Document Title	Design Patterns Catalogue
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	672
Document Classification	Auxiliary

Document Status	Final
Part of AUTOSAR Release	4.2.2

Document Change History		
Release	**Changed by**	**Description**
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
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

Advice for users

AUTOSAR specifications may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the specifications for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such specifications, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

References

[1] ANTLR parser generator V3

[2] Standardization Template
AUTOSAR_TPS_StandardizationTemplate

[3] SW-C and System Modeling Guide
AUTOSAR_TR_SWCModelingGuide

[4] XML Specification of Application Interfaces
AUTOSAR_MOD_AISpecification

[5] Main Requirements
AUTOSAR_RS_Main

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

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

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

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

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

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

[12] Software Component Template
AUTOSAR_TPS_SoftwareComponentTemplate

[13] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture

1 Introduction

1.1 Document conventions

Technical terms (Class Names) are typeset in monospaced 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 [TR_SWNR_0064] 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 captial letter as well.

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

1.2 Requirements Tracing

Requirements against this document are stated in the requirements document [5].

The following table references the requirements specified in [5] and provides information about individual specification items that fulfill a given requirement.

Requirement	Description	Satisfied by
[RS_MAIN_00080]	AUTOSAR shall provide means to describe a component model for application software	[TR_AIDPC_00001] [TR_AIDPC_00002]
[RS_MAIN_00130]	AUTOSAR shall provide an abstraction from hardware	[TR_AIDPC_00001] [TR_AIDPC_00002]
[RS_MAIN_00140]	AUTOSAR shall provide network independent communication mechanisms for applications	[TR_AIDPC_00001] [TR_AIDPC_00002] [TR_AIDPC_00003]
[RS_MAIN_00150]	AUTOSAR shall support the reallocation of Software Components	[TR_AIDPC_00001] [TR_AIDPC_00002]
[RS_MAIN_00400]	AUTOSAR shall provide a layered software architecture	[TR_AIDPC_00001] [TR_AIDPC_00002] [TR_AIDPC_00003] [TR_AIDPC_00004]
[RS_MAIN_00410]	AUTOSAR shall provide specifications for routines commonly used by Software Components to support sharing and optimization	[TR_AIDPC_00003]
[RS_MAIN_00500]	AUTOSAR shall provide naming conventions	[TR_AIDPC_00005]

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 [6].

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 [7].

Solution Pattern A solution pattern describes a generic solution for a specific problem like for example error handling or job scheduling [6].

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 [8].

Anti-Patterns In software engineering, an anti-pattern (or antipattern) 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 [9].

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 [7], [10], [11], [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 Problem

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 Sensor/Actuator Design Pattern 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.

3.2 Also Known As

This pattern is also known as *Device Abstraction*.

3.3 Applicability

[TR_AIDPC_00001] Access to Hardware by `PSnsrAct` ⌈

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. ⌋*(RS_MAIN_00080, RS_MAIN_00130, RS_MAIN_00140, RS_MAIN_00150, RS_MAIN_00400)*

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.

[TR_AIDPC_00002] Collaboration supported by `PSnsrAct` ⌈ 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. ⌋*(RS_MAIN_00080, RS_MAIN_00130, RS_MAIN_00140, RS_MAIN_00150, RS_MAIN_00400)*

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.

[TR_AIDPC_00003] Deployment/Relocation supported by `PSnsrAct` ⌈ 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. ⌋*(RS_MAIN_00140, RS_MAIN_00400, RS_MAIN_00410)*

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 [12] 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.

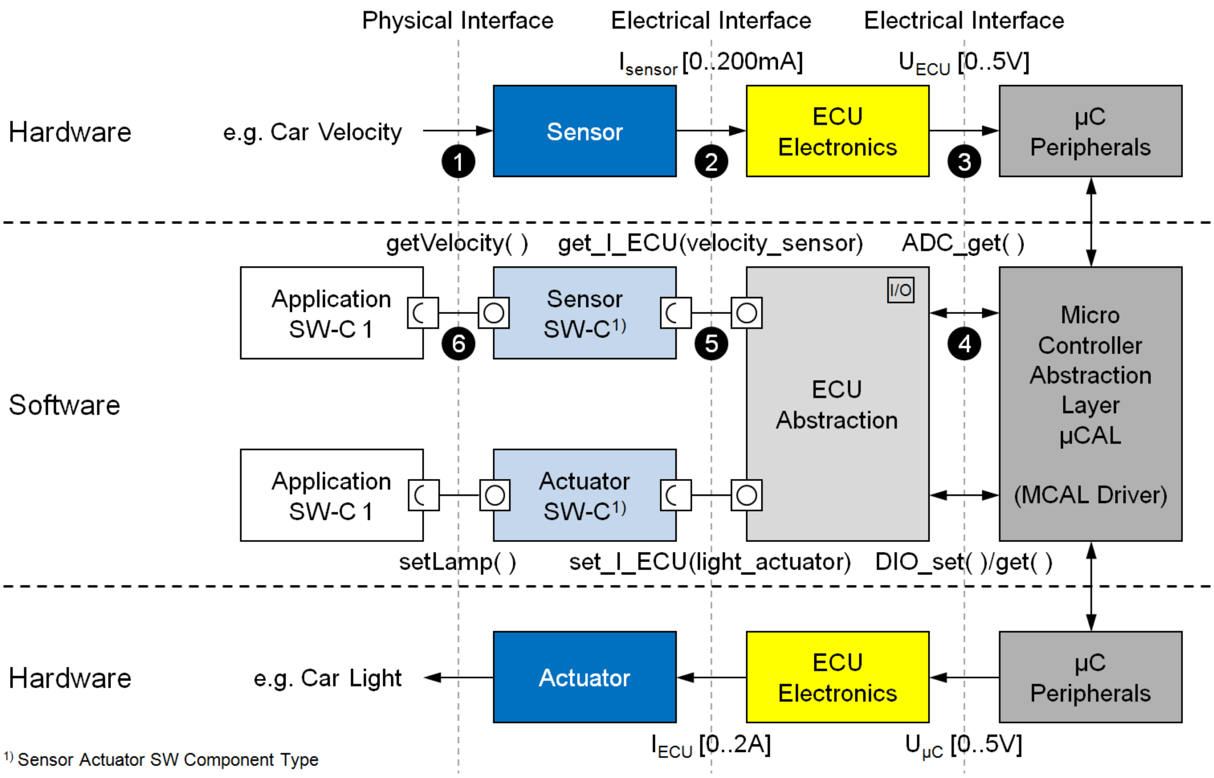

Figure 3.1: Sensor Actuator Signal Flow [12]

[TR_AIDPC_00004] Layers of `PSnsrAct` ⌈ 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.

⌋*(RS_MAIN_00400)*

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.

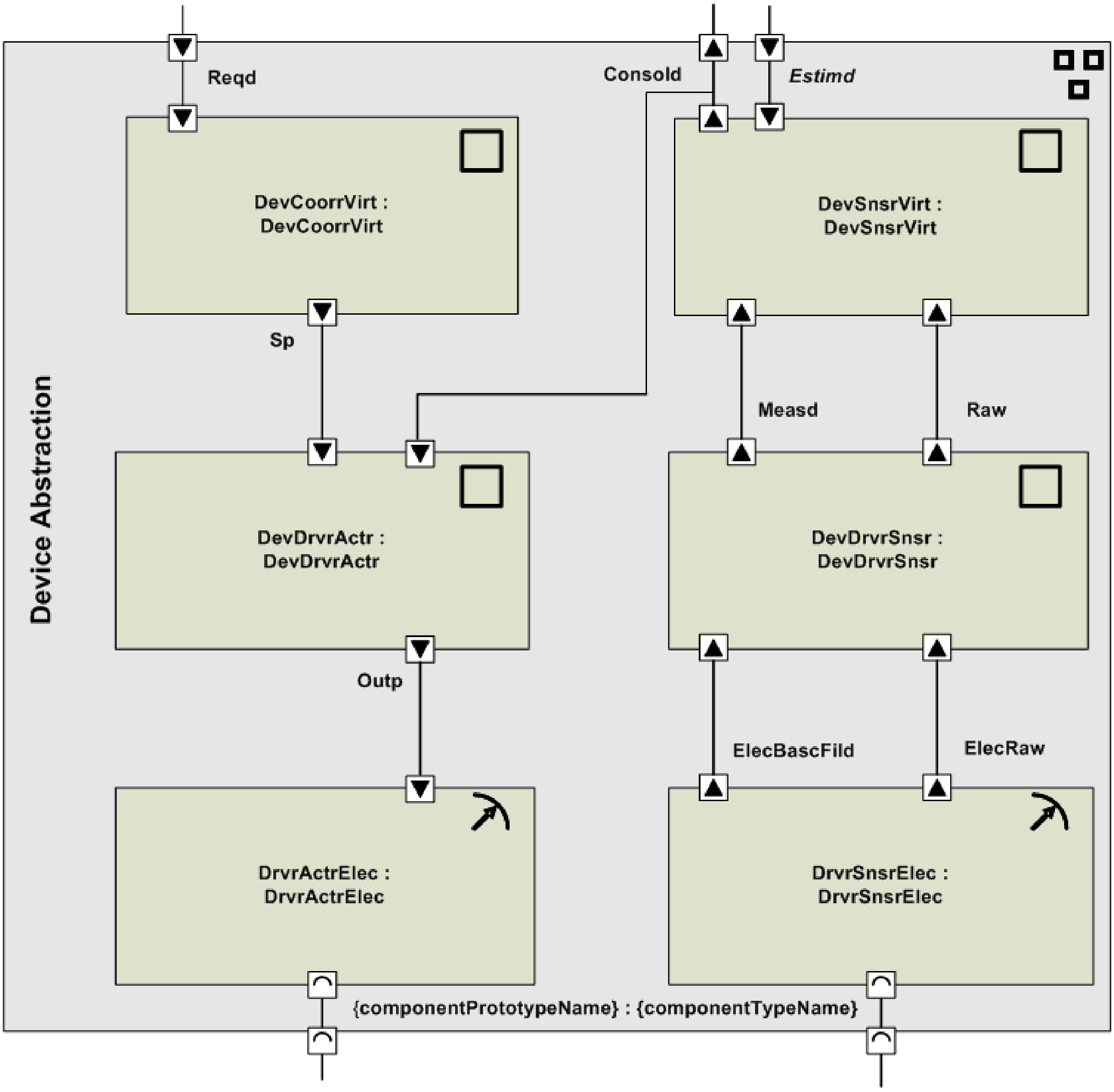

Figure 3.2: Sensor Actuator Pattern for Closed Loop

The application software can rely on the existance 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 3.8 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 is provided (Consold) and the estimated value (Estimd) is requested, see Figure 3.9.

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.

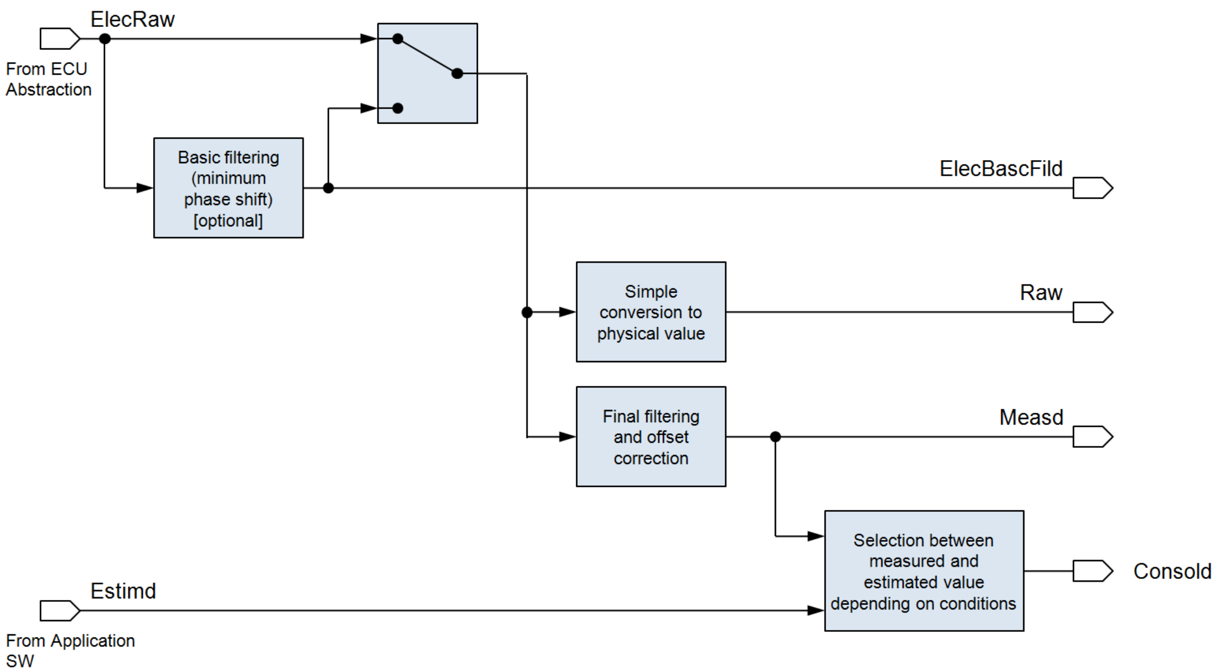

Figure 3.3: Signal Flow within Sensor and Actuator Pattern

Please be aware: `SensorActuatorSwComponentType`s are the only components that are allowed to access ECU Abstraction Software, namely `EcuAbstraction-SwComponentType`. This is shown in Figure 3.4 taken from [13]. Access is denoted by ”IO”.

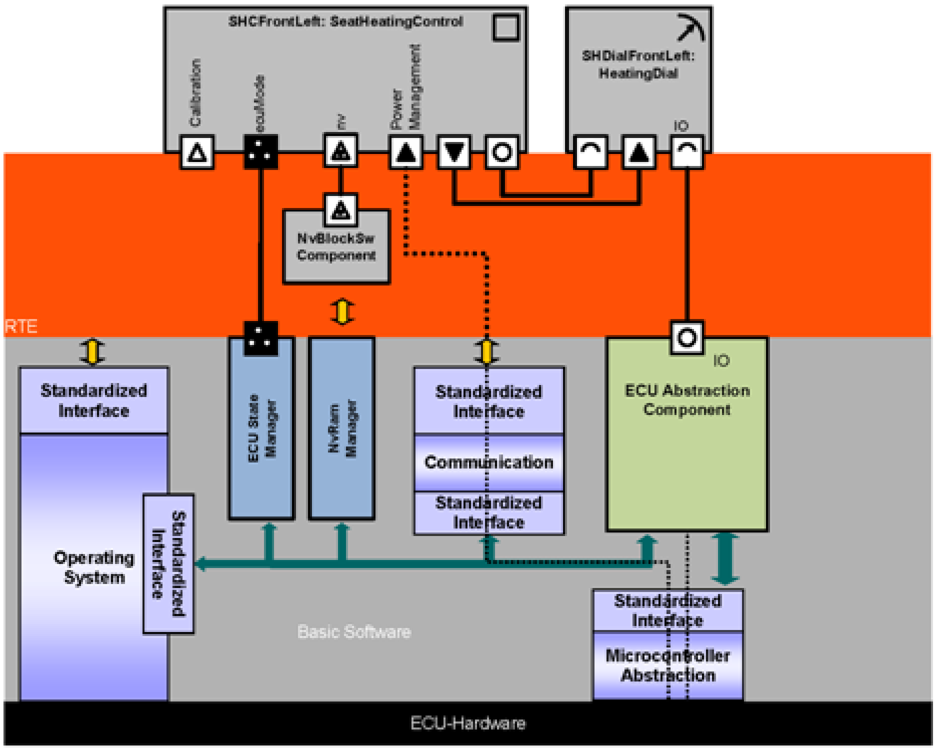

Figure 3.4: Access to ECU Abstraction

3.5 Naming

[TR_AIDPC_00005] Naming within `PSnsrAct` ⌈ 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.
⌋*(RS_MAIN_00500)*

Listing 3.1: Name Pattern for Ports in Device Abstraction

```
grammar PSnsrActrPortNames;

portName
    :    {'sensorActuatorSignal'} ;

sensorActuatorSignal
    :    {anyName}{'sensorActuatorSignalType'} ;

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

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

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. Electrical signals can only be represented in voltage, current and time [12].
`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 [12]. Electrical signals can only be represented in voltage, current and time.
`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	Final filtered and offset corrected physical sensor value replacement model value for physical sensor value/standard sensor value.

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 intialization 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.

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
TrboChrgrStg3AtBnk2	FlatInstanceDescriptor	Value of Turbo Charger at Third Stage at Second Bank
TrboChrgr	PortPrototype	Value of Turbo Charger

Table 3.2: Port Names Examples

In grammar 3.2 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.

Listing 3.2: Name Pattern for Atomic Software Component Types in Device Abstraction

```
grammar PSnsrActrAtomicSwcShortName;

sensorActuatorComponentTypeName
   :   sensorActuatorComponentName ;

sensorActuatorComponentPrototypeName
   :   sensorActuatorComponentName ;

sensorActuatorComponentName
```

```
    :    (Drv{Device}Elec | DevDrv{Device} | Dev{Device}Virt | DevCoorrVirt)(
        'anyNamePart') ;

Device
    :    ( Snsr | Actr ) ;

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

In grammar 3.3 the pattern is more refined but still conforming to grammar 3.2 because "For" is a standardized keyword. Note: the refined grammar is following [TR_SWNR_0034] that requests that field blocks are concatenated by adding an appropriate preposition.

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

```
grammar PSnsrActrAtomicSwcShortNameRefined;

sensorActuatorComponentTypeName
    :    sensorActuatorComponentName ;

sensorActuatorComponentPrototypeName
    :    sensorActuatorComponentName ;

sensorActuatorComponentName
    :    (Drv{deviceType}Elec | DevDrv{deviceType} | Dev{deviceType}Virt |
        DevCoorrVirt) ({device}) ;

deviceType
    :    ( Snsr | Actr ) ;

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

sensor
    :    'anyName' ;

actuator
    :    'anyName' ;

anyName
    :    ('keyword')* ;

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

In grammar 3.4 the pattern for the corresponding English long names of the components is described.

Listing 3.4: Pattern for English Long Names Atomic Software Component Types in Device Abstraction

```
grammar PSnsrActrAtomicSwcLongName;

sensorActuatorComponentLongName
```

```
    :   sensorActuatorComponentName ;

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

anyLongName
    :   ('keyword')* ;

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

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	SensorActuatorSwComponentType	TLE8209: Electrical Sensor Driver
DrvrActrElecForTrboChrgr	SwComponentPrototype	Turbo Charger: Electrical Sensor Driver
DevSnsrVirtForAnyTSnsr	ApplicationSwComponentType	Virtual Device Driver for Any Temperature Sensor
DevSnsrVirtForTrboChrgr	SwComponentPrototype	Turbo Charger: Virtual Device Driver
TrboChrgrAcmeT064	CompositionSwComponentType	Turbo Charger: ACME T064
TrboChrgrStg3AtBnk2	SwComponentPrototype	Turbo Charger at Third Stage at First Bank

Table 3.4: Examples for Sensor and Actuator Names

In grammar 3.5 a pattern is described how to refine 'anyNamePart' as defined in grammar 3.3 in case of a system with several banks and stages. In Table 3.5 corresponding name examples are shown using this grammar part.

Listing 3.5: Name Pattern for Signals in Device Abstraction in Case of a System with Several Banks

```
grammar PSnsrActrStgBnkShortNames;

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

indexStg
```

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

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

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: Examples for Sensor and Actuator Names

3.6 Example

3.6.1 Throttle Valve

Figure 3.5 shows an example device abstraction for a throttle valve.

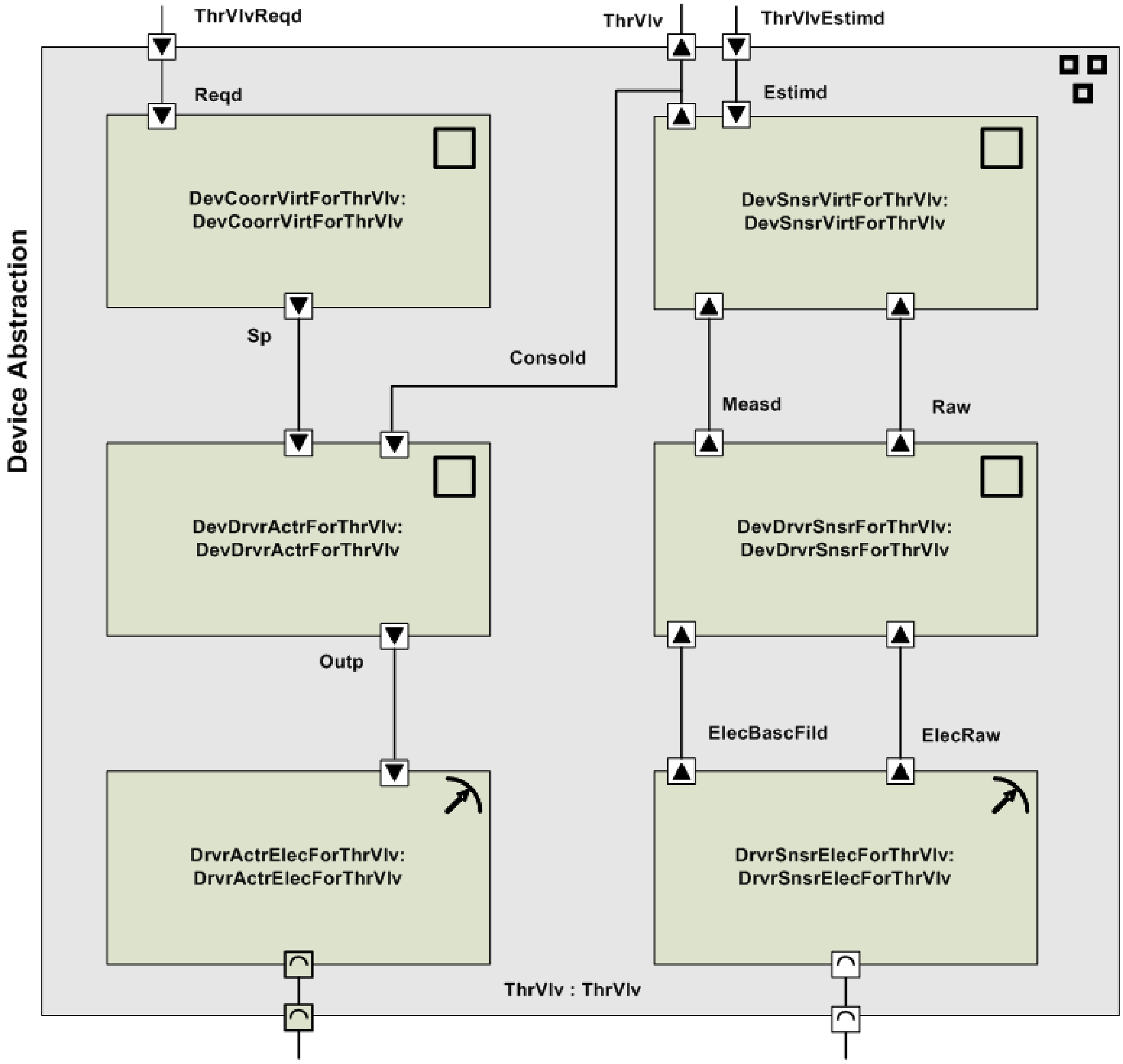

Figure 3.5: Device Abstraction for a Throttle Valve

3.6.2 Turbo Charger

In Figure 3.6 an example of a closed looped controlled device with position feedback — a turbo charger — is shown.

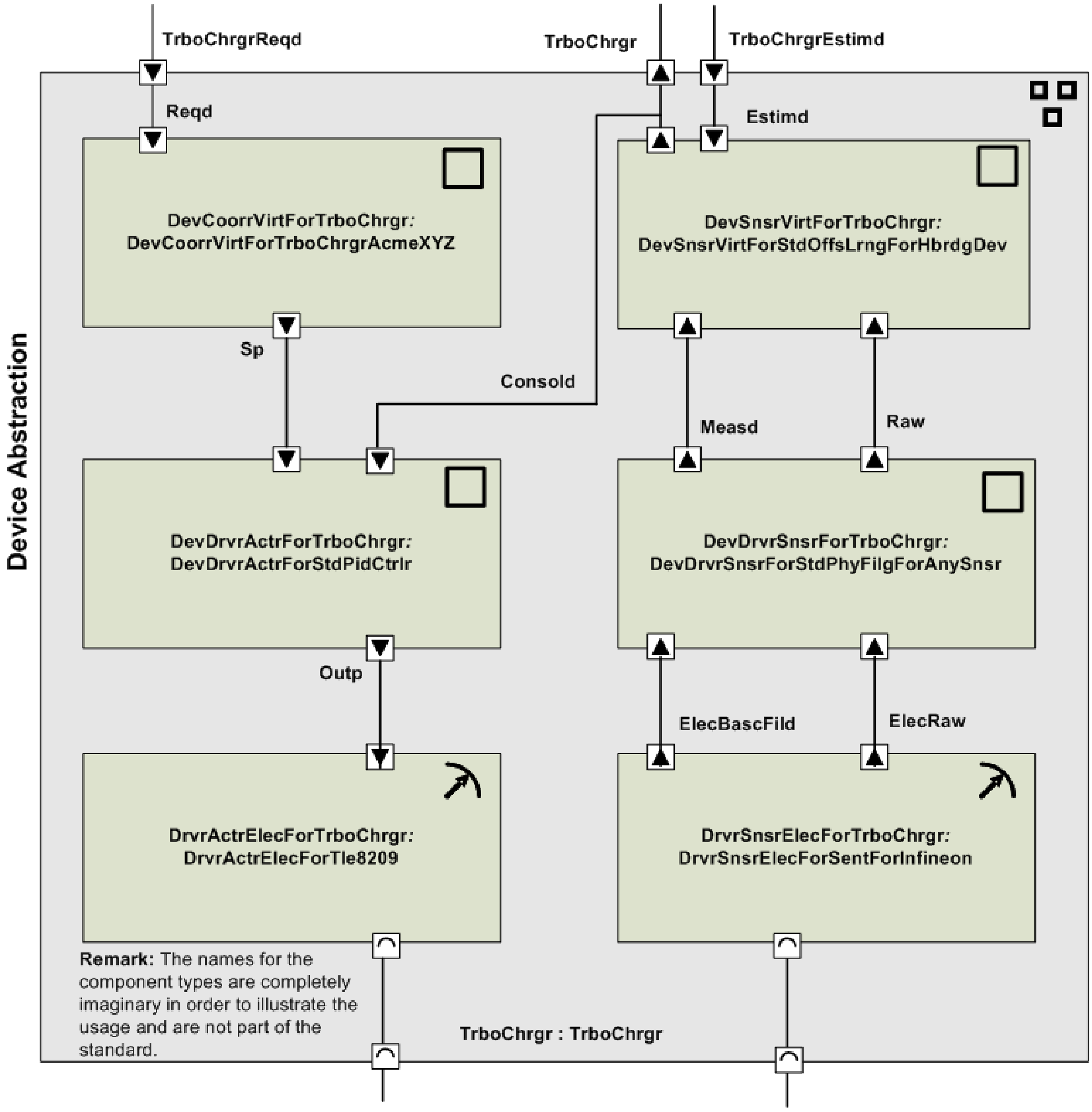

Figure 3.6: Device Abstraction for a Turbo Charger

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.

3.6.3 Turbo Charger with Stages and Banks

In Figure 3.7 a project system configuration for turbo charger with several stages and banks is shown.

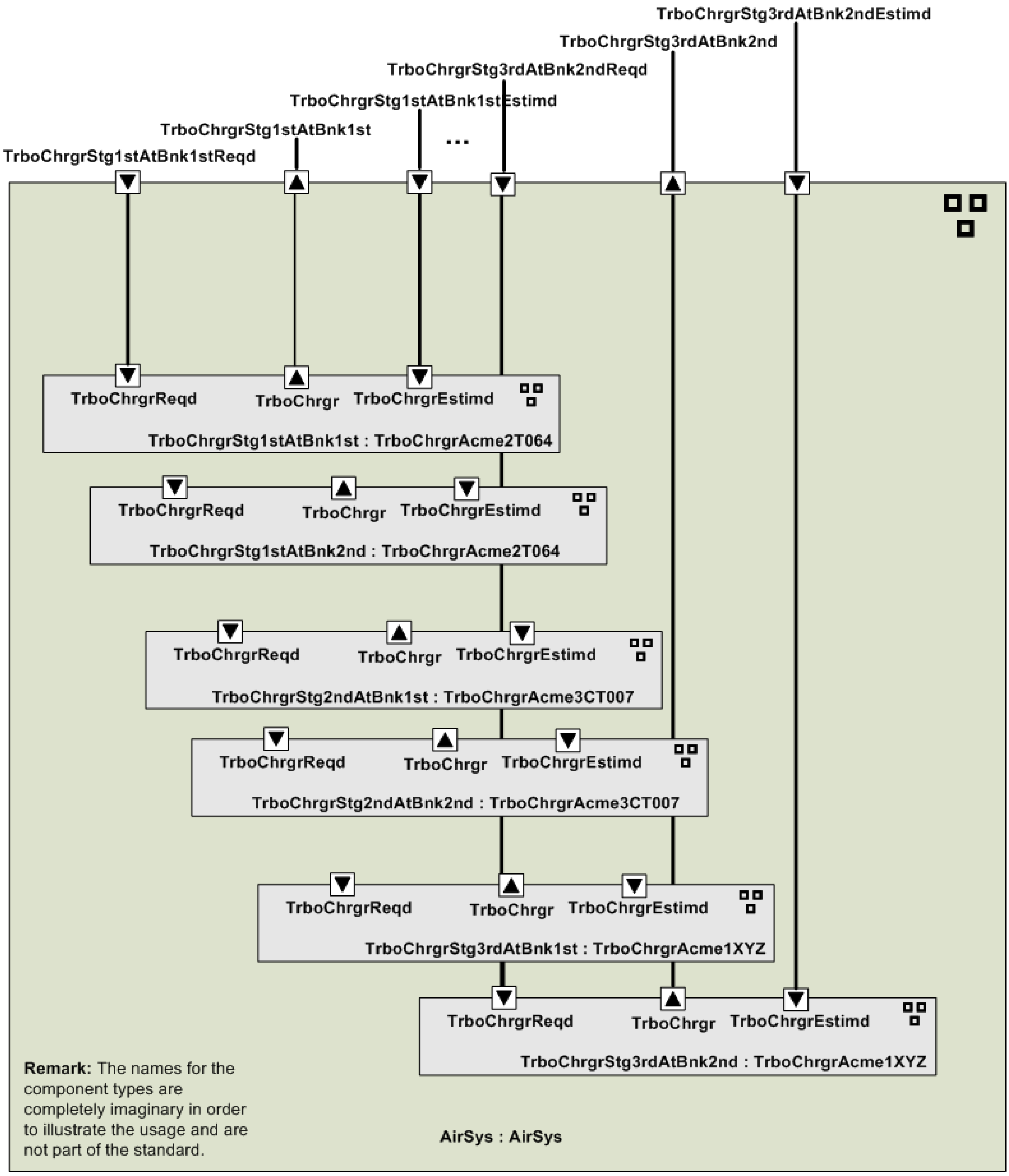

Figure 3.7: Device Abstraction for a Turbo Charger with Banks and Stages

3.6.4 Actuator without Feedback Loop

In Figure 3.8 an open loop controlled actuator is shown that calculates the consolidated value using the setpoint input as input. As described before there are alternatives how to calculate the consolidated value.

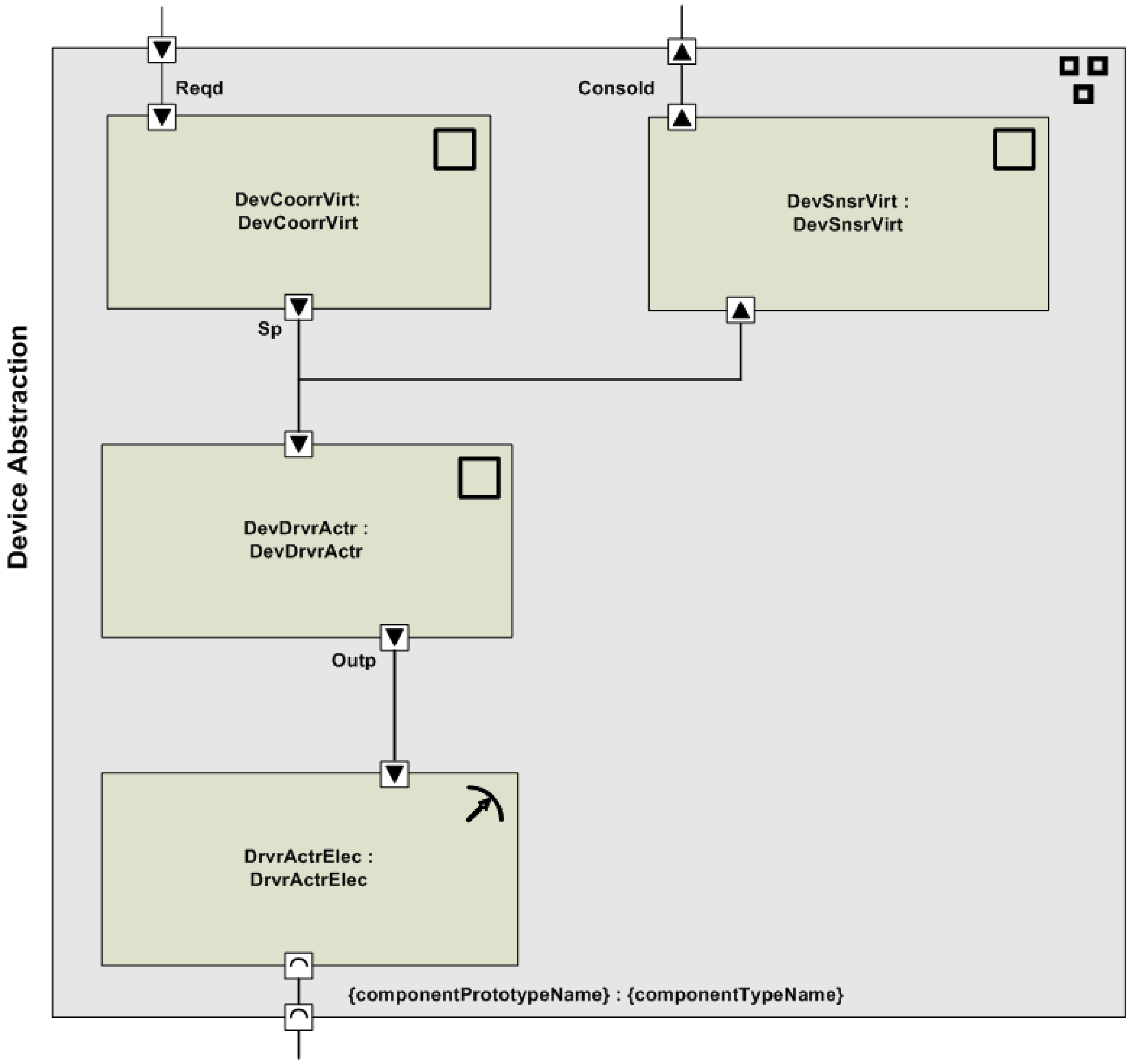

Figure 3.8: Example Actuator without Feedback Loop (Setpoint Alternative)

3.6.5 Standard Sensor

In Figure 3.9 a design pattern of blueprint components for a standard sensor is shown.

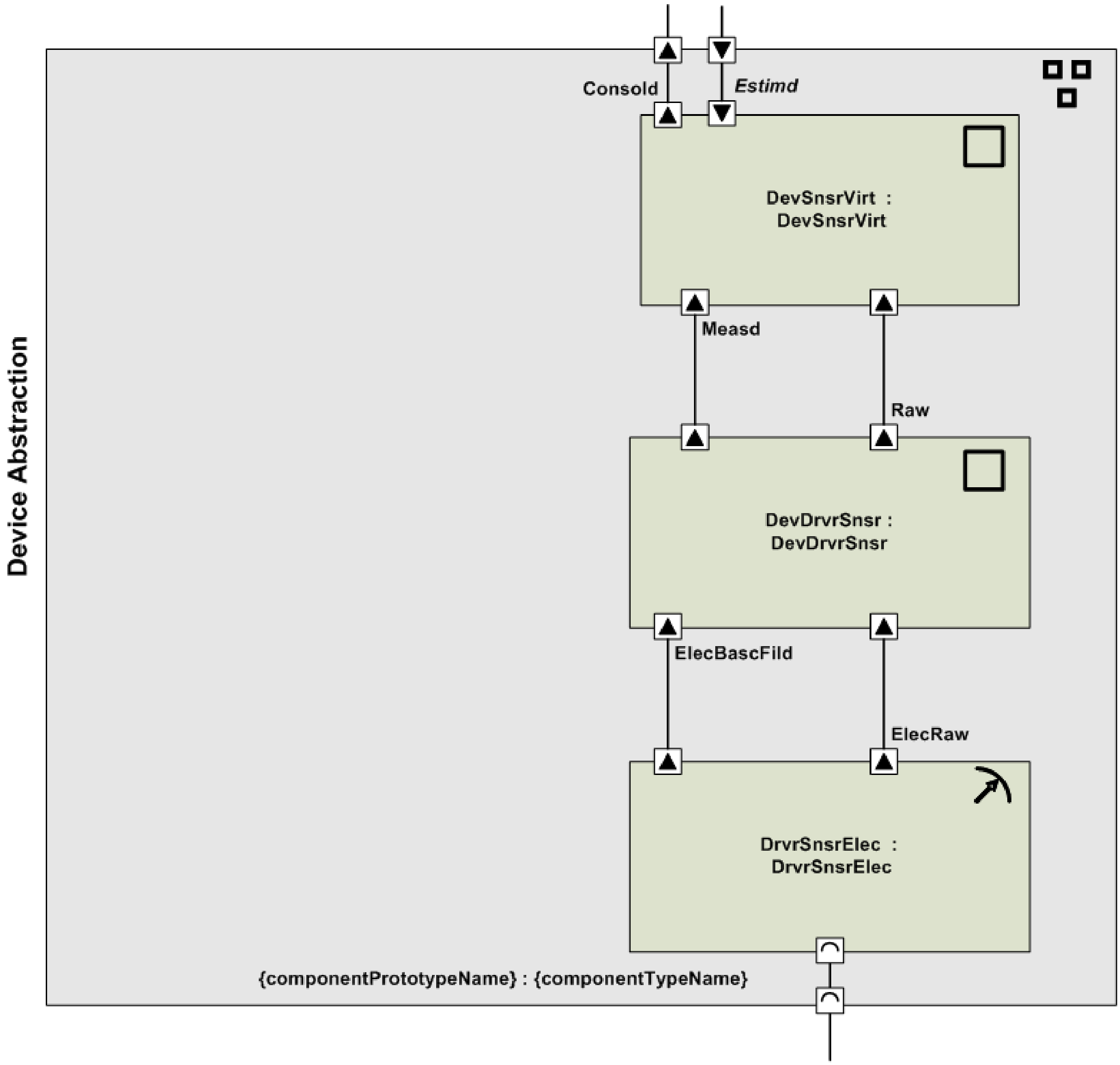

Figure 3.9: Device Abstraction for Standard Sensor

3.6.6 Standard Sensor for Environment Temperature

In Figure 3.10 a standard sensor for environment temperature is shown.

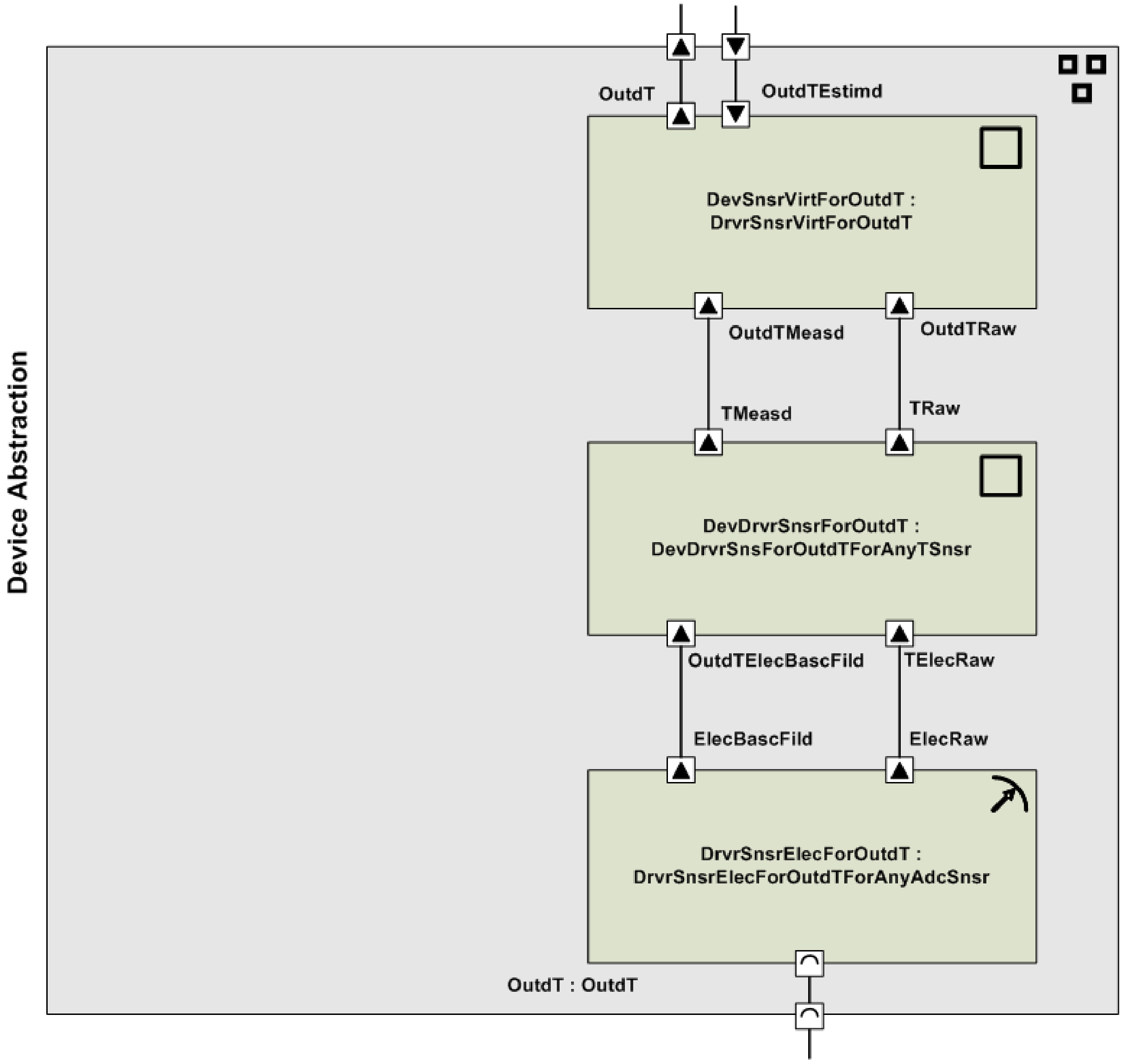

Figure 3.10: Device Abstraction for a Sensor measuring the Environment Temperature

3.6.7 Distributing Device Abstraction

In Figure 3.12 the ECU view derived from the VFB view of a temperature sensor as shown in Figure 3.11 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.

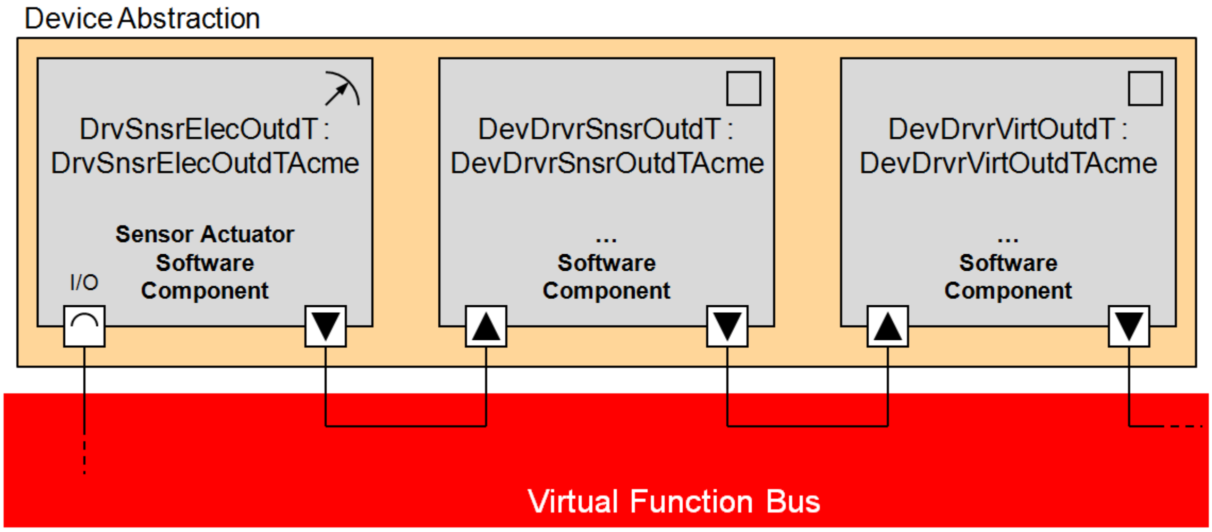

Figure 3.11: VFB View of Temperature Sensor Example

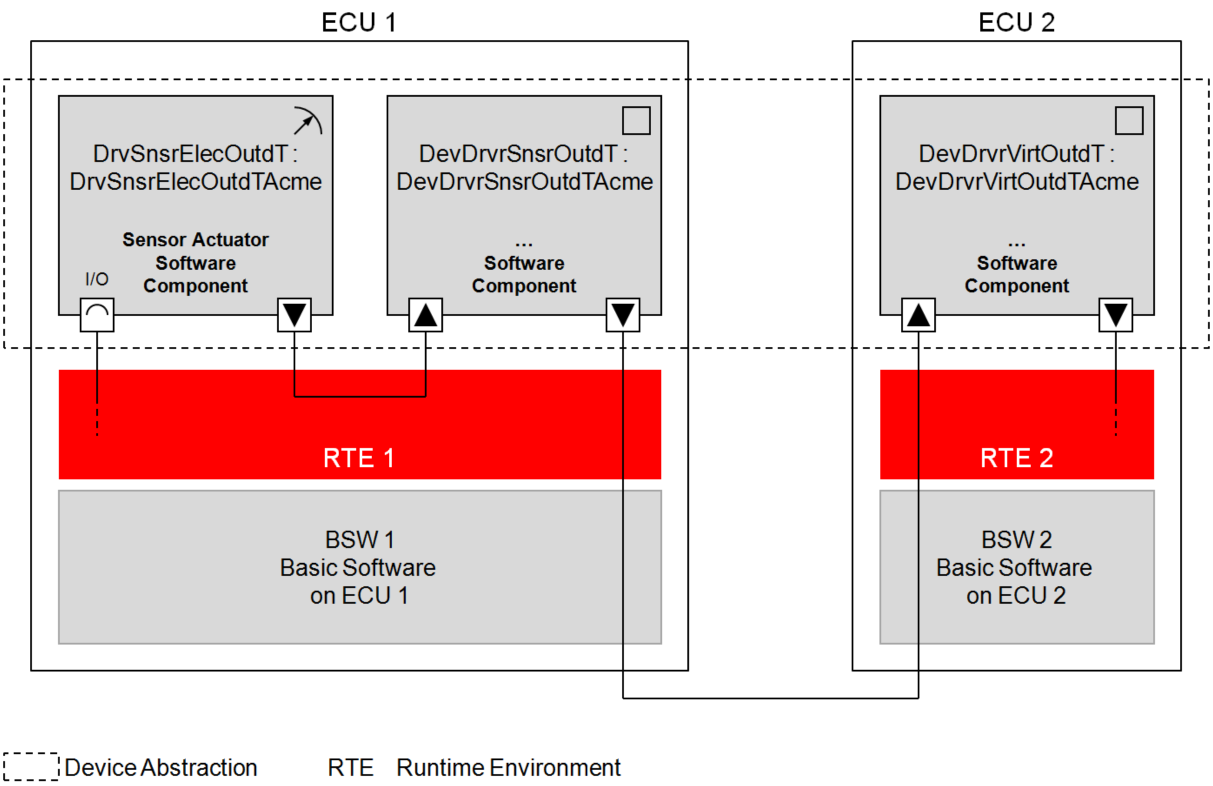

Figure 3.12: ECU Views after Distribution of SW-Cs of Temperature Sensor to two ECUs

3.7 Sample Code and Model

In Listing 3.6 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, [12]. 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.

Listing 3.6: Sensor/Actuator Pattern

```
<AR-PACKAGE>
  <SHORT-NAME>SwComponentTypes_Blueprint</SHORT-NAME>
  <CATEGORY>BLUEPRINT</CATEGORY>
  <REFERENCE-BASES>
    <REFERENCE-BASE>
      <SHORT-LABEL NAME-PATTERN="{anyName}">HwDescriptionEntitys</SHORT
         -LABEL>
      <IS-DEFAULT>false</IS-DEFAULT>
      <IS-GLOBAL>false</IS-GLOBAL>
      <BASE-IS-THIS-PACKAGE>false</BASE-IS-THIS-PACKAGE>
      <PACKAGE-REF DEST="AR-PACKAGE"><?xm-replace_text {PACKAGE-REF}?><
         /PACKAGE-REF><!--add
        package path -->
    </REFERENCE-BASE>
    <REFERENCE-BASE>
      <SHORT-LABEL NAME-PATTERN="{anyName}">PortInterfaces_Blueprint</
         SHORT-LABEL>
      <IS-DEFAULT>false</IS-DEFAULT>
      <IS-GLOBAL>false</IS-GLOBAL>
      <BASE-IS-THIS-PACKAGE>false</BASE-IS-THIS-PACKAGE>
      <PACKAGE-REF DEST="AR-PACKAGE"><?xm-replace_text {PACKAGE-REF}?><
         /PACKAGE-REF><!--add
        package path -->
    </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_Blueprint">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_Blueprint">ElecBascFild1</PROVIDED-
              INTERFACE-TREF>
        </P-PORT-PROTOTYPE>
      </PORTS>
      <!-- add correct reference to sensor actuator type -->
      <SENSOR-ACTUATOR-REF DEST="HW-DESCRIPTION-ENTITY" BASE="
         HwDescriptionEntitys">SensorActuatorType</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>
  <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/SensorActuator</HW-CATEGORY-REF>
      </HW-CATEGORY-REFS>
    </HW-TYPE>
  </ELEMENTS>
</AR-PACKAGE>
<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/
          SensorActuatorType</HW-TYPE-REF>
      </HW-ELEMENT>
    </ELEMENTS>
  </AR-PACKAGE>
```

The HwCategorys should be provided centrally because they are standardized. Definition of HwCategory "SensorActuator" is shown in Listing 3.7.

Listing 3.7: HW Categories as used in Sensor/Actuator Pattern

```
<AR-PACKAGE>
  <SHORT-NAME>HwCategorys_Blueprint</SHORT-NAME>
  <CATEGORY>BLUEPRINT</CATEGORY>
  <ELEMENTS>
    <HW-CATEGORY>
      <SHORT-NAME NAME-PATTERN="blueprintName">SensorActuator</SHORT-
        NAME>
    </HW-CATEGORY>
```

3.8 Known Uses

None.

3.9 Related Patterns

Pattern	Description
Layering Pattern	The layered architecture of AUTOSAR specifies the allowed relationships between different types of software components and basic software modules [12]. It is the basis for the sensor/actuator pattern as well. The layering pattern itself was applied within the pattern to separate concerns, especially with respect to hardware dependency.
Arbitration Pattern	The sensor/actuator pattern is typically combined with the arbitration pattern to allow several set point requesters and not only one like assumed in the sensor/actuator pattern.
Diagnosis Pattern	The sensor/actuator pattern is typically combined with the diagnosis pattern because diagnosis of the sensors or actuators, resp. is essential part of device abstraction.

Table 3.6: Related Patterns

3.10 Anti-Patterns One Should be Aware of

None.

3.11 Further Readings

More information could be found in [12] and [13].

4 Arbitration between Several Set-point Requester

Classification Design Pattern

4.1 Problem

Arbitration between several different setpoint requesters.

4.2 Applicability

This pattern can be applied in the context of Sensor/Actuator Design Pattern. The number of requesters has to be known at pre-compile time. The number of requesters has to be known at implementation or generation time of the arbiter component.

4.3 Solution

A new component for managing all requests from different setpoint requesters is introduced. In Figure 4.2 the overall pattern 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. 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.

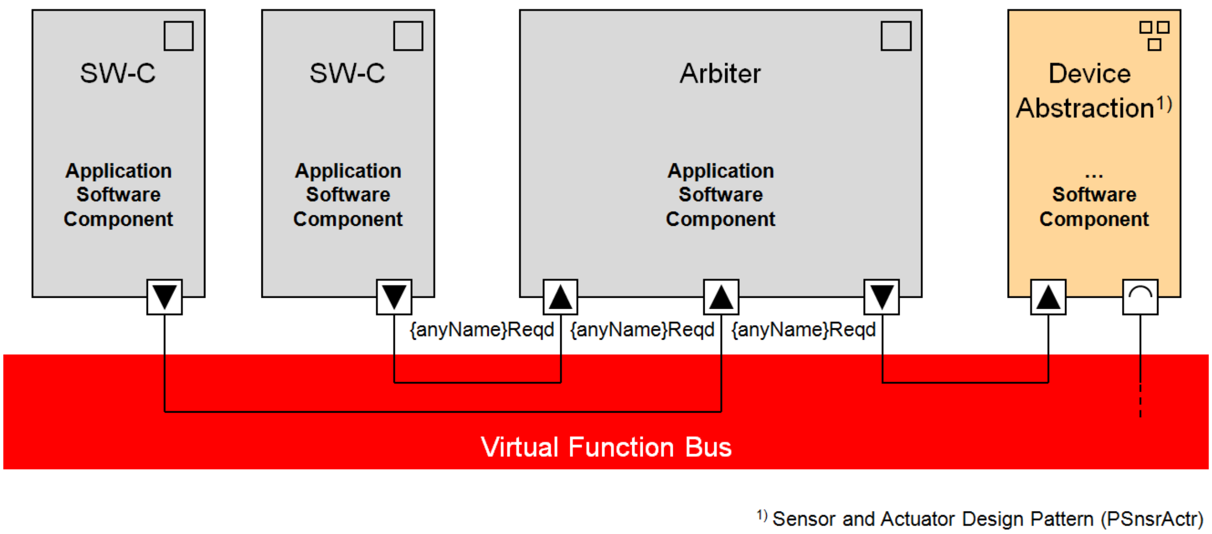

Figure 4.1: Pattern "Arbitration between Several Set-point Requester"

Figure 4.2 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.

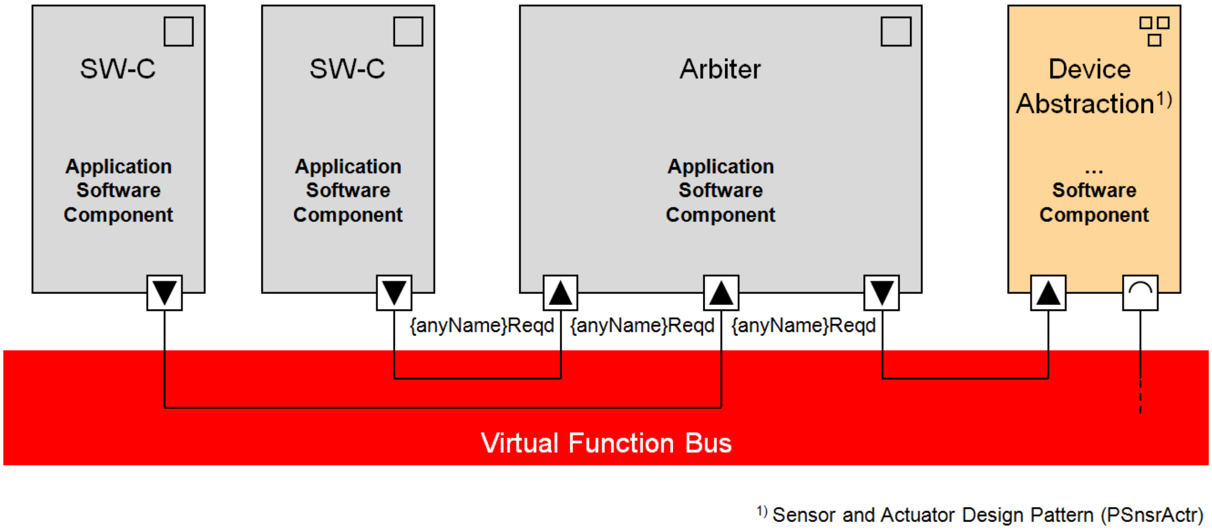

Figure 4.2: Arbitration between Several Set-point Requesters via RTE

4.4 Naming

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 prefix ”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.

Listing 4.1: Name Pattern for Ports of Arbiter and Requesters

```
grammar PArbSpReqPortNames;

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

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

4.5 Example

None.

4.6 Sample Code and Model

None.

4.7 Known Uses

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

4.8 Related Patterns

Pattern	Description
Sensor Actuator Pattern	The sensor/actuator pattern is typically combined with the arbitration pattern to allow several set point requesters and not only one like assumed in the sensor/actuator pattern.
Manager Pattern	A manager component manages respectively controls a set of entities of the same type.
Coordinator Pattern	A coordinator component also allows arbitration between several different set-point requesters but the number of requesters is not fixed at implementation time of the arbiter/coordinator.

Table 4.1: Related Patterns

A History of Constraints and Specification Items

A.1 Constraint History of this Document related to AUTOSAR R4.2.2

A.1.1 Changed Constraints in R4.2.2

No constraints were changed in this release.

A.1.2 Added Constraints in R4.2.2

No constraints were added in this release.

A.1.3 Deleted Constraints in R4.2.2

No constraints were deleted in this release.

A.1.4 Added Specification Items in R4.2.2

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

Table A.1: Added Specification Items in 4.2.2

A.2 Constraint History of this Document related to AUTOSAR R4.2.1

A.2.1 Added Constraints in R4.2.1

No constraints were added in this initial release.

A.2.2 Added Specification Items in R4.2.1

No specification items were added in this initial release.

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**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	The ApplicationSwComponentType is used to represent the application software. **Tags:** atp.recommendedPackage=SwComponentTypes			
Base	ARElement,ARObject,AtomicSwComponentType,AtpBlueprint,AtpBlueprintable,AtpClassifier,AtpType,CollectableElement,Identifiable,MultilanguageReferrable,PackageableElement,Referrable,SwComponentType			
Attribute	***Datatype***	***Mul.***	***Kind***	***Note***
–	–	–	–	–

Table B.1: ApplicationSwComponentType

Class	**CompositionSwComponentType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Note	A CompositionSwComponentType aggregates SwComponentPrototypes (that in turn are typed by SwComponentTypes) as well as SwConnectors for primarily connecting SwComponentPrototypes among each others and towards the surface of the CompositionSwComponentType. By this means hierarchical structures of software-components can be created. **Tags:** atp.recommendedPackage=SwComponentTypes			
Base	ARElement,ARObject,AtpBlueprint,AtpBlueprintable,AtpClassifier,AtpType,CollectableElement,Identifiable,MultilanguageReferrable,PackageableElement,Referrable,SwComponentType			
Attribute	***Datatype***	***Mul.***	***Kind***	***Note***

Attribute	Datatype	Mul.	Kind	Note
component	SwComponentPrototype	*	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 SwComponentPrototypes is resolved post-build the deselected SwComponentPrototypes are still contained in the ECUs build but the instances are inactive in in that they are not scheduled by the RTE. The aggregation is marked as atpSplitable in order to allow the addition of service components to the ECU extract during the ECU integration. 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:** atpSplitable; atpVariation **Tags:** atp.Splitkey=shortName, variationPoint.shortLabel vh.latestBindingTime=postBuild
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. The aggregation is marked as atpSplitable in order to allow the extension of the ECU extract with AssemblySwConnectors between ApplicationSwComponentTypes and ServiceSwComponentTypes during the ECU integration. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=shortName, variationPoint.shortLabel vh.latestBindingTime=postBuild
constantValueMapping	ConstantSpecificationMappingSet	*	ref	Reference to the ConstantSpecificationMapping to be applied for initValues of PPortComSpecs and RPortComSpec. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=constantValueMapping

Attribute	*Datatype*	*Mul.*	*Kind*	*Note*
dataTypeMapping	DataTypeMappingSet	*	ref	Reference to the DataTypeMapping 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 implementers 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. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=dataTypeMapping
instantiationRTEEventProps	InstantiationRTEEventProps	*	aggr	This allows to define instantiation specific properties for RTE Events, in particular for instance specific scheduling. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=shortLabel, variationPoint.shortLabel vh.latestBindingTime=codeGenerationTime

Table B.2: CompositionSwComponentType

Class	**EcuAbstractionSwComponentType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	The ECUAbstraction is a special AtomicSwComponentType that resides between a software-component that wants to access ECU periphery and the Microcontroller Abstraction. The EcuAbstractionSwComponentType 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,AtpBlueprintable,AtpClassifier,AtpType,CollectableElement,Identifiable,MultilanguageReferrable,PackageableElement,Referrable,SwComponentType			
Attribute	*Datatype*	*Mul.*	*Kind*	*Note*
hardwareElement	HwDescriptionEntity	*	ref	Reference from the EcuAbstractionComponentType to the description of the used HwElements.

Table B.3: EcuAbstractionSwComponentType

Class	**FlatInstanceDescriptor**			
Package	M2::AUTOSARTemplates::CommonStructure::FlatMap			
Note	Represents exactly one node (e.g. a component instance or data element) of the instance tree of a software system. The purpose of this element is to map the various nested representations of this instance to a flat representation and assign a unique name (shortName) to it. Use cases: • Specify unique names of measurable data to be used by MCD tools • Specify unique names of calibration data to be used by MCD tool • Specify a unique name for an instance of a component prototype in the ECU extract of the system description Note that in addition it is possible to assign alias names via AliasNameAssignment.			
Base	ARObject,Identifiable,MultilanguageReferrable,Referrable			
Attribute	***Datatype***	***Mul.***	***Kind***	***Note***
ecuExtract Reference	AtpFeature	0..1	iref	Refers to the instance in the ECU extract. This is valid only, if the FlatMap is used in the context of an ECU extract. The reference shall be such that it uniquely defines the object instance. For example, if a data prototype is declared as a role within an SwcInternalBehavior, it is not enough to state the SwcInternalBehavior as context and the aggregated data prototype as target. In addition, the reference shall also include the complete path identifying instance of the component prototype and the AtomicSoftwareComponentType, which is refered by the particular SwcInternalBehavior. **Tags:** xml.sequenceOffset=40
role	Identifier	0..1	ref	The role denotes the particular role of the downstream memory location described by this FlatInstanceDescriptor. It applies to use case where one upstream object results in multiple downstream objects, e.g. ModeDeclarationGroupPrototypes which are measurable. In this case the RTE will provide locations for current mode, previous mode and next mode.
swDataDef Props	SwDataDefProp s	0..1	aggr	The properties of this FlatInstanceDescriptor.

Attribute	Datatype	Mul.	Kind	Note
upstreamReference	AtpFeature	0..1	iref	Refers to the instance in the context of an "upstream" descriptions, wich could be the system or system extract description, the basic software module description or (if a flat map is used in preliminary context) a description of an atomic component or composition. This reference is optional in case the flat map is used in ECU context. The reference shall be such that it uniquely defines the object instance in the given context. For example, if a data prototype is declared as a role within an SwcInternalBehavior, it is not enough to state the SwcInternalBehavior as context and the aggregated data prototype as target. In addition, the reference shall also include the complete path identifying the instance of the component prototype that contains the particular instance of SwcInternalBehavior. **Tags:** xml.sequenceOffset=20

Table B.4: FlatInstanceDescriptor

Class	**HwCategory**			
Package	M2::AUTOSARTemplates::EcuResourceTemplate::HwElementCategory			
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			
Attribute	**Datatype**	**Mul.**	**Kind**	**Note**
hwAttributeDef	HwAttributeDef	*	aggr	This aggregation describes particular hardware attribute definition.

Table B.5: HwCategory

Class	**HwDescriptionEntity (abstract)**			
Package	M2::AUTOSARTemplates::EcuResourceTemplate			
Note	This meta-class represents the ability to describe a hardware entity.			
Base	ARObject,Referrable			
Attribute	**Datatype**	**Mul.**	**Kind**	**Note**
hwAttributeValue	HwAttributeValue	*	aggr	This aggregation represents a particular hardware attribute value. **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=systemDesignTime xml.sequenceOffset=50

Attribute	Datatype	Mul.	Kind	Note
hwCategory	HwCategory	*	ref	One of the associations representing one particular category of the hardware entity. **Tags:** xml.sequenceOffset=30
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 HwTypes can not be redefined and therefore shall not have a hwType reference.

Table B.6: HwDescriptionEntity

Class	HwElement			
Package	M2::AUTOSARTemplates::EcuResourceTemplate			
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			
Attribute	**Datatype**	**Mul.**	**Kind**	**Note**
hwElementConnection	HwElementConnector	*	aggr	This represents one particular connection between two hardware elements. **Stereotypes:** atpVariation **Tags:** 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:** 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:** vh.latestBindingTime=systemDesignTime xml.sequenceOffset=70

Table B.7: HwElement

Class	**HwType**			
Package	M2::AUTOSARTemplates::EcuResourceTemplate::HwElementCategory			
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			
Base	ARElement,ARObject,CollectableElement,HwDescriptionEntity,Identifiable,MultilanguageReferrable,PackageableElement,Referrable			
Attribute	*Datatype*	*Mul.*	*Kind*	*Note*
–	–	–	–	–

Table B.8: HwType

Primitive	**Identifier**			
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::PrimitiveTypes			
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	*Datatype*	*Mul.*	*Kind*	*Note*
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.9: Identifier

Class	**Keyword**			
Package	M2::AUTOSARTemplates::StandardizationTemplate::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			
Attribute	*Datatype*	*Mul.*	*Kind*	*Note*

Attribute	Datatype	Mul.	Kind	Note
abbrName	NameToken	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.10: Keyword

Class	PortPrototype (abstract)			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
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,AtpBlueprintable,AtpFeature,AtpPrototype,Identifiable,MultilanguageReferrable,Referrable			
Attribute	**Datatype**	**Mul.**	**Kind**	**Note**
clientServerAnnotation	ClientServerAnnotation	*	aggr	Annotation of this PortPrototype with respect to client/server communication.
delegatedPortAnnotation	DelegatedPortAnnotation	0..1	aggr	Annotations on this delegated port.
ioHwAbstractionServerAnnotation	IoHwAbstractionServerAnnotation	*	aggr	Annotations on this IO Hardware Abstraction port.
modePortAnnotation	ModePortAnnotation	*	aggr	Annotations on this mode port.
nvDataPortAnnotation	NvDataPortAnnotation	*	aggr	Annotations on this non voilatile data port.
parameterPortAnnotation	ParameterPortAnnotation	*	aggr	Annotations on this parameter port.
senderReceiverAnnotation	SenderReceiverAnnotation	*	aggr	Collection of annotations of this ports sender/receiver communication.
triggerPortAnnotation	TriggerPortAnnotation	*	aggr	Annotations on this trigger port.

Table B.11: PortPrototype

Class	**PortPrototypeBlueprint**			
Package	M2::AUTOSARTemplates::StandardizationTemplate::BlueprintDedicated::Port ProtoypeBlueprint			
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 PortInterfaces. **Tags:** atp.recommendedPackage=PortPrototypeBlueprints			
Base	ARElement,ARObject,AtpBlueprint,AtpClassifier,AtpFeature,AtpStructure Element,CollectableElement,Identifiable,MultilanguageReferrable,Packageable Element,Referrable			
Attribute	**Datatype**	**Mul.**	**Kind**	**Note**
initValue	PortPrototypeBl ueprintInitValue	*	aggr	This specifies the init values for the dataElements in the particular PortPrototypeBlueprint.
interface	PortInterface	1	ref	This is the interface for which the blueprint is defined. It may be a blueprint itself or a standardized PortInterface
providedC omSpec	PPortComSpec	*	aggr	Provided communication attributes per interface element (data element or operation).
requiredCo mSpec	RPortComSpec	*	aggr	Required communication attributes, one for each interface element.

Table B.12: PortPrototypeBlueprint

Class	**SensorActuatorSwComponentType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
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,AtpBlueprintable,Atp Classifier,AtpType,CollectableElement,Identifiable,Multilanguage Referrable,PackageableElement,Referrable,SwComponentType			
Attribute	**Datatype**	**Mul.**	**Kind**	**Note**
sensorActu ator	HwDescriptionE ntity	1	ref	Reference from the Sensor Actuator Software Component Type to the description of the actual hardware.

Table B.13: SensorActuatorSwComponentType

Class	**SwComponentPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Note	Role of a software component within a composition.			
Base	ARObject,AtpFeature,AtpPrototype,Identifiable,MultilanguageReferrable,Referrable			
Attribute	**Datatype**	**Mul.**	**Kind**	**Note**

Attribute	Datatype	Mul.	Kind	Note
type	SwComponentType	1	tref	Type of the instance. **Stereotypes:** isOfType

Table B.14: SwComponentPrototype

Class	**SwComponentType (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	Base class for AUTOSAR software components.			
Base	ARElement,ARObject,AtpBlueprint,AtpBlueprintable,AtpClassifier,AtpType,CollectableElement,Identifiable,MultilanguageReferrable,PackageableElement,Referrable			
Attribute	**Datatype**	**Mul.**	**Kind**	**Note**
consistencyNeeds	ConsistencyNeeds	*	aggr	This represents the colelction of ConsistencyNeeds owned by the enclosing SwComponentType. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=shortName, variationPoint.shortLabel vh.latestBindingTime=preCompileTime
port	PortPrototype	*	aggr	The ports through which this component can communicate. The aggregation of PortPrototype is subject to variability with the purpose to support the conditional existence of PortPrototypes. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=shortName, variationPoint.shortLabel vh.latestBindingTime=preCompileTime
portGroup	PortGroup	*	aggr	A port group being part of this component. **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=preCompileTime
swComponentDocumentation	SwComponentDocumentation	0..1	aggr	This adds a documentation to the SwComponentType. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=swComponentDocumentation, variationPoint.shortLabel vh.latestBindingTime=preCompileTime xml.sequenceOffset=-10
unitGroup	UnitGroup	*	ref	This allows for the specification of which UnitGroups are relevant in the context of referencing SwComponentType.

Table B.15: SwComponentType