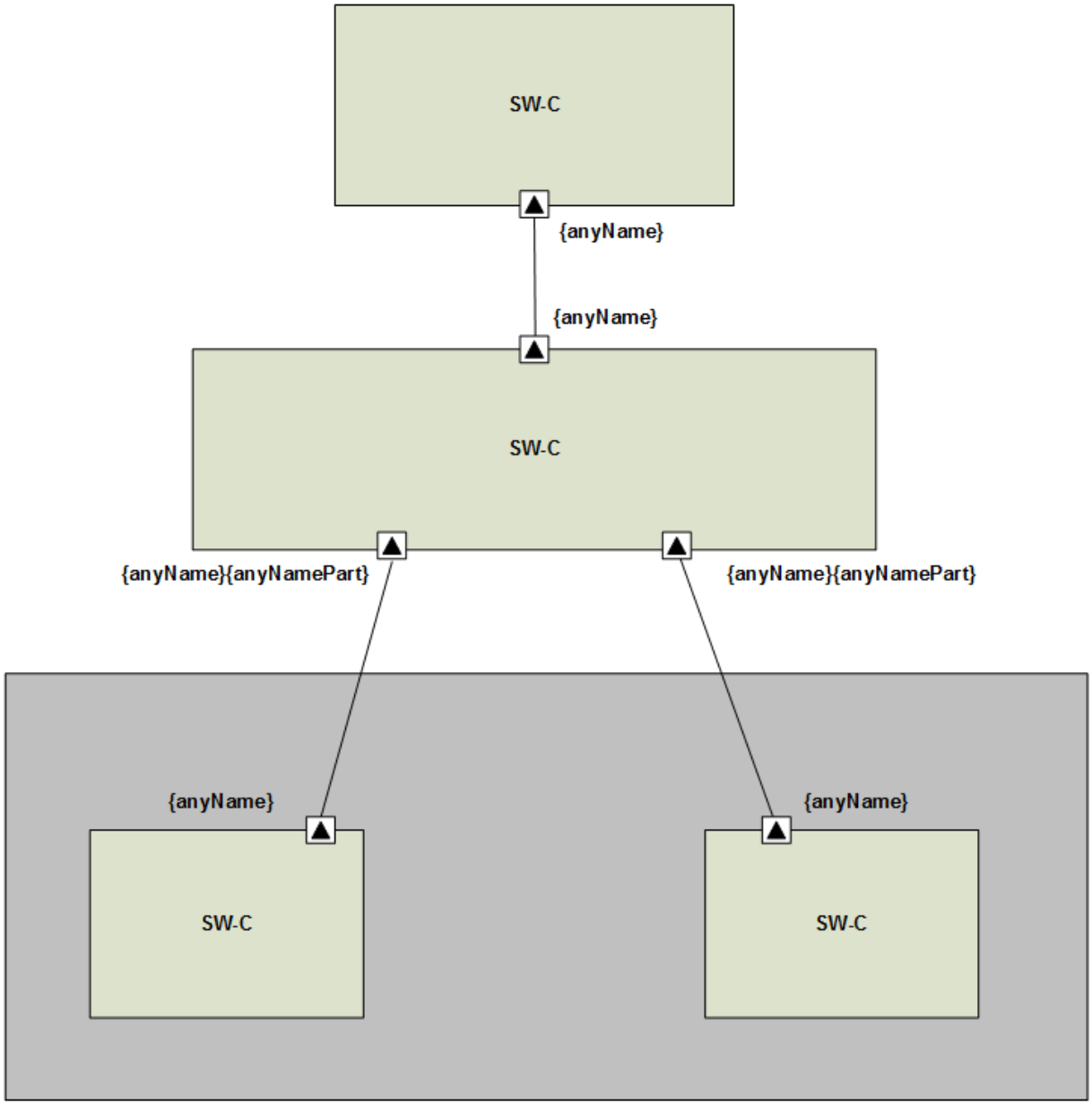


Figure 4.2: Pattern ”Arbitration between Several Providers”

An arbitration component is introduced to support several requesters of the same action but not necessarily of the same value.

An arbitration component is introduced to support several providers of the same signal.

4.4 Examples

4.4.1 Several Setpoint Requesters

In the context of the sensor/actuator pattern (see Chapter 3) there might be several conflicting setpoint requesters. In this case a new component for managing all requests from different setpoint requesters is introduced, see Figure 4.3.

When using sender/receiver interfaces the arbitration component, also called ”arbiter”, needs to have unique names for the different requests. This is realized by different request ports, one per requester. The port interface or at least the application data type is typically the same for all of these requesters and the resulting request.

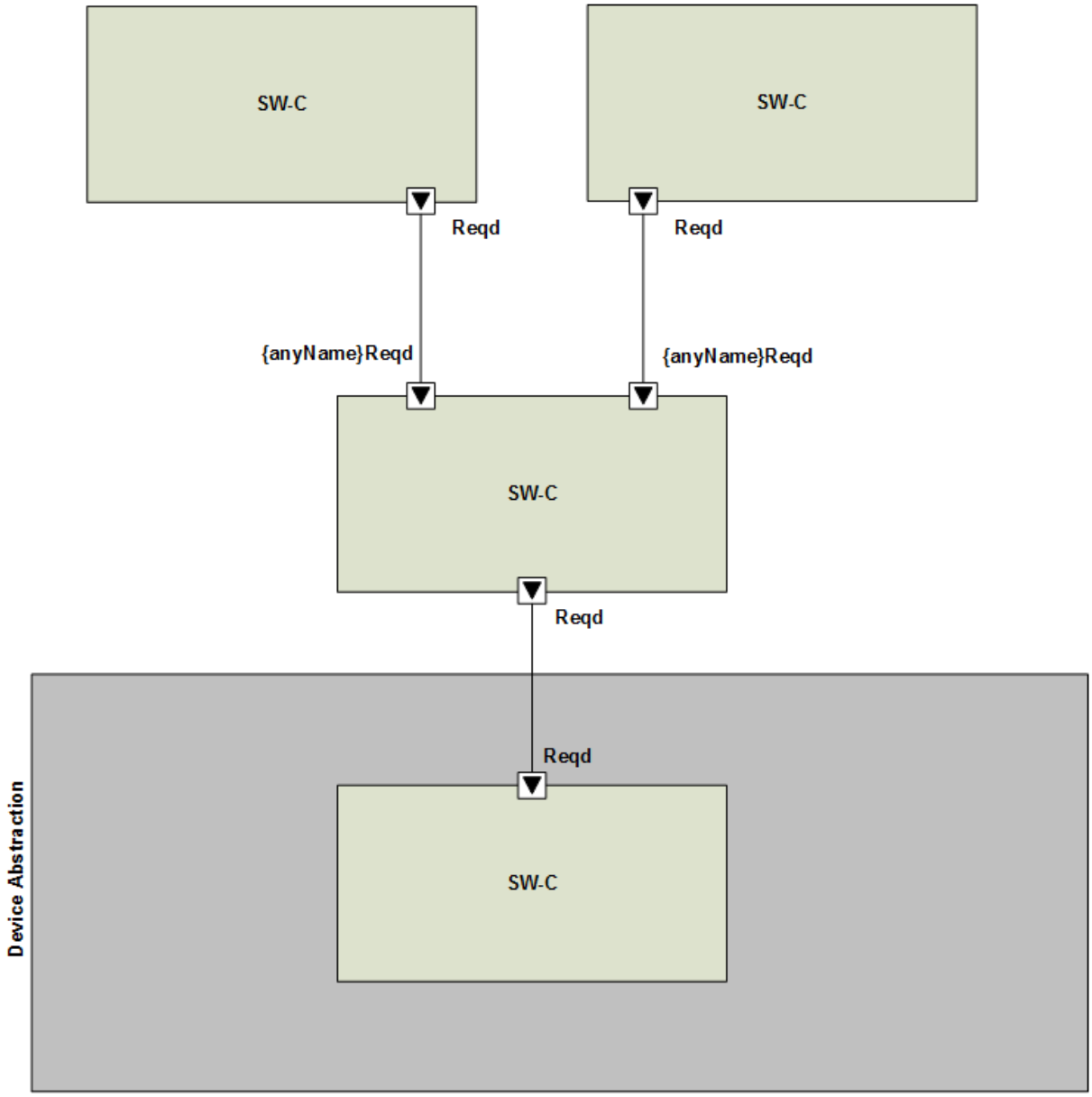


Figure 4.3: Pattern ”Arbitration between Several Set-point Requester”

In grammar 4.1 it is described how the provide ports of the requesters as well as the request ports of the arbiter should be named: they all have the suffix ”Reqd” for ”Required”. So terms like ”desired”, ”wished” etc. should not be used to avoid that too many terms with similar meanings are used without being able to distinguish them.

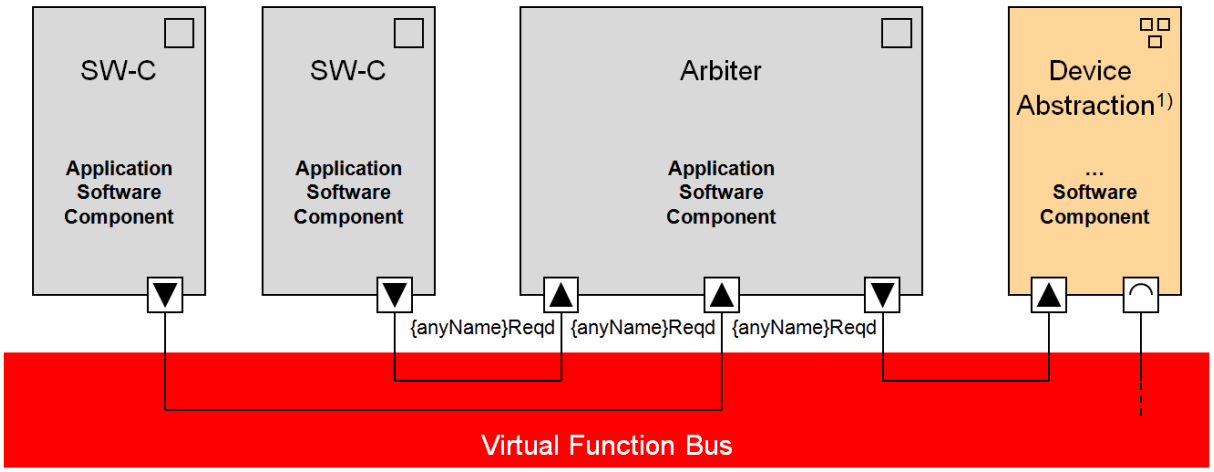
```
grammar PArbSpReqPortNames;

portName
    :    ({anyName}){'Reqd'} ;

anyName
    :    ('keyword')* ;
```

Listing 4.1: Name Pattern for Ports of Arbiter and Requesters

Figure 4.4 shows the pattern in the context of the RTE. The Device Abstraction is designed as one large composition but this is not requested by the Sensor/Actuator pattern.



1) Sensor and Actuator Design Pattern (PSnsrActr)

Figure 4.4: Arbitration between Several Requesters via RTE

4.4.2 Several Providers of Consolidated Values

In the context of the sensor/actuator pattern (3) there might be several sensors providing the same physical information. This is, there are several component all providing a consolidated values for a specific physical signal.

A new component for managing all consolidated values from different providers is introduced, see Figure 4.5.

When using sender/receiver interfaces the arbitration component, also called ”arbiter”, needs to have unique names for the different providers. This is realized by different

request ports, one per provider. The port interface or at least the application data type is typically the same for all of these providers and the resulting consolidated value.

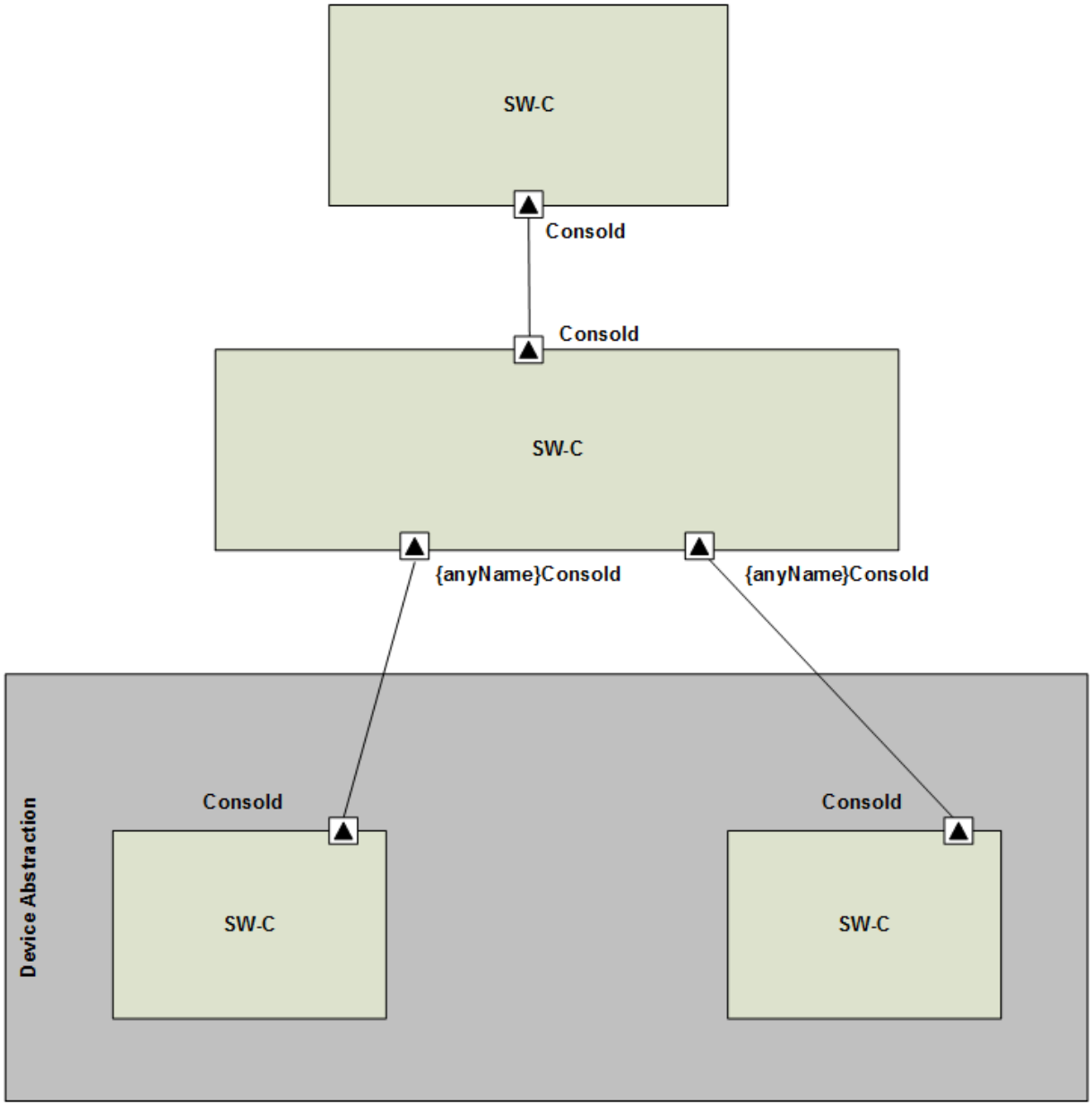


Figure 4.5: Pattern ”Arbitration between Several Providers of Consolidated Values”

In grammar 4.2 it is described how the provide ports of the providers as well as the provide port of the arbiter should be named: they all have the suffix ”Consold” for ”Consolidated”. So terms like ”modeled” etc. should not be used to avoid that too many terms with similar meanings are used without being able to distinguish them.

```
grammar PArbrConsoldPortNames;

portName
    :    ({anyName}){'Consold'} ;

anyName
```

```
:   ('keyword')* ;
```

Listing 4.2: Name Pattern for Ports of Arbiter and Providers of Consolidated Values

4.4.3 Several Providers of Estimated Values

In the context of the sensor/actuator pattern (3) there might be several model for calculating an estimation value. However, in the end only one of the estimated values should be input to the sensor/actuator pattern. Therefore, a new component for managing all estimated values from different providers is introduced, see Figure 4.6.

When using sender/receiver interfaces the arbitration component, also called ”arbiter”, needs to have unique names for the different providers. This is realized by different request ports, one per provider. The port interface or at least the application data type is typically the same for all of these providers and the resulting estimated value.

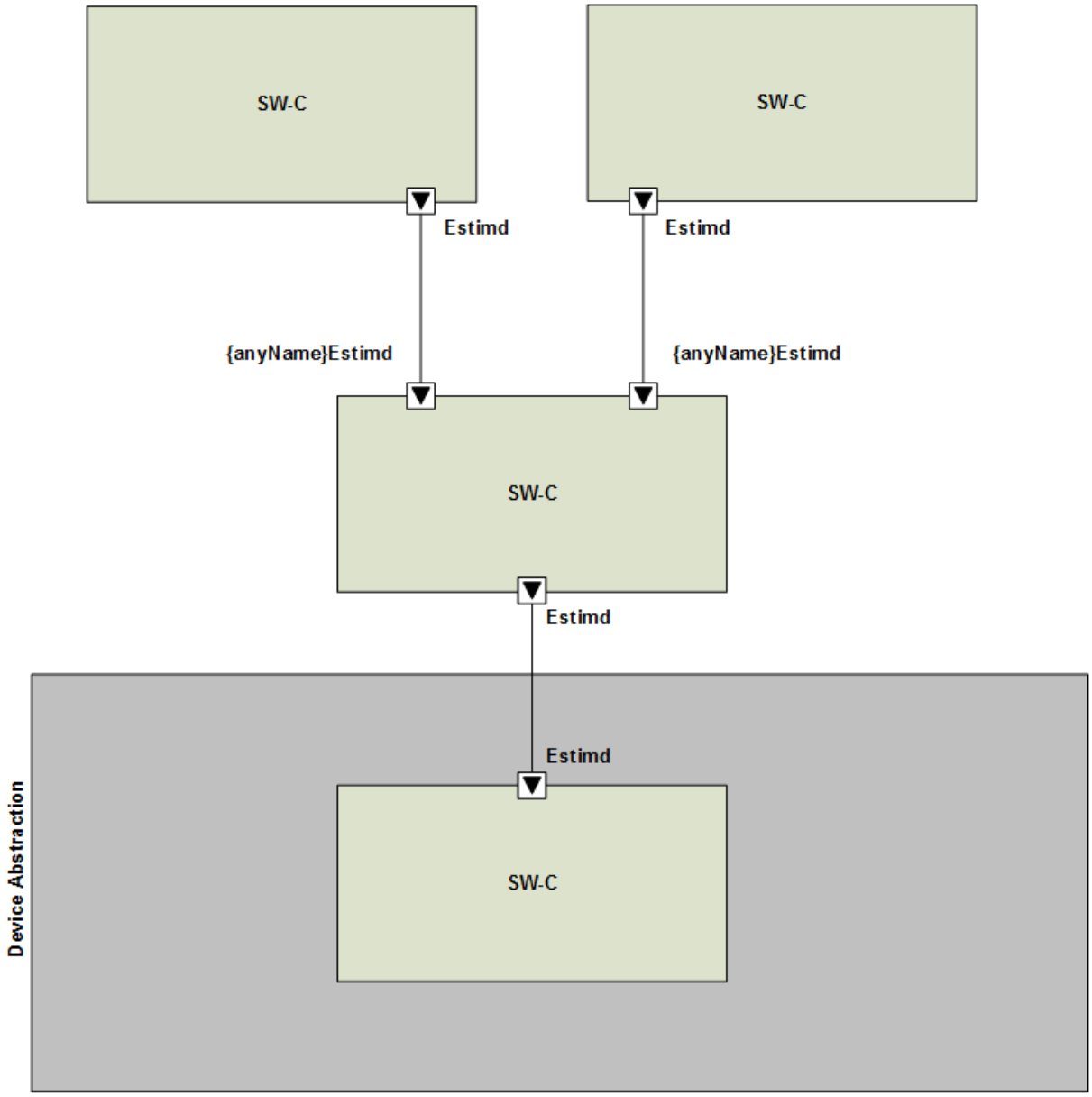


Figure 4.6: Pattern ”Arbitration between Several Providers of Estimated Values”

In grammar 4.3 it is described how the provide ports of the providers as well as the provide port of the arbiter should be named: they all have the suffix ”Estimd” for ”Estimated”. So terms like ”modeled” etc. should not be used to avoid that too many terms with similar meanings are used without being able to distinguish them.

```
grammar PArbEstimdPortNames;

portName
    :    ({anyName}){'Estimd'} ;

anyName
    :    ('keyword')* ;
```

Listing 4.3: Name Pattern for Ports of Arbiter and Providers of Estimated Values

4.5 Sample Code and Model

None.

4.6 Known Uses

This pattern is typically applied in the context of usage of the Sensor/Actuator Design Pattern.

4.7 Related Patterns

Pattern	Description
Sensor Actuator Pattern (see Chapter 3)	The sensor/actuator pattern is typically combined with the arbitration pattern to allow several set point requesters, several providers of consolidated values or several providers of estimated values. This is, arbitration is not done within the sensor/actuator pattern but outside the device abstraction.

Table 4.1: Related Patterns

5 Signal Quality States

Classification Design Pattern

5.1 Problem

For each (sensor) signal / value the corresponding quality information is also needed to be transferred along with the signal value.

The main intention is to have a common understanding of signal quality and to standardize the states a signal quality can have.

5.2 Applicability

This scope of this pattern is the definition of signal quality states (e.g. the content of the signal quality interfaces). The implementation of such a signal quality interface is not in scope of this document as there are several implementations possible.

The signal quality states defined in this document are a minimum set of recommended signal quality states.

5.3 Solution

Signal quality	State of related signal value	Meaning
`UNDEFINED`	Undefined value	No information about quality at all. It means that signal quality is not defined and the signal value is not initialized / calculated yet or is not calculated any more (e.g. desired deactivation of functionality)
`VALID`	Valid value	Trustworthy value from main signal source
`REPLACEMENT`	Replacement value with reduced validity	Modelled value or even defined constant value (mostly done by calibration). There is no information about the validity of the signal value, i.e. there is no additional information how "'good"' the replacement value represents the original value.
`FROZEN`	Frozen value	Frozen value. A valid value must have been calculated before. There is no information about since how long the signal value is frozen
`INVALID`	Invalid value	Value is not trustworthy and must not be used

Table 5.1: Signal Quality States

Additional information to table 5.1:

- Transitions from `UNDEFINED` to `FROZEN` is not allowed, because the previous value was not a valid value

- `UNDEFINED` level is default value of signal quality interfaces

5.3.1 Implementation proposal

For easier reuse and collaboration an example implementation of a signal quality in C code is given. It is recommended to store the signal quality along with its value in a `RECORD` (=`struct`) datatype. Having that `RTE` will guarantee data consistency.

```
typedef struct {
   float32 value;   //datatype depending on need
   uint8  quality;
} SignalQuality_t;
```

Listing 5.1: Signal quality implementation proposal

This is an example for a value given in `float` and its corresponding quality information, since `float` is the preferred datatype within *Device Abstraction* pattern for sensor signals. The datatype is variable (also refer to table 3.6).

A Change history of AUTOSAR traceable items

A.1 Traceable item history of this document according to AUTOSAR Release R24-11

A.1.1 Added Specification Items in R24-11

none

A.1.2 Changed Specification Items in R24-11

none

A.1.3 Deleted Specification Items in R24-11

none

A.2 Traceable item history of this document according to AUTOSAR Release R23-11

A.2.1 Added Specification Items in R23-11

none

A.2.2 Changed Specification Items in R23-11

none

A.2.3 Deleted Specification Items in R23-11

none

A.3 Traceable item history of this document according to AUTOSAR Release R22-11

A.3.1 Added Specification Items in R22-11

none

A.3.2 Changed Specification Items in R22-11

none

A.3.3 Deleted Specification Items in R22-11

Number	Heading
[TR_AIDPC_00001]	Access to Hardware by `PSnsrAct`
[TR_AIDPC_00002]	Collaboration supported by `PSnsrAct`
[TR_AIDPC_00003]	Deployment/Relocation supported by `PSnsrAct`
[TR_AIDPC_00004]	Layers of `PSnsrAct`
[TR_AIDPC_00005]	Naming within `PSnsrAct`
[TR_AIDPC_00006]	Arbitration of requesters
[TR_AIDPC_00007]	Arbitration of providers

Table A.1: Deleted Specification Items in R22-11

A.3.4 Added Constraints in R22-11

none

A.3.5 Changed Constraints in R22-11

none

A.3.6 Deleted Constraints in R22-11

none

B Mentioned Class Tables

For the sake of completeness, this chapter contains a set of class tables representing meta-classes mentioned in the context of this document but which are not contained directly in the scope of describing specific meta-model semantics.

Class	**ApplicationSwComponentType**			
Note	The `ApplicationSwComponentType` is used to represent the application software. **Tags:** atp.recommendedPackage=SwComponentTypes			
Base	*ARElement*, *ARObject*, *AtomicSwComponentType*, *AtpBlueprint*, *AtpClassifier*, *AtpType*, *Collectable Element*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *SwComponentType*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
–	–	–	–	–

Table B.1: ApplicationSwComponentType

Class	**CompositionSwComponentType**			
Note	A `CompositionSwComponentType` aggregates `SwComponentPrototype`s (that in turn are typed by `SwComponentType`)s as well as `SwConnector`s for primarily connecting `SwComponentPrototype`s among each others and towards the surface of the `CompositionSwComponentType`. By this means, a hierarchical structures of software-components can be created. **Tags:** atp.recommendedPackage=SwComponentTypes			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *SwComponentType*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
component	SwComponent Prototype	*	aggr	The instantiated components that are part of this composition. The aggregation of `SwComponentPrototype` is subject to variability with the purpose to support the conditional existence of a `SwComponentPrototype`. Please be aware: if the conditional existence of `SwComponentPrototype`s is resolved post-build, the deselected `SwComponentPrototype`s are still contained in the ECUs build but the instances are inactive in that they are not scheduled by the RTE. The use case for having 0 components owned by the `CompositionSwComponentType` could be to deliver an empty `CompositionSwComponentType` to e.g. a supplier for filling the internal structure. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=component, component.shortName, component.variationPoint.shortLabel vh.latestBindingTime=postBuild

▽

△

Class	**CompositionSwComponentType**			
connector	SwConnector	*	aggr	SwConnectors have the principal ability to establish a connection among PortPrototypes. They can have many roles in the context of a CompositionSwComponentType. Details are refined by subclasses. The aggregation of SwConnectors is subject to variability with the purpose to support variant data flow. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=connector, connector.shortName, connector.variationPoint.shortLabel vh.latestBindingTime=postBuild
constantValue Mapping	ConstantSpecification MappingSet	*	ref	Reference to the ConstantSpecificationMapping to be applied for initValues of PPortComSpecs and RPortComSpec. **Tags:** atp.Splitkey=constantValueMapping
dataType Mapping	DataTypeMappingSet	*	ref	Reference to the DataTypeMappingSet to be applied for the used ApplicationDataTypes in PortInterfaces. Background: when developing subsystems it may happen that ApplicationDataTypes are used on the surface of CompositionSwComponentTypes. In this case it would be reasonable to be able to also provide the intended mapping to the ImplementationDataTypes. However, this mapping shall be informal and not technically binding for the implementors mainly because the RTE generator is not concerned about the CompositionSwComponentTypes. Rationale: if the mapping of ApplicationDataTypes on the delegated and inner PortPrototype matches then the mapping to ImplementationDataTypes is not impacting compatibility. **Tags:** atp.Splitkey=dataTypeMapping
instantiation RTEEventProps	InstantiationRTEEvent Props	*	aggr	This allows to define instantiation specific properties for RTE Events, in particular for instance specific scheduling. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=instantiationRTEEventProps, instantiation RTEEventProps.shortLabel, instantiationRTEEvent Props.variationPoint.shortLabel vh.latestBindingTime=codeGenerationTime This Attribute is only used by the AUTOSAR Classic Platform.
physical Dimension Mapping	PhysicalDimension MappingSet	0..1	ref	This reference identifies the PhysicalDimensionMappingSet that is applicable in the context of the enclosing CompositionSwComponentType. The PhysicalDimensionMappings contained in the PhysicalDimensionMappingSet shall be taken into account for the assessment of the compatibility of PhysicalDimensions in the context of creation of a PortInterfaceMapping in the scope of the CompositionSwComponentType. **Tags:** atp.Splitkey=physicalDimensionMapping

Table B.2: CompositionSwComponentType

Class	**EcuAbstractionSwComponentType**			
Note	The ECUAbstraction is a special AtomicSwComponentType that resides between a software-component that wants to access ECU periphery and the Microcontroller Abstraction. The EcuAbstractionSw ComponentType introduces the possibility to link from the software representation to its hardware description provided by the ECU Resource Template. **Tags:** atp.recommendedPackage=SwComponentTypes			
Base	*ARElement*, *ARObject*, *AtomicSwComponentType*, *AtpBlueprint*, *AtpClassifier*, *AtpType*, *Collectable Element*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *SwComponentType*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
hardware Element	HwDescriptionEntity	*	ref	Reference from the EcuAbstractionComponentType to the description of the used HwElements. **Tags:** atp.Splitkey=hardwareElement

Table B.3: EcuAbstractionSwComponentType

Class	**HwCategory**			
Note	This metaclass represents the ability to declare hardware categories and its particular attributes. **Tags:** atp.recommendedPackage=HwCategorys			
Base	*ARElement*, *ARObject*, *AtpDefinition*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
hwAttributeDef	HwAttributeDef	*	aggr	This aggregation describes particular hardware attribute definition. **Tags:** atp.Splitkey=hwAttributeDef, hwAttributeDef.short Name

Table B.4: HwCategory

Class	***HwDescriptionEntity*** (abstract)			
Note	This meta-class represents the ability to describe a hardware entity.			
Base	*ARObject*, *Referrable*			
Subclasses	HwElement, HwPin, HwPinGroup, HwType			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
hwAttribute Value	HwAttributeValue	*	aggr	This aggregation represents a particular hardware attribute value. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=hwAttributeValue, hwAttributeValue.variation Point.shortLabel vh.latestBindingTime=systemDesignTime xml.sequenceOffset=50
hwCategory	HwCategory	*	ref	One of the associations representing one particular category of the hardware entity. **Tags:** atp.Splitkey=hwCategory xml.sequenceOffset=30

▽

△

Class	***HwDescriptionEntity*** (abstract)			
hwType	HwType	0..1	ref	This association is used to assign an optional HwType which contains the common attribute values for all occurences of this HwDescriptionEntity. Note that Hw Types can not be redefined and therefore shall not have a hwType reference. **Tags:** atp.Splitkey=hwType

Table B.5: HwDescriptionEntity

Class	**HwElement**			
Note	This represents the ability to describe Hardware Elements on an instance level. The particular types of hardware are distinguished by the category. This category determines the applicable attributes. The possible categories and attributes are defined in HwCategory. **Tags:** atp.recommendedPackage=HwElements			
Base	*ARElement*, *ARObject*, *CollectableElement*, *HwDescriptionEntity*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
hwElement Connection	HwElementConnector	*	aggr	This represents one particular connection between two hardware elements. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=hwElementConnection, hwElement Connection.variationPoint.shortLabel vh.latestBindingTime=systemDesignTime xml.sequenceOffset=110
hwPinGroup	HwPinGroup	*	aggr	This aggregation is used to describe the connection facilities of a hardware element. Note that hardware element has no pins but only pingroups. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=hwPinGroup, hwPinGroup.shortName, hw PinGroup.variationPoint.shortLabel vh.latestBindingTime=systemDesignTime xml.sequenceOffset=90
nestedElement	HwElement	*	ref	This association is used to establish hierarchies of hw elements. Note that one particular HwElement can be target of this association only once. I.e. multiple instantiation of the same HwElement is not supported (at any hierarchy level). **Stereotypes:** atpVariation **Tags:** atp.Splitkey=nestedElement, nestedElement.hwElement, nestedElement.variationPoint.shortLabel vh.latestBindingTime=systemDesignTime xml.sequenceOffset=70

Table B.6: HwElement

Class	**HwType**
Note	This represents the ability to describe Hardware types on an abstract level. The particular types of hardware are distinguished by the category. This category determines the applicable attributes. The possible categories and attributes are defined in HwCategory. **Tags:** atp.recommendedPackage=HwTypes

▽

△

Class	**HwType**			
Base	*ARElement*, *ARObject*, *CollectableElement*, *HwDescriptionEntity*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
–	–	–	–	–

Table B.7: HwType

Primitive	**Identifier**			
Note	An Identifier is a string with a number of constraints on its appearance, satisfying the requirements typical programming languages define for their Identifiers. This datatype represents a string, that can be used as a c-Identifier. It shall start with a letter, may consist of letters, digits and underscores. **Tags:** xml.xsd.customType=IDENTIFIER xml.xsd.maxLength=128 xml.xsd.pattern=[a-zA-Z][a-zA-Z0-9_]* xml.xsd.type=string			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
blueprintValue	String	0..1	attr	This represents a description that documents how the value shall be defined when deriving objects from the blueprint. **Tags:** atp.Status=draft xml.attribute=true
namePattern	String	0..1	attr	This attribute represents a pattern which shall be used to define the value of the identifier if the identifier in question is part of a blueprint. For more details refer to TPS_StandardizationTemplate. **Tags:** xml.attribute=true

Table B.8: Identifier

Class	**Keyword**			
Note	This meta-class represents the ability to predefine keywords which may subsequently be used to construct names following a given naming convention, e.g. the AUTOSAR naming conventions. Note that such names is not only shortName. It could be symbol, or even longName. Application of keywords is not limited to particular names.			
Base	*ARObject*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	KeywordSet.keyword			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
abbrName	NameToken	0..1	attr	This attribute specifies an abbreviated name of a keyword. This abbreviation may e.g. be used for constructing valid shortNames according to the AUTOSAR naming conventions. Unlike shortName, it may contain any name token. E.g. it may consist of digits only.
classification	NameToken	*	attr	This attribute allows to attach classification to the Keyword such as MEAN, ACTION, CONDITION, INDEX, PREPOSITION

Table B.9: Keyword

Class	***PortPrototype*** (abstract)			
Note	Base class for the ports of an AUTOSAR software component. The aggregation of PortPrototypes is subject to variability with the purpose to support the conditional existence of ports.			
Base	*ARObject*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Subclasses	*AbstractProvidedPortPrototype*, *AbstractRequiredPortPrototype*			
Aggregated by	*AtpClassifier*.atpFeature, *SwComponentType*.port			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
clientServer Annotation	ClientServerAnnotation	*	aggr	Annotation of this `PortPrototype` with respect to client/ server communication. **Tags:** atp.Splitkey=clientServerAnnotation
delegatedPort Annotation	DelegatedPort Annotation	0..1	aggr	Annotations on this delegated port. **Tags:** atp.Splitkey=delegatedPortAnnotation
ioHwAbstraction Server Annotation	IoHwAbstractionServer Annotation	*	aggr	Annotations on this IO Hardware Abstraction port. **Tags:** atp.Splitkey=ioHwAbstractionServerAnnotation
modePort Annotation	ModePortAnnotation	*	aggr	Annotations on this mode port. **Tags:** atp.Splitkey=modePortAnnotation
nvDataPort Annotation	NvDataPortAnnotation	*	aggr	Annotations on this non voilatile data port. **Tags:** atp.Splitkey=nvDataPortAnnotation
parameterPort Annotation	ParameterPort Annotation	*	aggr	Annotations on this parameter port. **Tags:** atp.Splitkey=parameterPortAnnotation
senderReceiver Annotation	SenderReceiver Annotation	*	aggr	Collection of annotations of this ports sender/receiver communication. **Tags:** atp.Splitkey=senderReceiverAnnotation
triggerPort Annotation	TriggerPortAnnotation	*	aggr	Annotations on this trigger port. **Tags:** atp.Splitkey=triggerPortAnnotation

Table B.10: PortPrototype

Class	**PortPrototypeBlueprint**			
Note	This meta-class represents the ability to express a blueprint of a PortPrototype by referring to a particular PortInterface. This blueprint can then be used as a guidance to create particular PortPrototypes which are defined according to this blueprint. By this it is possible to standardize application interfaces without the need to also standardize software-components with PortPrototypes typed by the standardized Port Interfaces. **Tags:** atp.recommendedPackage=PortPrototypeBlueprints			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *Collectable Element*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Aggregated by	ARPackage.element, *AtpClassifier*.atpFeature			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
initValue	PortPrototypeBlueprint InitValue	*	aggr	This specifies the init values for the dataElements in the particular PortPrototypeBlueprint. **Tags:** atp.Splitkey=initValue
interface	PortInterface	0..1	ref	This is the interface for which the blueprint is defined. It may be a blueprint itself or a standardized PortInterface **Tags:** atp.Splitkey=interface

▽

△

Class	**PortPrototypeBlueprint**			
providedCom Spec	PPortComSpec	*	aggr	Provided communication attributes per interface element (data element or operation). **Tags:** atp.Splitkey=providedComSpec, providedCom Spec.dataElement, providedComSpec.getter, provided ComSpec.modeGroup, providedComSpec.operation, providedComSpec.parameter, providedComSpec.setter, providedComSpec.variable
requiredCom Spec	RPortComSpec	*	aggr	Required communication attributes, one for each interface element. **Tags:** atp.Splitkey=requiredComSpec, requiredCom Spec.dataElement, requiredComSpec.getter, required ComSpec.modeGroup, requiredComSpec.operation, requiredComSpec.parameter, requiredComSpec.setter, requiredComSpec.shortLabel, requiredComSpec.variable

Table B.11: PortPrototypeBlueprint

Class	***Referrable*** (abstract)			
Note	Instances of this class can be referred to by their identifier (while adhering to namespace borders).			
Base	*ARObject*			
Subclasses	*AtpDefinition*, BswDistinguishedPartition, *BswModuleCallPoint*, BswModuleClientServerEntry, Bsw VariableAccess, CouplingPortTrafficClassAssignment, *DiagnosticEnvModeElement*, EthernetPriority Regeneration, ExclusiveAreaNestingOrder, *HwDescriptionEntity*, *ImplementationProps*, LinSlaveConfig Ident, ModeTransition, *MultilanguageReferrable*, PncMappingIdent, *SingleLanguageReferrable*, SoCon IPduIdentifier, TpConnectionIdent			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
shortName	Identifier	1	attr	This specifies an identifying shortName for the object. It needs to be unique within its context and is intended for humans but even more for technical reference. **Stereotypes:** atpIdentityContributor **Tags:** xml.enforceMinMultiplicity=true xml.sequenceOffset=-100
shortName Fragment	ShortNameFragment	*	aggr	This specifies how the Referrable.shortName is composed of several shortNameFragments. **Tags:** atp.Splitkey=shortNameFragment xml.sequenceOffset=-90

Table B.12: Referrable

Class	**SensorActuatorSwComponentType**			
Note	The SensorActuatorSwComponentType introduces the possibility to link from the software representation of a sensor/actuator to its hardware description provided by the ECU Resource Template. **Tags:** atp.recommendedPackage=SwComponentTypes			
Base	*ARElement*, *ARObject*, *AtomicSwComponentType*, *AtpBlueprint*, *AtpClassifier*, *AtpType*, *Collectable Element*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *SwComponentType*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***

▽

△

Class	**SensorActuatorSwComponentType**			
sensorActuator	HwDescriptionEntity	0..1	ref	Reference from the Sensor Actuator Software Component Type to the description of the actual hardware. **Tags:** atp.Splitkey=sensorActuator

Table B.13: SensorActuatorSwComponentType

Class	**SwComponentPrototype**			
Note	Role of a software component within a composition.			
Base	*ARObject*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	*AtpClassifier*.atpFeature, CompositionSwComponentType.component			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
type	SwComponentType	0..1	tref	Type of the instance. **Stereotypes:** isOfType **Tags:** atp.Splitkey=type

Table B.14: SwComponentPrototype

Class	***SwComponentType*** (abstract)			
Note	Base class for AUTOSAR software components.			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Subclasses	*AtomicSwComponentType*, CompositionSwComponentType, ParameterSwComponentType			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
consistency Needs	ConsistencyNeeds	*	aggr	This represents the collection of `ConsistencyNeeds` owned by the enclosing `SwComponentType`. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=consistencyNeeds, consistencyNeeds.shortName, consistencyNeeds.variationPoint.shortLabel vh.latestBindingTime=preCompileTime This Attribute is only used by the AUTOSAR Classic Platform.
port	PortPrototype	*	aggr	The `PortPrototype`s through which this `SwComponentType` can communicate. The aggregation of `PortPrototype` is subject to variability with the purpose to support the conditional existence of `PortPrototype`s. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=port, port.shortName, port.variationPoint.shortLabel vh.latestBindingTime=preCompileTime
portGroup	PortGroup	*	aggr	A port group being part of this component. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=portGroup, portGroup.shortName, portGroup.variationPoint.shortLabel vh.latestBindingTime=preCompileTime

▽

△

Class	***SwComponentType*** (abstract)			
swcMapping Constraint	SwComponentMapping Constraints	*	ref	Reference to constraints that are valid for this Sw ComponentType. **Tags:** atp.Splitkey=swcMappingConstraint This Attribute is only used by the AUTOSAR Classic Platform.
swComponent Documentation	SwComponent Documentation	0..1	aggr	This adds a documentation to the `SwComponentType`. **Stereotypes:** atpVariation **Tags:** atp.Splitkey=swComponentDocumentation, sw ComponentDocumentation.variationPoint.shortLabel vh.latestBindingTime=preCompileTime xml.sequenceOffset=-10
unitGroup	UnitGroup	*	ref	This allows for the specification of which `UnitGroup`s are relevant in the context of referencing `SwComponentType`. **Tags:** atp.Splitkey=unitGroup This Attribute is only used by the AUTOSAR Classic Platform.

Table B.15: SwComponentType