Document Title	Software Component Template
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	062
Document Classification	Standard

Document Version	3.2.0
Document Status	Final
Part of Release	3.0
Revision	7

Document Change History

Date	Version	Changed by	Description
2010-09-02	3.2.0	AUTOSAR Administration	• Fixed usage of Categories in XML examples • Signal invalidation mechanism becomes optional
2010-01-26	3.1.0	AUTOSAR Administration	• Allow for communication attributes in CompositionTypes • Allow for providing initial values for calibration parameters
2007-11-13	3.0.0	AUTOSAR Administration	• Improved support for measurement and calibration • Improved semantics of delegation ports • Introduction of abstract memory classes • Document meta information extended • Small layout adaptations made

2007-01-31	2.1.0	AUTOSAR Administration	• Harmonization of the document with other specifications (e.g. RTE) • Introduction of a new concept to support calibration and measurement - harmonized with RTE • Description of needs of the Software Component Template toward AUTOSAR services and of the interaction of the Software Component Template and services (on XML level) • Legal disclaimer revised • Release notes added • ”‘Advice for users”’ added • ”‘Revision information”’ added
2006-05-18	2.0.0	AUTOSAR Administration	Second
2005-05-09	1.0.0	AUTOSAR Administration	Initial release

Disclaimer

The AUTOSAR specifications have been developed for automotive applications only. They have neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Advice for users

AUTOSAR Specification Documents 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 Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, 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

Bibliography

[1] AUTOSAR RTE Software Specification
AUTOSAR_SWS_RTE.pdf

[2] Requirements on Basic Software: Layered Software Architecture
AUTOSAR_LayeredSoftwareArchitecture.pdf

[3] Specification of the Virtual Functional Bus
AUTOSAR_Spec_of_VFB.pdf

[4] Methodology
AUTOSAR_Methodology.pdf

[5] Specification of Interoperability of Authoring Tools
AUTOSAR_InteroperabilityAuthoringTools.pdf

[6] Template UML Profile and Modeling Guide
AUTOSAR_TemplateModelingGuide.pdf

[7] Model Persistence Rules for XML
AUTOSAR_ModelPersistenceRulesXML.pdf

[8] Specification of the BSW Module Description Template
AUTOSAR_BSWMDTemplate.pdf

[9] Design Specification for the ECU Resource Template
AUTOSAR_ResourceTemplateECU.pdf

[10] System Template
AUTOSAR_SystemTemplate.pdf

[11] AUTOSAR Template Modeling Patterns
AUTOSAR_TemplateModelingPatterns.pdf

[12] Specification of Graphical Notation
AUTOSAR_GraphicalNotation.pdf

[13] Specification of IO Hardware Abstraction
AUTOSAR_SRS_IOHW_Abstraction.pdf

[14] Specification of Communication
AUTOSAR_SWS_COM.pdf

[15] Specification of Module Operating System
AUTOSAR_SWS_OS.pdf

[16] Specification of ECU Configuration Parameters
AUTOSAR_ECU_ConfigationParameters.pdf

[17] Specification of NVRAM Manager
AUTOSAR_SWS_NVRAMManager.pdf

[18] Specification of Module Watchdog Manager
AUTOSAR_SWS_WatchdogManager.pdf

[19] Specification of Communication Manager
AUTOSAR_SWS_ComManager.pdf

[20] Specification of ECU State Manager
AUTOSAR_SWS_ECU_StateManager.pdf

[21] Specification of Module DEM
AUTOSAR_SWS_DEM.pdf

[22] Specification of Module FIM
AUTOSAR_SWS_FIM.pdf

[23] Specification of Module DCM
AUTOSAR_SWS_DCM.pdf

[24] Glossary
AUTOSAR_Glossary.pdf

1 Introduction

1.1 Overview

This document contains the specification of the AUTOSAR `Software-Component Template`. Actually, it has been created as a supplement to the formal definition of the `Software-Component Template` by means of the AUTOSAR meta-model. In other words, this document in addition to the formal specification provides introductory description and rationale for the part of the AUTOSAR meta-model relevant for the definition of software-components.

Nevertheless, the core part of the specification is directly based on the content of the AUTOSAR meta-model. Therefore, this document contains a summary of the main concepts of the AUTOSAR meta-model, see chapters 1.2 and 1.4.

In this context, the term software-component refers to a formally described piece of software existing above the AUTOSAR RTE [1]. In other words, this document emphasizes on application software as opposed to standard basic software modules existing in an AUTOSAR ECU [2].

Please note that the general ideas behind the semantics of application software-components have been described in the specification of the `Virtual Functional Bus` [3]. The latter, however, represents conceptual work that strongly influences but does not totally govern the formal definition of software-components.

Note further that this document does not provide any ”best practice” recommendations of software-component modeling nor does it require or enforce a certain methodology. Note however, that the methodology aspect is covered by the specification of the AUTOSAR methodology [4].

Although it is beyond any doubt reasonable to use a suitable AUTOSAR Authoring Tool for dealing with AUTOSAR software-components, this specification does not make any assumptions nor does it give recommendations regarding the tooling. Please refer to [5] for more details about AUTOSAR Authoring Tools are supposed to work and interact.

1.2 Methodology for Defining Formal Template

Figure 1.1 illustrates the overall methodology used to define formal templates. As explained in [6], it is important to separate a precise and concise model of the information that needs to be captured from the concrete XML-Schemas or other technology that is used to define the actual templates.

The following documents describe the various aspects of the methodology:

1. The document called `Software Component Template` (i.e. this document) describes the information that can be captured in the description of software-

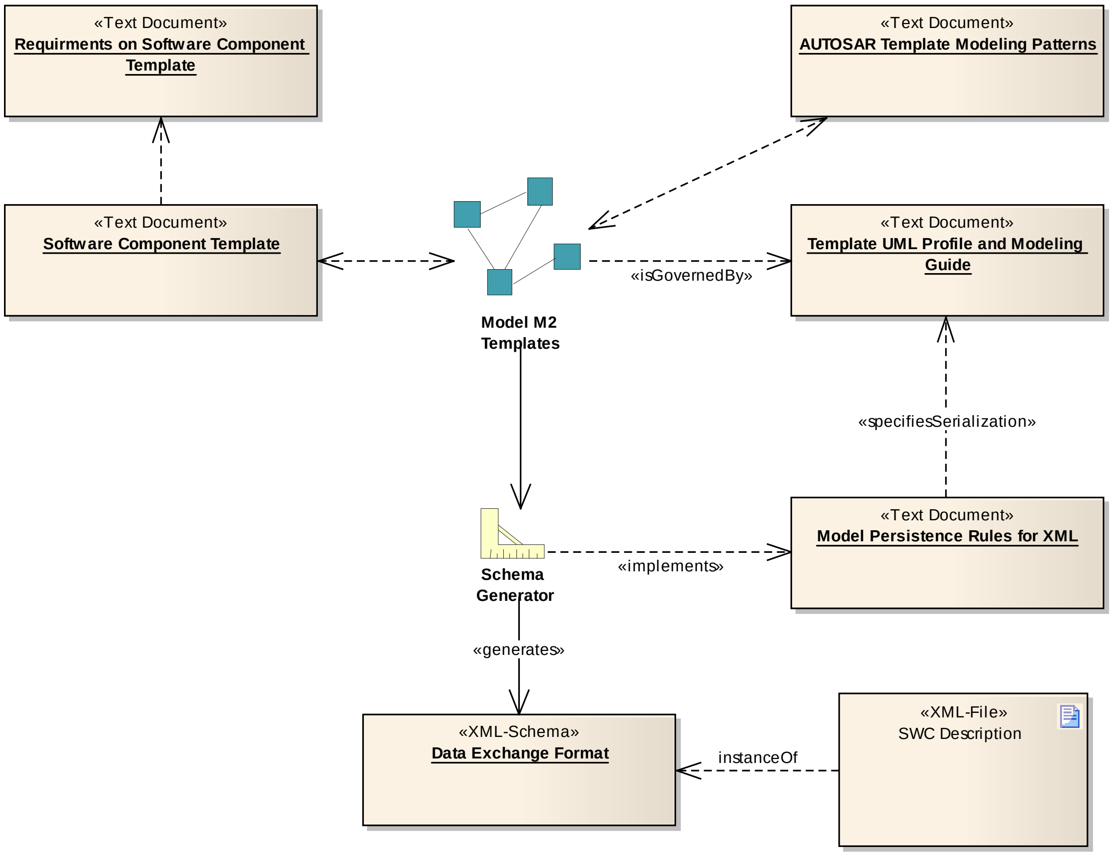


Figure 1.1: Methodology to define templates in AUTOSAR

component, independently from the mapping of this model on XML-technology. This document is based upon the AUTOSAR meta-model and contains an elaborate description of the semantics (the precise meaning) of all the information that can be captured within the relevant parts of this meta-model.

2. The *Template UML Profile and Modeling Guide* [6] describes the basic concepts that should be used when creating content of the meta-model.
3. The document called *Model Persistence Rules for XML* [7] describes how XML is used and how the meta-model designed in the ”Software Component Template” should be translated by the ”Schema Generator” (MDS) into XML-Schema (XSD) ”Data Exchange Format”.
   This ”formalization strategy” is supposed to be used for all data that is formally described in the meta-model. In particular this document is worth to read in order to understand the mapping of the meta-model and the XML based Software component template.
4. The ”AUTOSAR Template Modeling Patterns” are represented as predefined Classes in the meta-model which are incorporated in the generated schema. Examples for such patterns are the ”common attributes” which are added to each generated class even if not explicitly inherited in the meta-model.

5. The concrete ”Template” is an XML schema automatically generated out of the meta-model described in the `Software Component Template` using the approach and the patterns defined in the ”Model Persistence Rules for XML”. This schema is typically used as input to AUTOSAR tools.
   The M1-level [6] software component descriptions are XML files that can be validated against the XML schema. In other words, the XML files are instances of the schema defining the XML representation of the template. Note that the concrete XML Schema file might also cover aspects of the meta model that are not relevant for the description of software-components.

In figure 1.2 the relationship between the AUTOSAR templates and their associated template specification documents is illustrated.

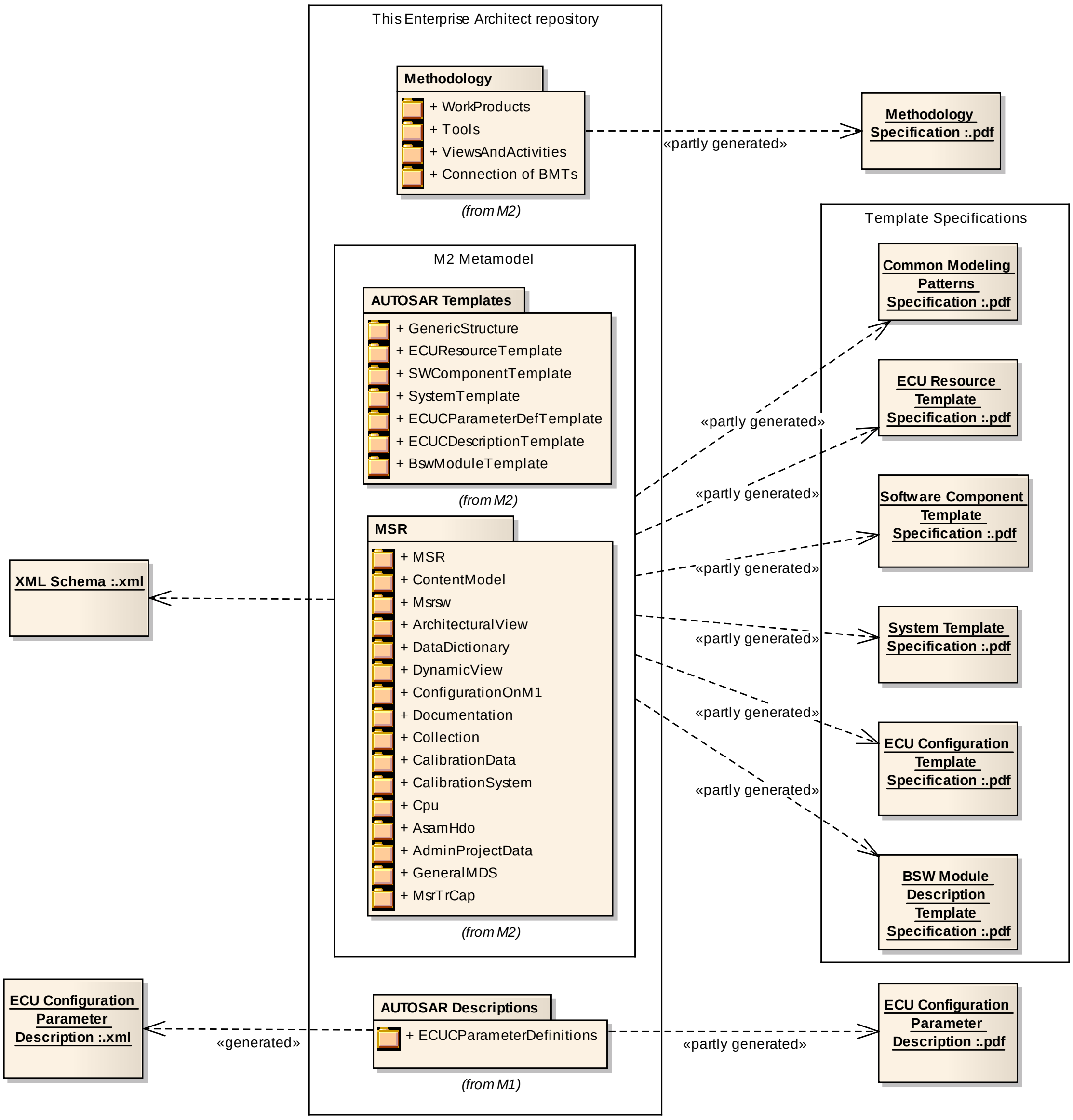


Figure 1.2: Structure and Dependencies of AUTOSAR Templates

1.3 Scope

As already mentioned in chapter 1.1, the Scope of this document is the description of AUTOSAR software-components. This work covers the following three aspects:

- A general description of `ComponentTypes` using `PortPrototypes` and `PortInterfaces`, i.e. this document defines the `ComponentType` as an entity which can be described through `PortPrototypes` which provide or require `PortInterfaces`.
- A description of `CompositionTypes`, which are sub-systems consisting out of connected instances of software-components, i.e. software-components may be defined in the form of hierarchical subsystems, which in turn consist of software-components again. The description of such hierarchical structures is in scope of this document.
- A description of `AtomicSoftwareComponentType` which is implemented as a piece of software that can be mapped to an AUTOSAR ECU.
  An `AtomicSoftwareComponentType` therefore shows up in the ECU Software Architecture depicted in Figure 1.3. In this figure, the green (vertically striped) and blue (diagonally striped) borders show the aspects that are described by the `Software-Component Template`.

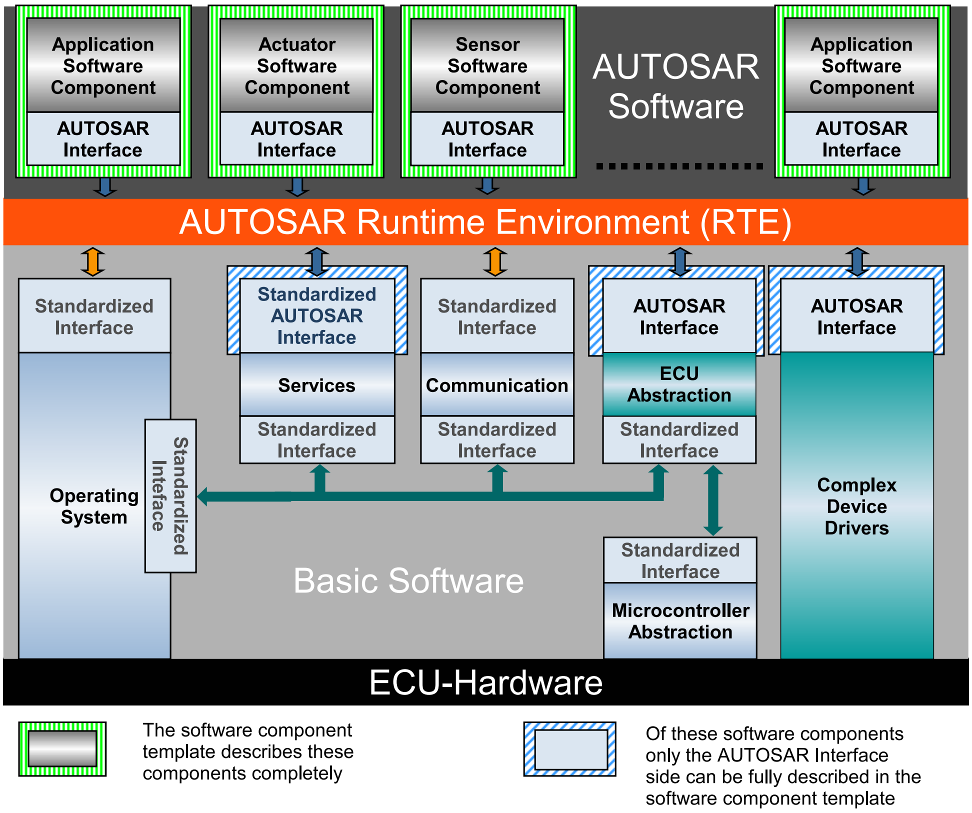


Figure 1.3: Scope of this document in the ECU SW Architecture [2]

Aspects of AUTOSAR Basic Software not relevant for the RTE are out of scope; these are covered by the `Basic Software Module Description Template` [8].

1.4 Organization of the Meta-Model

Figure 1.4 sketches the overall structure of the meta-model, which formally defines the vocabulary required to describe AUTOSAR software-components. As the diagram points out, other template specifications (e.g. `ECU Resource Template` [9] and `System Template` [10]) also use the same modeling approach in order to define an overall consistent model of AUTOSAR software description.

The dashed arrows in the diagram describe dependencies in terms of import-relationships between the packages within the meta-model. For example, the package `SWComponentTemplate` imports meta-classes defined in the packages `GenericStructure` [11] and `ECUResourceTemplate` [9].

Please note that this specification document will only discuss meta-model elements defined in the package `SWComponentTemplate`.

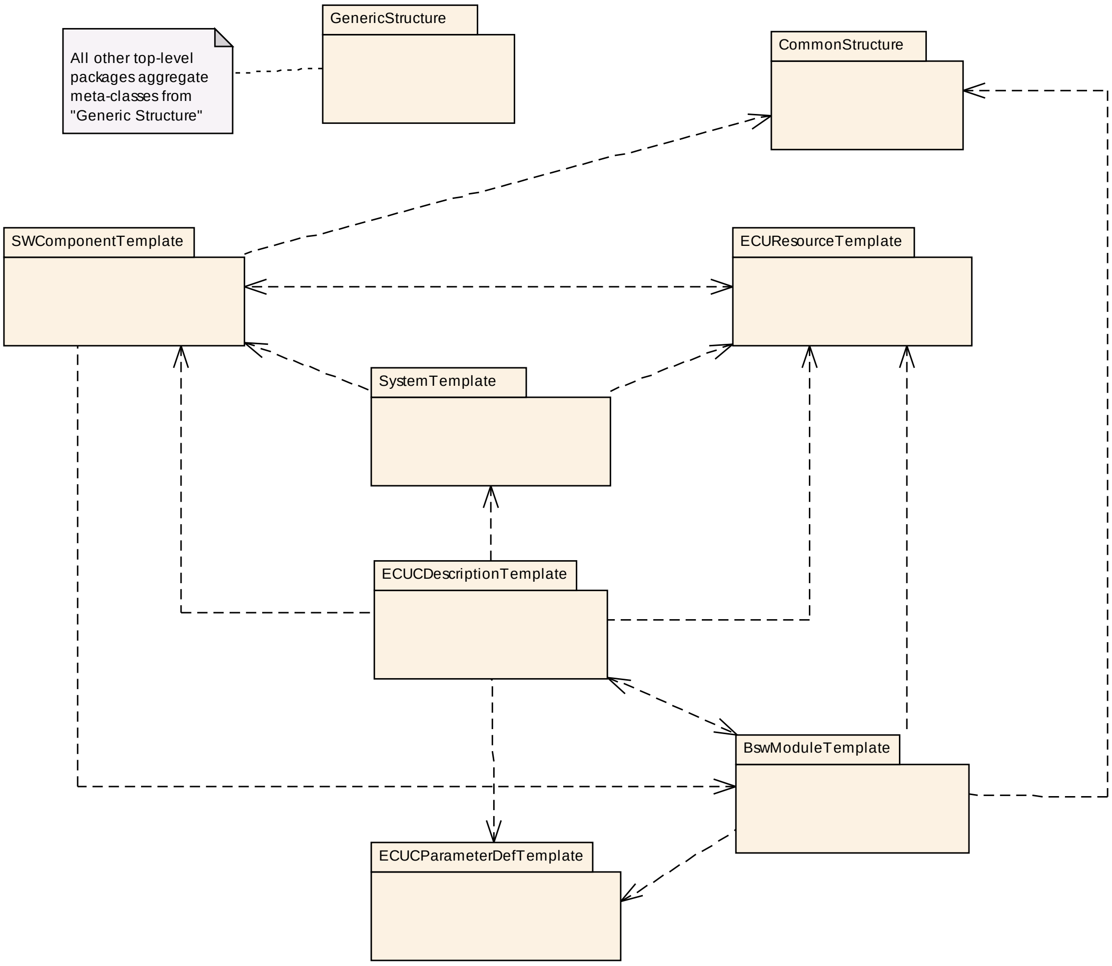


Figure 1.4: Structure of the meta-model

For clarification, please note that the package `GenericStructure` contains some fundamental infrastructure meta-classes and common patterns that are described in [11]. As these are used by all other template specification the dependency associations are not depicted in the diagram for the sake of clarity.

1.5 Structure of the Template

AUTOSAR software components are described on three distinctive levels, as shown in Figure 1.5.

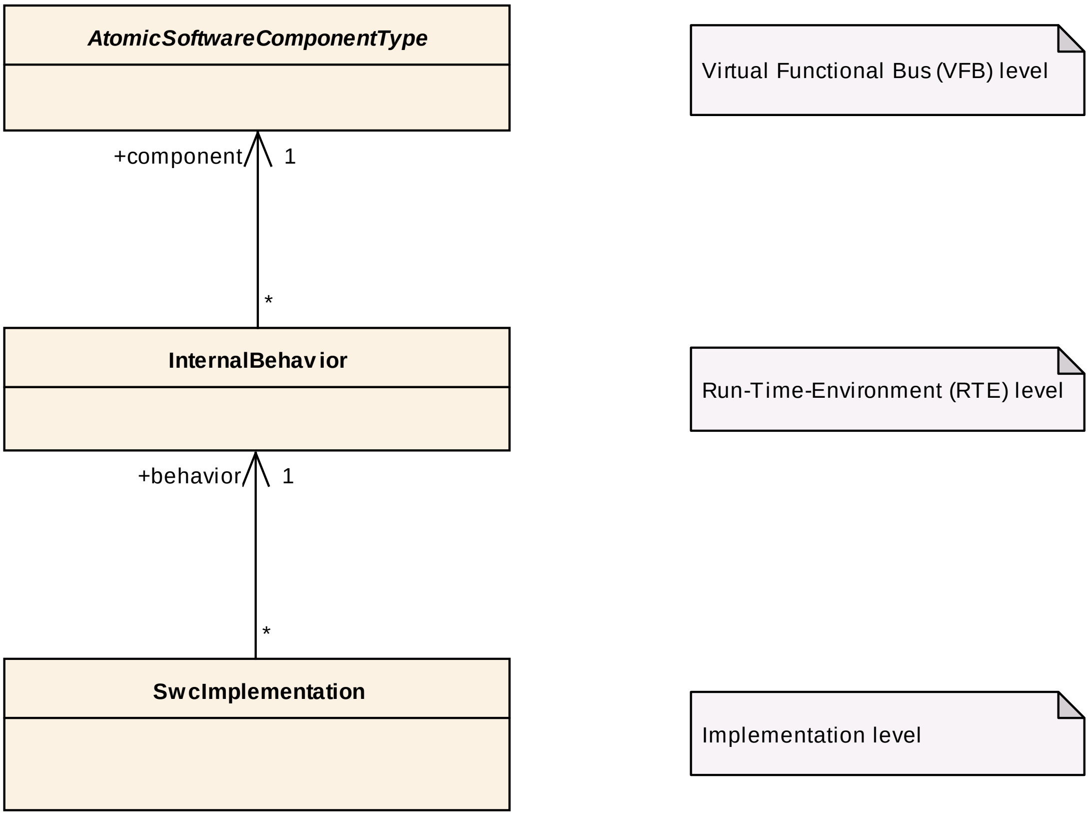


Figure 1.5: The description of a software component is done on three levels

1.5.1 Description of software-components on VFB level

The highest (most abstract) description level is the `Virtual Functional Bus` [3]. In this document `ComponentTypes` are described with the means of `DataTypes`, `PortInterfaces`, `PortPrototypes`, and connections between them. At this level, the fundamental communication properties of components and their communication relationships among each other are expressed.

In the diagram depicted in figure 1.5, this aspect is expressed by means of the description of `AtomicSoftwareComponentType`[1].

---

[1] To avoid clutter and require additional up-front information about the meta model, compositions have not been added to the diagram.

1.5.2 Description of software-components on RTE level

The middle level allows for behavior description of a given `AtomicSoftwareComponentType`. This so-called `InternalBehavior` is expressed according to AUTOSAR RTE concepts, e.g. `RTEEvents` and in terms of schedulable units, so-called `RunnableEntities`.

For instance, for an `OperationPrototype` defined in the scope of a `ClientServerInterface` on the VFB, the behavior specifies which `RunnableEntity` is activated as a consequence of the invocation of the specific `OperationPrototype`. As sketched by 1.5, there may be multiple `InternalBehaviors` referencing a given `AtomicSoftwareComponentType`.

1.5.3 Descriptions of software-components on implementation level

The lowest (most concrete) level of description specifies the implementation (i.e. in terms of the AUTOSAR meta-model: the `Implementation`) of a given `InternalBehavior` description. More precisely, the `RunnableEntities` of such a behavior are mapped to code (source code or object code).

There may be different `Implementations` that reference a specific `InternalBehavior` description, e.g. in different programming languages, or with differently optimized code.

Please note that `Implementation` has been described in previous versions of this document. In response to the evolution of the AUTOSAR concept the description of the `Implementation` aspect has been moved to the ”GenericStructure” (see figure 1.4) because it is also used for creating the `Basic Software Module Description Template` [8].

1.6 Document Conventions

Technical terms are typeset in monospaced font, e.g. `PortPrototype`.

2 Overview: Software Components, Ports, and Interfaces

2.1 Introduction

The detailed introduction of all aspects of the software component template in one move is considered too complex. This chapter therefore provides an overview of the main conceptual aspects of software components, ports and interfaces. The overview will then be broken down into further details in chapter 3.

One of the goals of the AUTOSAR concept is the support of re-usability on the level of application software. In other words: it should be possible to re-use existing artifacts to create further model elements instead of being forced to create every single modeling detail from scratch. One of the consequences of this approach is the application of the so-called type-prototype pattern [6].

Among other things, this concept allows for creating hierarchical structures of software-components with arbitrary complexity. However, the creation of hierarchical structures itself does not have an impact on the run-time behavior of the overall system. The actual behavior is completely defined within the individual software-components.

This conclusion is backed by the understanding that software-components are developed against the so-called *Virtual Functional Bus* (VFB), an abstract communication channel without direct dependency on ECUs and communication buses. The VFB does not provide any means for expressing a hierarchy of software-components.

Of course, the usage of the VFB has further consequences on the design of software-components, which must not directly call the operating system or the communication hardware. As a result, software-components can be deployed to actual ECUs at a rather late stage in the development process.

In order to make the description more precise, the following text preferably uses accurate meta-model terms instead of the rather vague terminology of ”composition” and ”software-component”.

2.2 Software Component

Application software within AUTOSAR is organized in self-contained units called `AtomicSoftwareComponentTypes`. Such `AtomicSoftwareComponentTypes` encapsulate the implementation of their functionality and behavior and merely expose well-defined connection points, called `PortPrototypes`, to the outside world.

The graphical appearance of AUTOSAR software-components according to [12] is depicted in figure 2.1.

Class	⟨⟨**atpType**⟩⟩ **ComponentType (abstract)**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components

Class Desc.	Base class for AUTOSAR software components.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
port	PortPrototype	*	aggregation	The ports through which this component can communicate.

Table 2.1: ComponentType

`AtomicSoftwareComponentTypes` (and also the more general `ComponentTypes` may only interact by means of their `PortPrototypes`). Hidden dependencies that are *not* expressed by means of `PortPrototypes` are not allowed. Therefore, software-components are in theory exchangeable as long as they implement the same functionality and provide the same public communication interface to the remaining system.

As mentioned before, the term `AtomicSoftwareComponentType` is a specific form of the general concept of the `ComponentType`. The latter contributes the concept for interaction, mainly in form of `PortPrototypes`.

Class	**⟨⟨atpType⟩⟩ AtomicSoftwareComponentType (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Desc.	An atomic software component is atomic in the sense that it cannot be further decomposed and distributed across multiple ECUs.			
Base Class(es)	ComponentType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 2.2: AtomicSoftwareComponentType

There are several specialized `ComponentTypes` to describe specific software-components used in the different parts of the AUTOSAR Layered Architecture [2]. Further details are mentioned in chapter 9 and 10.

The `ApplicationSoftwareComponentType` is a specific class of `AtomicSoftwareComponentType` for representing hardware-independent application software.

Class	**⟨⟨atpType⟩⟩ ApplicationSoftwareComponentType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Desc.	The ApplicationSoftwareComponentType is used to represent the application software.			
Base Class(es)	AtomicSoftwareComponentType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 2.3: ApplicationSoftwareComponentType

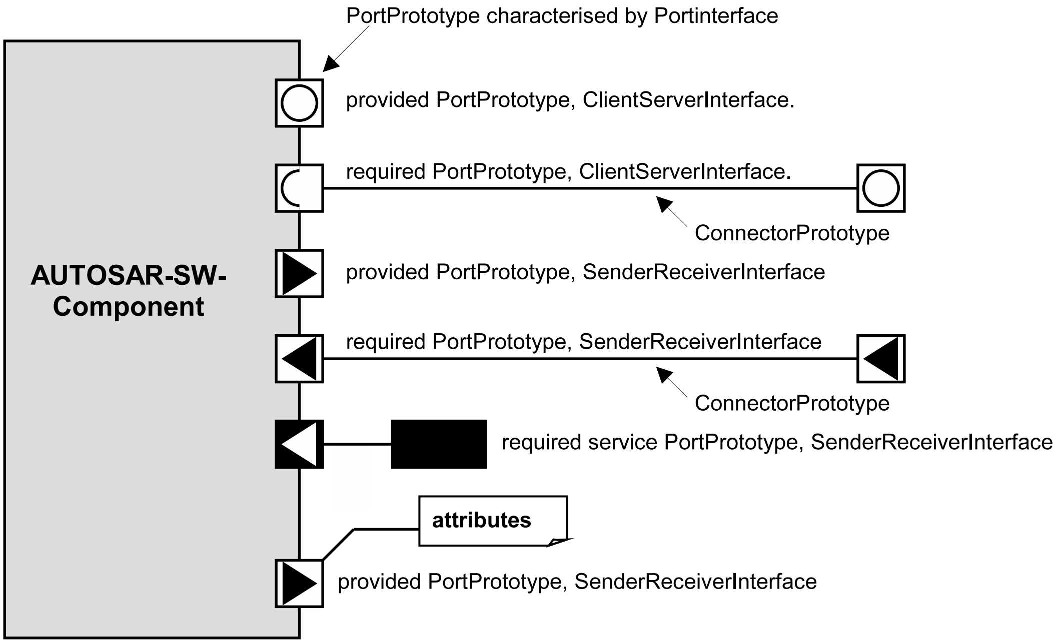


Figure 2.1: Graphical representation of software-components in AUTOSAR

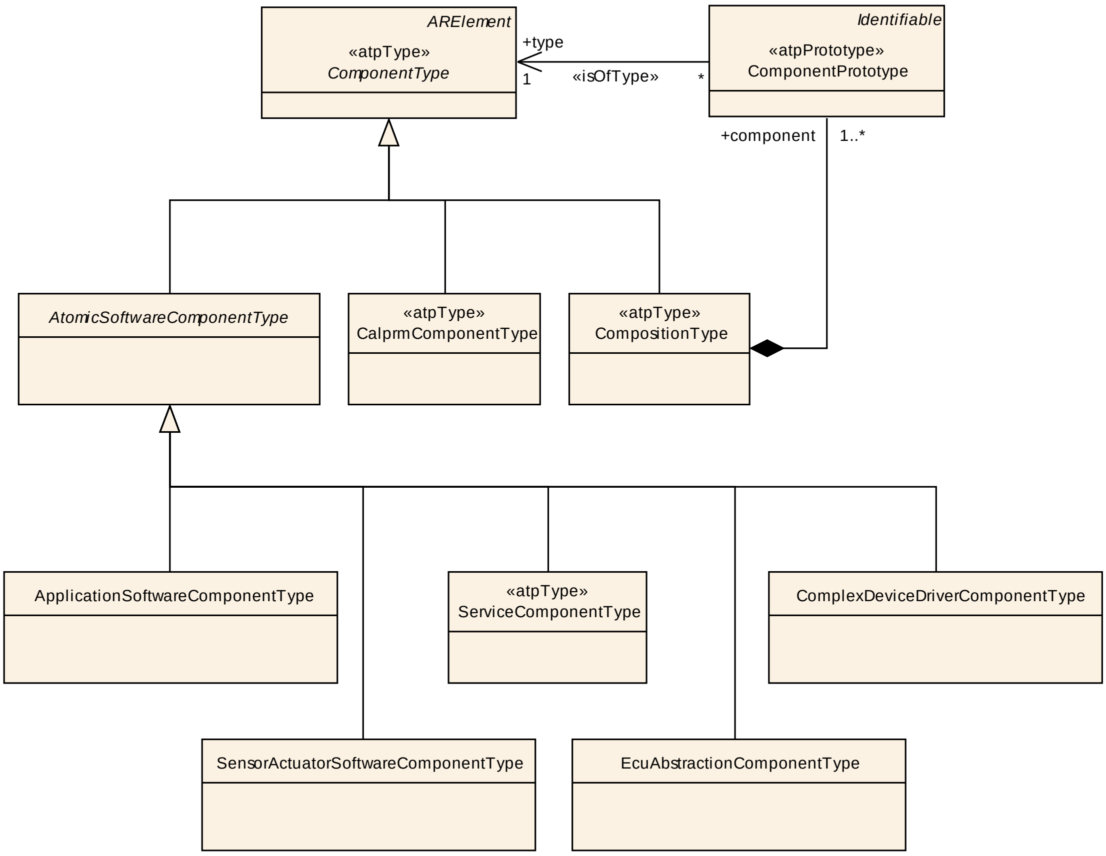


Figure 2.2: Overview of Component Types

More specifically, the `PortPrototypes` of a `ComponentType` can be used for attaching `ConnectorPrototypes` that establish an actual connection between `ComponentPrototypes` (see chapter 2.3).

Class	**⟨⟨atpPrototype⟩⟩ PortPrototype (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Desc.	Base class for the ports of an AUTOSAR software component.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
calibrationPortAnnotation	CalibrationPortAnnotation	*	aggregation	Annotations on this CalibrationPort.
delegatedPortAnnotation	DelegatedPortAnnotation	0..1	aggregation	
ioHwAbstraction Server Annotation	IoHwAbstraction Server Annotation	*	aggregation	
sender Receiver Annotation	Sender Receiver Annotation	*	aggregation	Collection of annotations of this ports sender/receiver communication.

Table 2.4: PortPrototype

Please note that `PortPrototypes` actually needs an additional model artifact, the `PortInterface` for fully describing the details of the `PortPrototype`. The concept of the `PortInterface` as another means for establishing a high degree of re-usability is described in chapter 2.4.

As depicted in figure 2.3, ports are either *require*- or *provide*-ports. A require-port (in technical terms: `RPortPrototype`) requires certain services or data, while a provide-port (or `PPortPrototype`) on the other hand provides those services or data. Two `ComponentPrototypes` are eventually connected by hooking up a `PPortPrototype` of one `ComponentPrototype` to a compatible `RPortPrototype` of the other `ComponentPrototypes`.

Class	⟨⟨**atpPrototype**⟩⟩ **RPortPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Desc.	Component port requiring a certain port interface.			
Base Class(es)	PortPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
required ComSpec	RPortCom Spec	*	aggregation	Required communication attributes, one for each interface element.
required Interface	PortInter-face	1	reference to type	The interface that this port requires, i.e. the port depends on another port providing the specified interface.

Table 2.5: RPortPrototype

Class	⟨⟨**atpPrototype**⟩⟩ **PPortPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Desc.	Component port providing a certain port interface.			
Base Class(es)	PortPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
provided ComSpec	PPortCom Spec	*	aggregation	Provided communication attributes per interface element (data element or operation).
provided Interface	PortInter-face	1	reference to type	The interface that this port provides.

Table 2.6: PPortPrototype

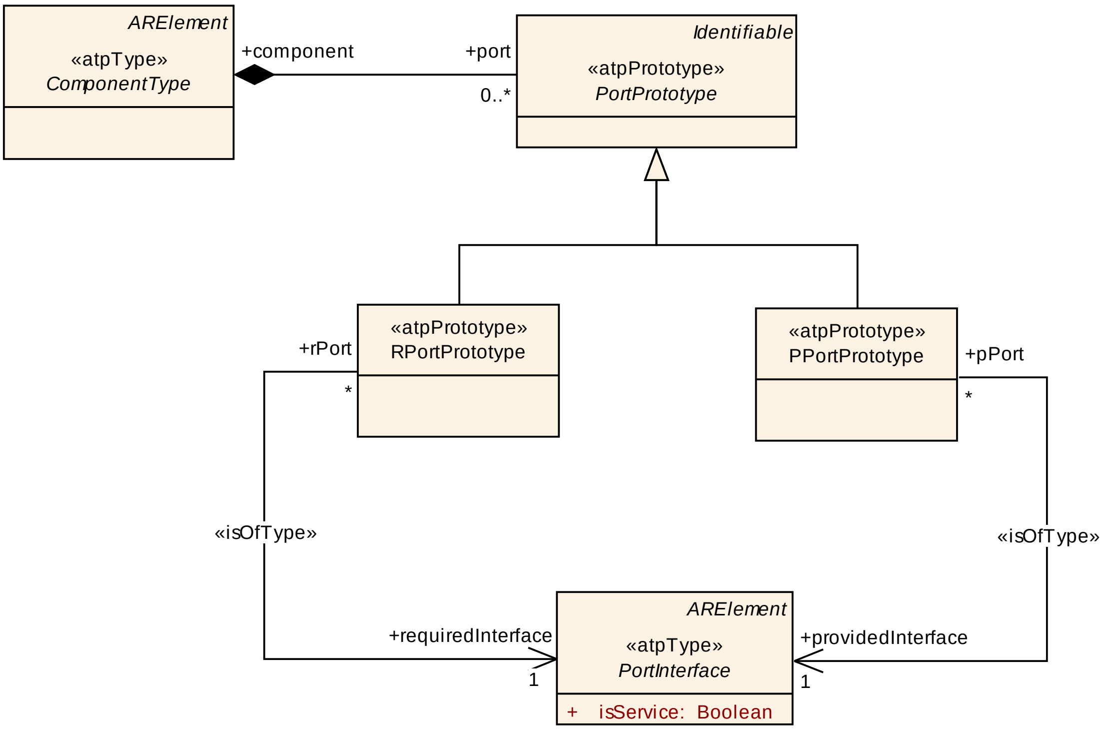


Figure 2.3: Components and Ports

2.3 Composition

The purpose of an AUTOSAR `CompositionType` is to allow the encapsulation of specific functionality by aggregating existing software-components. Since a `CompositionType` is also a `ComponentType`, it again may be aggregated in further `CompositionTypes`. This recursive relation is formally expressed in figure 2.4.

It is important to understand that while compositions allow for (sub-) system abstraction, they are solely an *architectural element for the implementation of model scalability*. They simply group existing software-components and thereby take away complexity when viewing or designing logical system architecture.

Therefore, the definition of `CompositionTypes` has no effect on how software-components interact with the Virtual Functional Bus (VFB). `CompositionTypes` do not add any new functionality to what is already provided by the software-components they aggregate. As the main consequence, `CompositionTypes` do not have any binary footprint in the ECU software.

In terms of the AUTOSAR meta-model, a composition of software-components realized by the meta-class `CompositionType` aggregates `ComponentPrototypes` which in turn are typed by a `ComponentType`. Please note that a `CompositionType` is also a `ComponentType`.

Class	⟨⟨**atpType**⟩⟩ **CompositionType**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition

Class Desc.	A CompositionType aggregates ComponentPrototypes (that in turn are typed by ComponentTypes) as well as ConnectorPrototypes for primarily connecting ComponentPrototypes among each others and towards the surface of the CompositionType. By this means hierarchical structures of software-components can be created.			
Base Class(es)	ComponentType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
component	Component Prototype	1..*	aggregation	The instantiated components that are part of this composition.
connector	Connector Prototype	*	aggregation	ConnectorPrototypes have the principal ability to establish a connection among PortPrototypes. They can have many roles in the context of a CompositionType. Details are refined by subclasses.

Table 2.7: CompositionType

Class	⟨⟨**atpPrototype**⟩⟩ **ComponentPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Class Desc.	Role of a software component within a composition.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
type	Component Type	1	reference to type	Type of the instance.

Table 2.8: ComponentPrototype

Therefore, a `ComponentPrototype` implements the usage of a `ComponentType` in a specific *role*. In general, arbitrary numbers of `ComponentPrototypes` that refer to specific `ComponentTypes` can be created. Note that `CompositionType` also aggregates the abstract meta-class `ConnectorPrototype` for connection the `ComponentPrototypes` contained among each others (see figure 2.5).

Example: a `ComponentPrototype` ”LeftDoorControl” fulfills the role of implementing the `ComponentType` ”DoorControl” for the left door of a vehicle while the `ComponentPrototype` ”RightDoorControl” fulfills the role of the `ComponentType` ”DoorControl” for the right door.

Note that being a `CompositionType`, a `CompositionType` also exposes `PortPrototypes` to the outside world. However, the `PortPrototypes` are only delegated and do not play the same role as `PortPrototypes` attached to `AtomicSoftwareComponentTypes`. Being a `PortPrototype` attached to a `CompositionType` has the following implications:

- The delegation has to follow the rules defined in chapter 3.4.

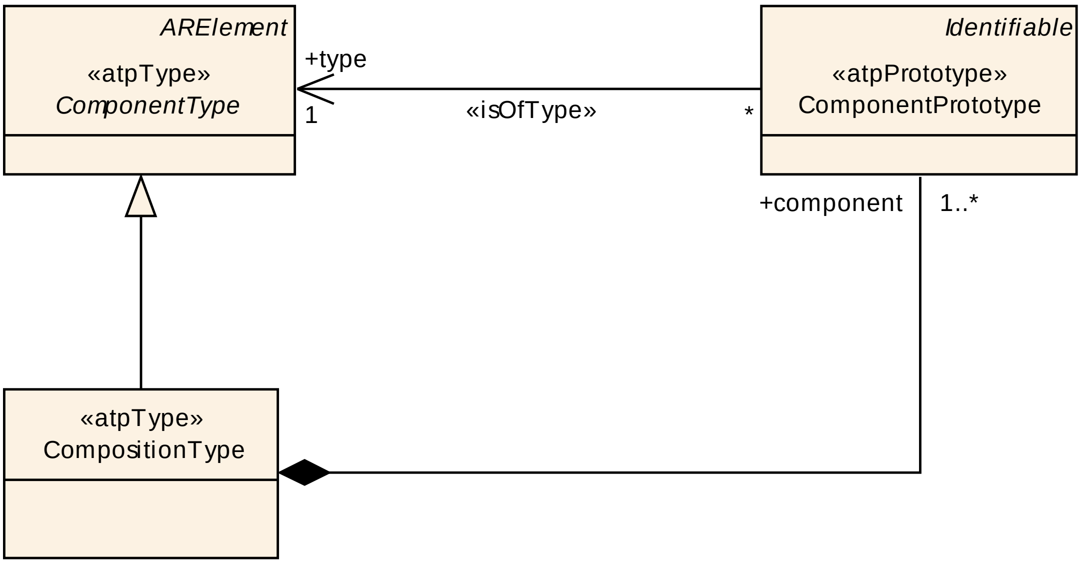


Figure 2.4: The recursive relation of software-components and compositions

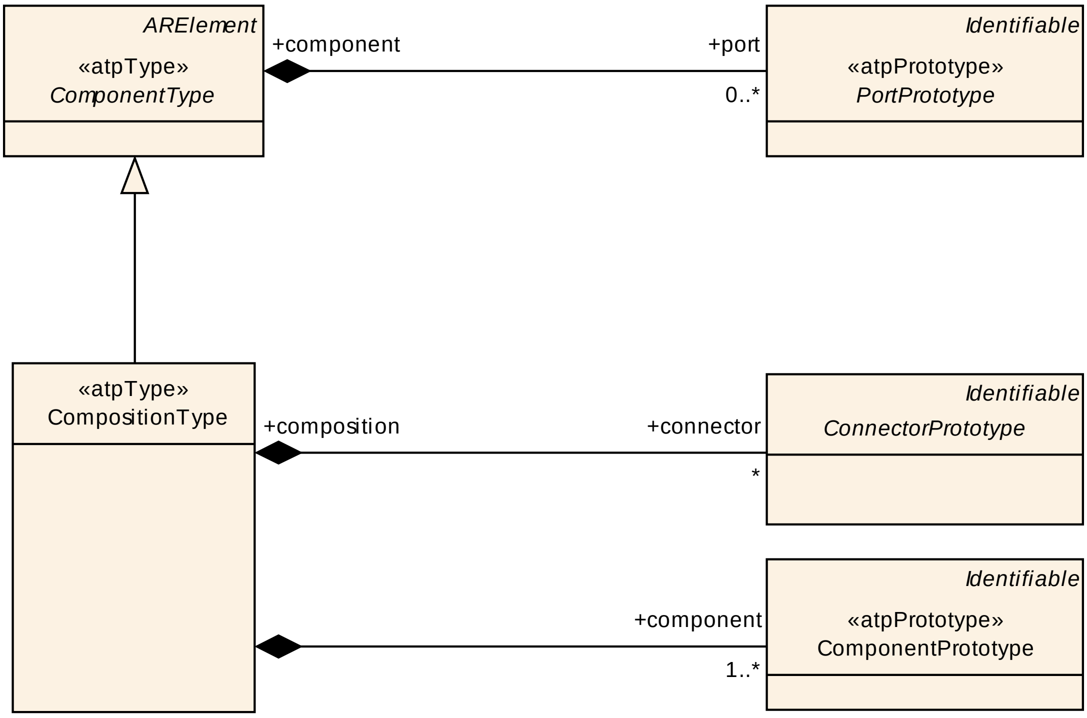


Figure 2.5: Composition and the meta-classes aggregated

- By creating `PortPrototypes` on the surface of a specific `CompositionType` it is explicitly decided whether or not the contents of an ”inner” port contained in the `CompositionType` is exposed to the outside world.

Please note that the semantics of the delegation of `PortPrototypes` are similar to encapsulation mechanisms like public and private members in object-oriented programming languages.

`CompositionTypes` contain three kinds of `ConnectorPrototypes`:

- `AssemblyConnectorPrototypes` to interconnect `PortPrototypes` of `ComponentPrototypes` that are part of the `CompositionType` as well as
- `DelegationConnectorPrototypes` to connect from ”inner” `PortPrototypes` to delegated ”outer” `PortPrototypes`.

  In the case that the outer `PortPrototypes` is referenced by multiple `DelegationConnectorPrototypes` the semantic is the multiplication of the `AssemblyConnectorPrototypes` referencing the outer `PortPrototypes`.
- `ServiceConnectorPrototype` is exclusively used for in the context of ECU configuration phase, and must not be used within `CompositionTypes` of software applications. Please find more details in chapter 10.

Class	⟨⟨**atpObject**⟩⟩ **ConnectorPrototype (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Class Desc.	The base class for connectors between ports. Connectors have to be identifiable to allow references from the system constraint template.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 2.9: ConnectorPrototype

Class	⟨⟨**atpStructureElement**⟩⟩ **AssemblyConnectorPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Class Desc.	AssemblyConnectorPrototypes are exclusively used to connect ComponentPrototypes in the context of a CompositionType.			
Base Class(es)	ConnectorPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
provider	PPort Prototype	1	instanceRef	Instance of providing port.
requester	RPort Prototype	1	instanceRef	Instance of requiring port.

Table 2.10: AssemblyConnectorPrototype

Class	⟨⟨**atpStructureElement**⟩⟩ **DelegationConnectorPrototype**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition
Class Desc.	A delegation connector delegates one inner PortPrototype (a port of a component that is used inside the composition) to a outer PortPrototype of compatible type that belongs directly to the composition (a port that is owned by the composition).
Base Class(es)	ConnectorPrototype

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
innerPort	PortPrototype	1	instanceRef	Connects these ports. The role (inner, outer) of those ports is derived from the context (port of composition or port of inner component).
outerPort	PortPrototype	1	reference	The port that is located on the outside of the CompositionType

Table 2.11: DelegationConnectorPrototype

Class	⟨⟨**atpStructureElement**⟩⟩ **ServiceConnectorPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Class Desc.	A ServiceConnectorPrototype connects a PortPrototype owned by an ComponentPrototype with the service PortPrototype owned by the ServiceComponentPrototype. A ServiceConnectorPrototype is only added to the model in ECU Configuration phase for the specific purpose of configuring services within an EcuSwComposition.			
Base Class(es)	ConnectorPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
applicationPort	PortPrototype	1	instanceRef	Service port to be connected on application component side
servicePort	PortPrototype	1	instanceRef	Service port to be connected on service component side

Table 2.12: ServiceConnectorPrototype

One implication of the concept of `CompositionType` is that the application software of an entire vehicle eventually is represented by one `CompositionType`. This so-called top-level composition has a special role in the context of the AUTOSAR System Template [10]. However, please note note that a top-level composition might have (unconnected) `PortPrototypes` in order to allow for reuse as part of another system.

2.4 Port Interface

A `PortPrototype` mainly contributes the functionality of being a connection point to the AUTOSAR concept. The details, i.e. what kind of information is actually transported between two `PortPrototypes` is defined by the `PortInterface`.

`PortInterfaces` (see figure 2.7) are used to support a design-by-contract work flow, i.e. they provide means to formally verify structural and dynamic compatibility between software-components. In other words: `PortInterfaces` represent a pivotal point in the AUTOSAR concept.

Please note that a `PortInterface` creates a name space for the information contained. This allows for defining the details of a specific `PortInterface` without hav-

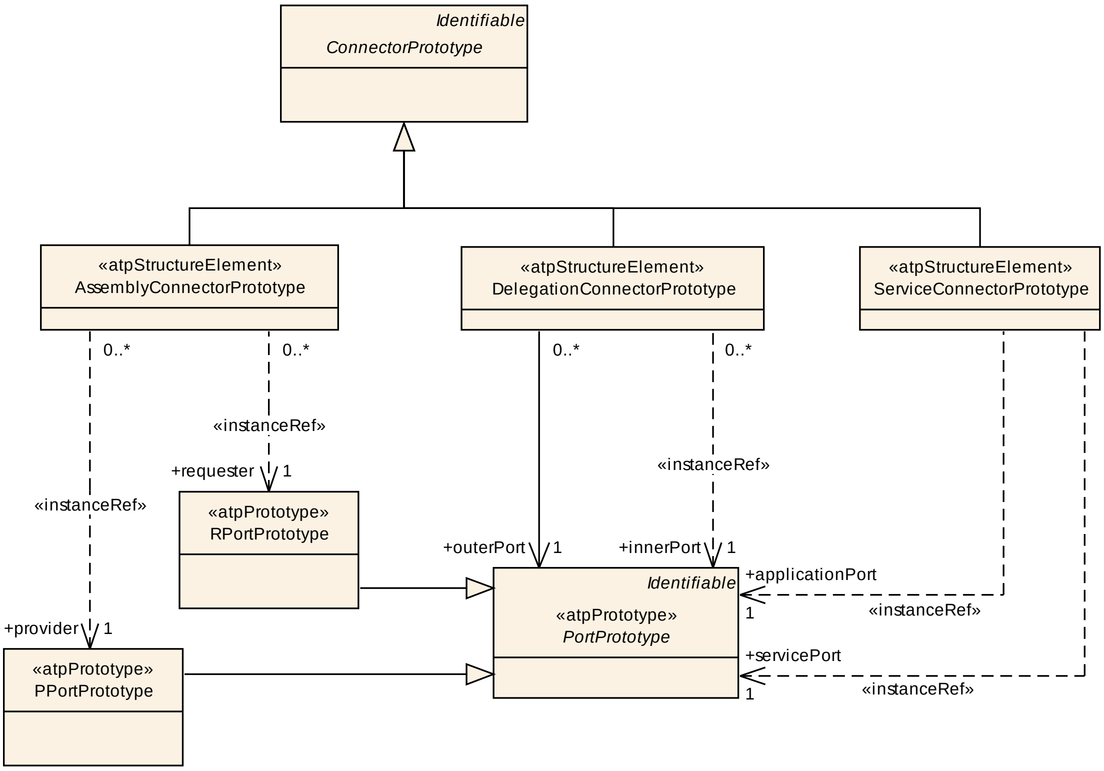


Figure 2.6: Connectors

ing to care for possible side-effects on other `PortInterfaces`. Again, this property of the AUTSOAR concept directly supports re-usability.

Within the AUTOSAR concept, different flavors of `PortInterfaces` are defined:

- `SenderReceiverInterface`,
- `ClientServerInterface`, and the
- `CalprmInterface`.

Please find more details about the specialization of the `PortInterface` concept in chapter 3.3 and 3.2.

Class	**⟨⟨atpType⟩⟩ PortInterface (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Desc.	Abstract base class for an interface that is either provided or required by a port of a software component.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

isService	Boolean	1	aggregation	This flag is set, if the PortInterface is to be used for communication between an ApplicationSoftwareComponentType and a ServiceComponentType (namely an AUTOSAR Service, ECU abstraction or Complex Driver) located on the same ECU. Otherwise the flag is not set.

Table 2.13: PortInterface

From an abstract point of view, a `PortInterface` acts as a *type* for a `PortPrototype`. This means in particular that several `PortPrototypes` can be typed by the same `PortInterface`. Of course, this aspect facilitates the creation of valid connections between software-components dramatically. By using a specific `PortInterface` for typing particular `PortPrototypes` the latter are eligible for being connected to each other by definition.

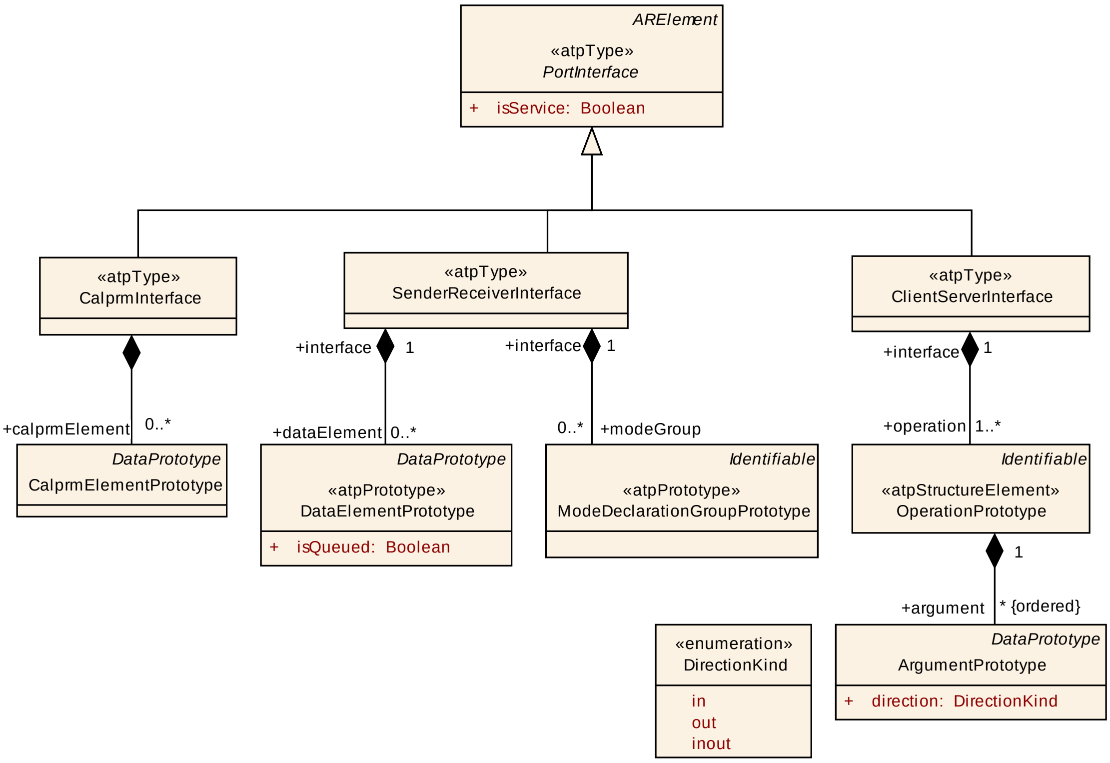


Figure 2.7: `PortInterfaces` in the AUTOSAR meta-model

However, the creation of a valid connection does not need to be based on the usage of identical `PortInterfaces`. It is also possible to use different, but *compatible* `PortInterfaces`. The details about compatibility of `PortInterfaces` are described in chapter 3.4.

Please note that `PortInterfaces` also play an important role in the context of defining so-called AUTOSAR services. Please find more details about this aspect in chapter 10.

3 Details: Software Components, Ports, and Interfaces

3.1 Introduction

The specification of the Virtual Functional Bus (VFB) [3] explains the main communication paradigms for communication among software-components: *client/server* for operation-based communication, and *sender/receiver* for data-based communication. The nature of the two communication paradigms is quite different, and so are the attributes for `SenderReceiverInterfaces` and `ClientServerInterfaces`.

`PortInterfaces` are limited to the description of the static structure of the exchanged information; the dynamic attributes (please refer to chapter 3.6.1) relevant for communication are attached to `PortPrototypes`.

3.2 Sender Receiver Communication

`SenderReceiverInterfaces` allow for the specification of the typically asynchronous communication pattern where a sender provides data that is required by one or more receivers. While the actual communication takes place via the respective `PortPrototypes`, a `SenderReceiverInterface` allows for formally describing what kind of information is sent and received.

Class	⟨⟨**atpType**⟩⟩ **SenderReceiverInterface**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Desc.	A sender/receiver interface declares a number of data elements to be sent and received.			
Base Class(es)	PortInterface			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
dataElement	DataElementPrototype	*	aggregation	The dataelements of this sender/receiver interface.
mode Group	ModeDeclarationGroup Prototype	*	aggregation	Modes which may be communicated via this interface.

Table 3.1: SenderReceiverInterface

A `SenderReceiverInterface` focuses on the description of information items represented by `DataElementPrototypes` and `ModeDeclarationGroupPrototypes`.

3.2.1 Data Element Prototype

A `DataElementPrototype` represents an atomic[1] piece of information transmitted among `PortPrototypes` typed by a `SenderReceiverInterface`. Any `DataElementPrototype` has a specific data type, i.e. technically speaking it is a `DataPrototype` (see figure 3.1).

Class	⟨⟨**atpPrototype**⟩⟩ **DataElementPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Desc.	A data element of a sender-receiver interface, supporting signal like communication patterns.			
Base Class(es)	DataPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
isQueued	Boolean	1	aggregation	Qualifies whether the content of the data element is queued. If it is queued, then the data element has ”event” semantics, i.e. data elements are stored in a queue and all data elements are processed in ”first in first out” order. If it is not queued, then the ”last is best” semantics applies. Please note: Depending on the read access cycle to the data element some values might not be processed by the receiver.

Table 3.2: DataElementPrototype

Class	⟨⟨**atpPrototype**⟩⟩ **DataPrototype (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	Base class for prototypical roles of a datatype.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

---

[1] Note that the term ”atomic” does not have any implication on the implementation on a concrete computing platform

swDataDefProps	SwDataDefProps	0..1	aggregation	This element describes all of the distinguishing characteristics of a data object (variable or parameter). <swDataDefProps> is used in every case, where characteristics of data objects must be given. It is inevitable that not all of the inputs are useful all of the time. Hence, the process definition or the DCI has the task of implementing limitations. The <swDataDefProps> describe the characteristics of all axes: * The characteristics of the argument axes (abscissas) are described in <swCalprmAxisSet> . * The characteristics of the value axis are described directly in <swDataDefProps> .
type	Datatype	1	reference to type	

Table 3.3: DataPrototype

Note that a `SenderReceiverInterface` provides a name space for the definition of `DataElementPrototypes`. In terms of the AUTOSAR meta-model this aspect is indicated by the inheritance relation to `DataPrototype` (which in turn inherits from `Identifiable`). Please find more information on the creation of name spaces in [6].

A further implication of this relationship is that a `DataElementPrototype` can be typed by a `PrimitiveType` but also by a `CompositeType`.

The attribute `isQueued` indicates the way how a `DataElementPrototype` must be processed at the receiver's side. If set to `TRUE` the semantics of the attribute is that the corresponding `DataElementPrototype` needs to be added to a *queue* (or in other words: a FIFO data structure) from which it is later consumed by the actual receiver software-component.

If the attribute is set to `FALSE` then *last is best* semantics applies. Please note that depending on the read access on the receiver side it might happen that some updates of the value of a `DataElementPrototype` with `isQueued` set to `FALSE` are actually missed.

Please note that the definition of `DataElementPrototype` may possibly come very close to the reader's idea of a *signal*. However, different kinds of signals have a specific meaning in the AUTOSAR concept, especially in the context of the AUTOSAR System Template [10].

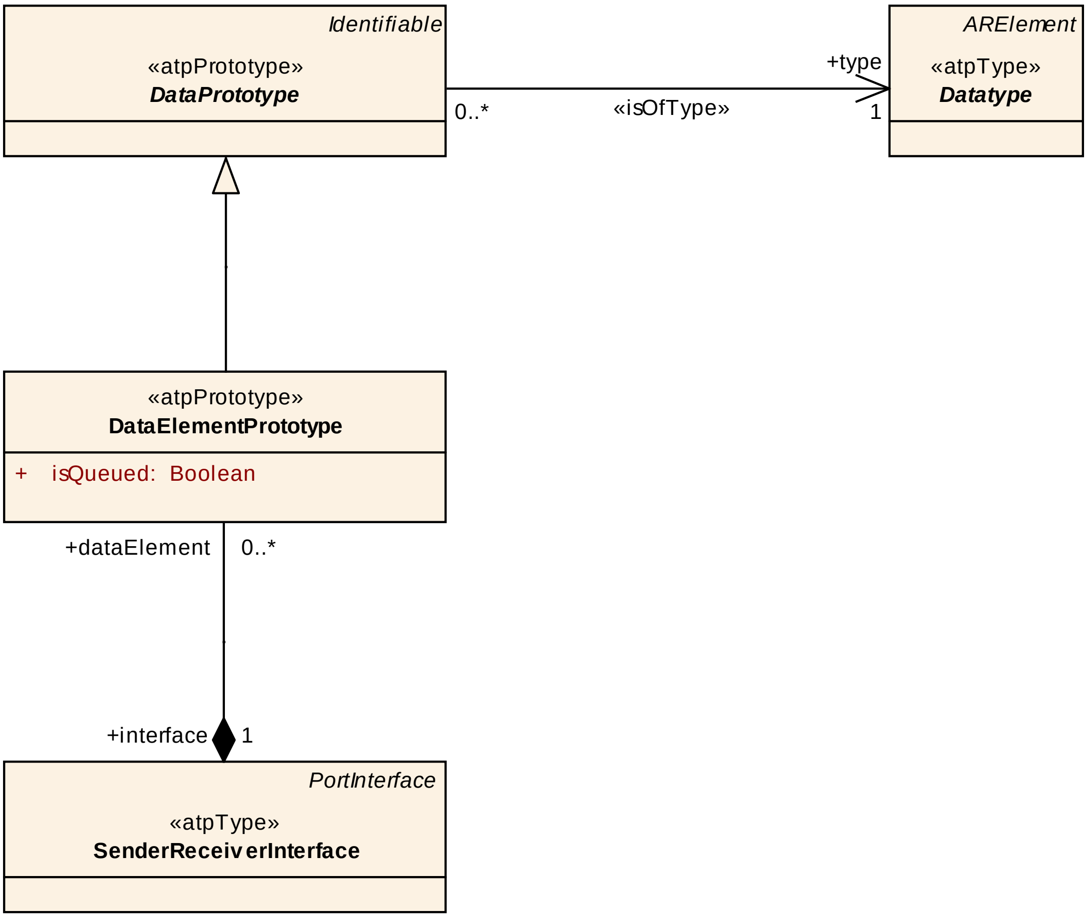


Figure 3.1: `DataElements` of a `SenderReceiverInterface`

3.2.2 Mode Declaration Group Prototype

In addition to the mere definition of exchanged information items by means of `DataElementPrototypes`, a `SenderReceiverInterface` may define `ModeDeclarationGroupPrototypes` which describe a collection of mode switches that can be communicated via the specific `SenderReceiverInterface`.

Class	⟨⟨**atpPrototype**⟩⟩ **ModeDeclarationGroupPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Desc.	The ModeDeclarationGroupPrototype specifies the set of Modes (ModeDeclarationGroup) that is supported by a ComponentType.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
type	ModeDeclarationGroup	1	reference to type	The ”collection of ModeDeclarations” (= ModeDeclarationGroup) supported by a component

Table 3.4: ModeDeclarationGroupPrototype

3.3 Client Server Communication

The underlying semantics of a client/server communication is that a client may initiate the execution of an operation by a server that supports the operation. The server executes the operation and immediately provides the client with the result (synchronous operation call) or else the client checks for the completion of the operation by itself (asynchronous operation call).

3.3.1 Client Server Interface

A `ClientServerInterface` therefore to some extent is a counterpart to the `SenderReceiverInterface`. Instead of defining pieces of information to be transferred among software-components, a `ClientServerInterface` defines a collection of `OperationPrototypes`.

Class	⟨⟨**atpType**⟩⟩ **ClientServerInterface**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Desc.	A client/server interface declares a number of operations that can be invoked on a server by a client.			
Base Class(es)	PortInterface			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
operation	Operation Prototype	1..*	aggregation	
possible Error	Application Error	*	aggregation	Application errors that are defined as part of this interface.

Table 3.5: ClientServerInterface

As depicted in figure 3.2, a `ClientServerInterface` is composed of `OperationPrototypes`, i.e. an `OperationPrototype` cannot be reused in the context of a different `ClientServerInterface`

An `OperationPrototype` consists of 0..* `ArgumentPrototypes`. The latter may be

- passed to the operation
- passed to and returned from the operation
- returned from the operation

Class	⟨⟨**atpStructureElement**⟩⟩ **OperationPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Desc.	An operation declared within the scope of a client/server interface.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

argument (ordered)	Argument Prototype	*	aggregation	
possible Error	Application Error	*	reference	Possible errors that may by raised by referring operation.

Table 3.6: OperationPrototype

Class	⟨⟨**atpPrototype**⟩⟩ **ArgumentPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Desc.	An argument of an operation, much like a data element, but also carries direction information and is associated with a particular operation.			
Base Class(es)	DataPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
direction	Direction Kind	1	aggregation	

Table 3.7: ArgumentPrototype

To cover these cases `ArgumentPrototype` defines an attribute `direction`, possible values are `in` (pass to operation), `out` (return from operation), and `inout` (pass to and return from operation).

In many common programming languages (like *C*), an operation is yet another data type. This makes it for example possible to pass a reference to an operation as an argument to another operation. This is *not* allowed in the AUTOSAR concept: it is not possible to pass a reference to an `OperationPrototype` as an `ArgumentPrototype` in another `OperationPrototype`.

Essentially all `ArgumentPrototypes` in an `OperationPrototype` can be passed (conceptually) by value (from the client to the server and/or from the server to the client depending on the `direction` of the `ArgumentPrototype`). Extending the model to allow this causes a huge additional level of complication within the RTE (as the RTE now would need to deal with references to remote objects).

When the client invokes an operation, it needs to provide a value for each `ArgumentPrototype` that is of direction `in` or `inout`. This value needs to be of the correct `Datatype`. In the case of synchronous operation call, the client expects to receive a response to the invocation of the operation. As part of the response, it receives a value (of the correct `Datatype`) for each `ArgumentPrototype` that is of direction `out` or `inout`.

Enumeration	**DirectionKind**
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface
Enum Desc.	
Literal	***Description***
out	The ArgumentPrototype is passed from the OperationPrototype to the caller.
inout	The ArgumentPrototype is passed to the OperationPrototype but also passed back from the OperationPrototype to the caller.

in	The ArgumentPrototype is passed to an OperationPrototype

Each `OperationPrototype` provides a name space for its `ArgumentPrototypes` and therefore has a unique identifier, which identifies the operation within the corresponding `ClientServerInterface`. The `OperationPrototypes` have no ordering within a `ClientServerInterface` (there is no such thing as the ”first” operation)[2].

It is not possible to define default values for `ArgumentPrototypes` defined in the context of an `OperationPrototype`. Default values might lead to complicated mappings to programming languages.

In contrast to the unordered relationship of `ClientServerInterface` to `OperationPrototype`, the definition of `ArgumentPrototypes` within the context of an `OperationPrototype` is ordered, i.e. an `OperationPrototype` may have a *first* argument[3].

Please note that `ArgumentPrototype` inherits from `DataPrototype` and therefore has a reference to a concrete `Datatype`.

Class	**⟨⟨atpPrototype⟩⟩ DataPrototype (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	Base class for prototypical roles of a datatype.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

---

[2] In different parts of the definition of a `ClientServerInterface`, a ”calling-order” of the `OperationPrototypes` might be prescribed: the client might be required to use the `OperationPrototypes` in a certain logical ordering. However, this ordering has nothing to do with the order in which the `OperationPrototypes` are listed in the definition of a `ClientServerInterface`

[3] Giving the `ArgumentPrototypes` of an `OperationPrototype` both an ordering and a unique identifier might seem redundant. For example, in the operation ”foo(a, b, c)”; we can refer to the ”second argument” or to ”the argument named b”. In many common programming languages (like C or Java), only the *ordering* is actually used by the client during the invocation of the server (the client invokes the operation as ”foo(1,2,3)” not as ”foo(a=1,c=3,b=2)”. In addition, the names of the arguments represent an arbitrary choice made when implementing of the invocation. In *C*, only the data types and ordering of the arguments constitute the signature, *not* the names of the arguments.

swDataDef Props	SwData DefProps	0..1	aggregation	This element describes all of the distinguishing characteristics of a data object (variable or parameter). <swDataDefProps> is used in every case, where characteristics of data objects must be given. It is inevitable that not all of the inputs are useful all of the time. Hence, the process definition or the DCI has the task of implementing limitations. The <swDataDefProps> describe the characteristics of all axes: * The characteristics of the argument axes (abscissas) are described in <swCalprmAxisSet> . * The characteristics of the value axis are described directly in <swDataDefProps> .
type	Datatype	1	reference to type	

Table 3.8: DataPrototype

Note further that a `ClientServerInterface` does not define any timing information (how quickly the client expects a response of the server). It does not define how the threading works (if the client for example blocks until the response comes back from the server).

It also does not define explicitly how information is passed between an implementation of the client and the server and the underlying RTE (for example: through ”pointers” or ”by value”).

3.3.2 Error Handling in client/server communication

This section describes the handling of errors occurring either within an application software-component or during the communication across the VFB [3]. Errors that are created and consumed by basic software modules are not in scope.

Therefore, errors in the scope of this document are divided into two simple classes:

- infrastructure errors and
- application errors.

A software-component implementation uses RTE API methods to communicate with other software-components. During this communication certain errors can occur as a result of infrastructure faults, like a bus not working, or an expected data value not arriving in time.

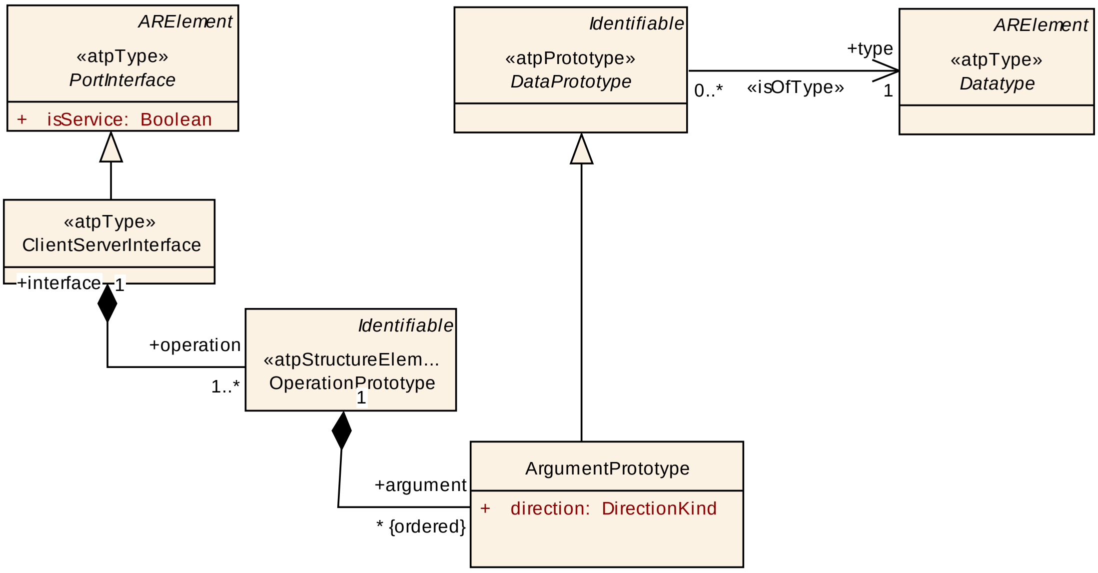


Figure 3.2: `Operations` of a `ClientServerInterface`

These errors are listed in the VFB specification [3], as they are an inherent feature of the infrastructure provided by the VFB. Software-components will therefore typically not raise infrastructure errors on their own. Instead, basic software and RTE will determine infrastructure faults and communicate the corresponding errors to the relevant software-components.

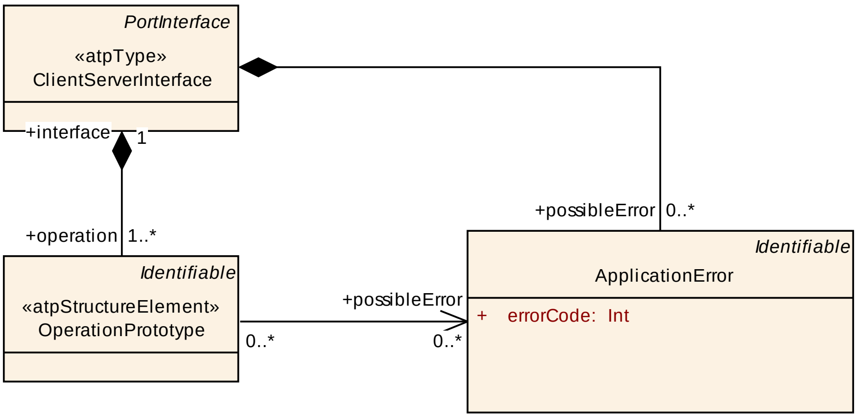


Figure 3.3: Application error meta-model

As the fixed set of infrastructure errors is defined as an implicit part of the VFB, a developer of an AUTOSAR system does not need to explicitly describe them. They are assumed to be possible and application developers should take measures to handle them.

Application errors on the other hand are specific to the functionality or information that is described in form of a `PortInterface`. It is not possible to define such errors up front, instead they are defined at design time of a certain `PortInterface`. In principle, such `ApplicationErrors` could be part of all kinds of `PortInterfaces`, but as of now, AUTOSAR supports (as depicted by figure 3.3) `ApplicationErrors` only for `ClientServerInterfaces`.

Class	⟨⟨**atpObject**⟩⟩ **ApplicationError**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Desc.	This is a user-defined error that is associated with an element of an AUTOSAR interface. It is specific for the particular functionality or service provided by the AUTOSAR software component.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
errorCode	Integer	1	aggregation	The RTE generator is forced to assign this value to the corresponding error symbol. Note that for error codes certain ranges are predefined (see RTE specification).

Table 3.9: ApplicationError

Consequently, `OperationPrototypes` may be associated with a number of `ApplicationErrors` they possibly raise. These errors are defined as part of the `ClientServerInterface`.

3.4 Compatibility

In order to connect `PortPrototypes` of `ComponentTypes`, the compatibility of `PortPrototypes` needs to be verified. This section defines the basic rules for formal compatibility of `PortPrototypes`. 3.4 depicts the meta-classes relevant for the discussion of compatibility.

Compatibility will be defined bottom-up, i.e. first the rules for compatible `Datatypes` are set up, then the rules for the different types of `PortInterfaces` are derived.

3.4.1 Compatibility of Data Types

To fully discuss compatibility rules for `Datatypes`, the different types and objects in the `Datatypes` part of the AUTOSAR meta models have to be cleanly distinguished. Please find more details on AUTOSAR `Datatypes` in chapter 4

The AUTOSAR meta model defines a number of meta classes (e.g. `IntegerType`), that own a set of attributes (e.g. a lower boundary for its values). Instantiating such a class and setting its attributes defines a new `Datatype` (e.g. *Uint16*). In other words: `IntegerType` is an M2 artifact; it is taken for creating an M1 artifact *Uint16*.

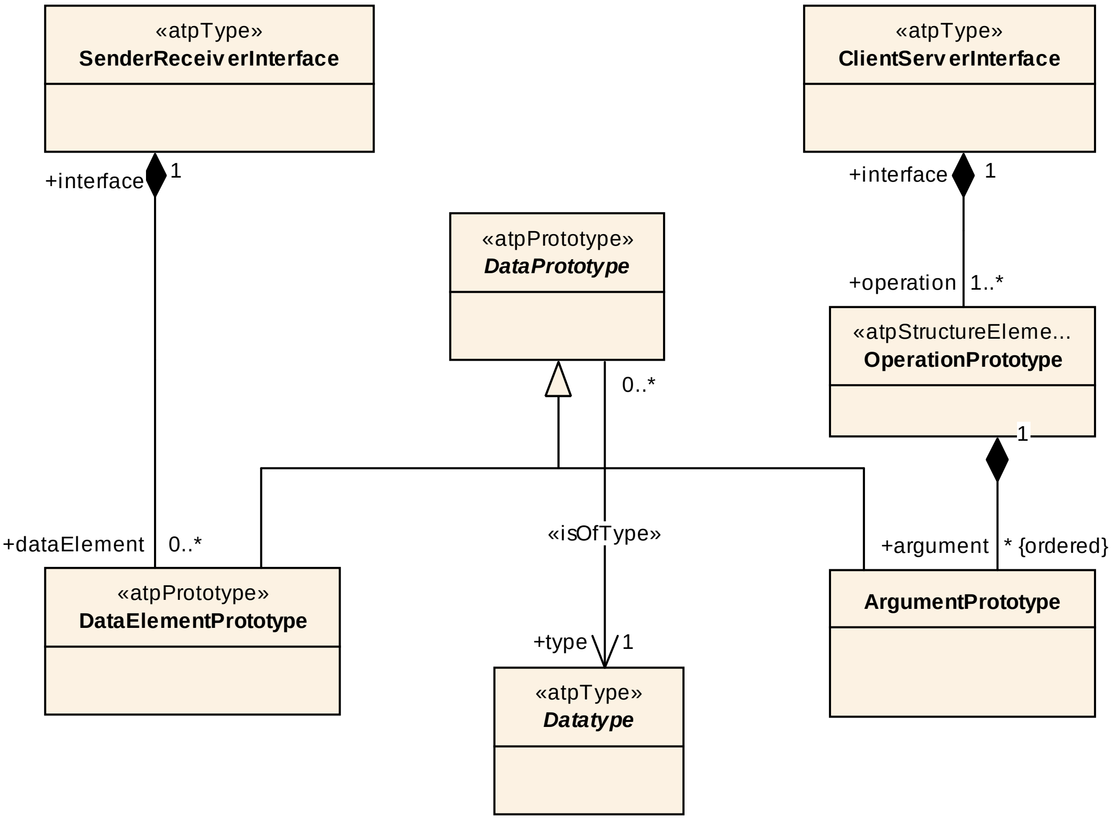


Figure 3.4: Relevant meta-classes for compatibility considerations

In this context, the issue of compatibility refers to the M1 objects, i.e. the instances of `Datatype` need to be considered.

3.4.1.1 PrimitiveType

Instances of `PrimitiveType` are compatible if and only if

1. The examined M1 data types are derived from the same `PrimitiveType`.
2. All attributes match exactly, with one exception: the name of the M1 data type. This rule also covers aliases, which by definition differ only in `shortName` from the original.
3. The semantics of the M1 data types are compatible.

3.4.1.2 CompositeType

Instances of `CompositeType` are compatible if and only if

1. The underlying `CompositeTypes` are identical.

2. They are composed of compatible `Datatypes` (either `CompositeTypes` or `PrimitiveTypes`) in *the same order* (e.g. for `RecordType`).
3. All attributes match exactly, with the exception of the `shortName` of the M1 data type.

3.4.2 Compatibility of Semantics

`PrimitiveTypes` may have associated semantics via aggregated `SwDataDefProps`, which contains semantics in form of a `CompuMethod`, a physical unit (class `Unit`) and an `invalidValue`. These meta-classes are further explained in chapter 4.5. Semantics thus consist of several characteristics that all need to be compatible to satisfy the overall compatibility requirement. This is automatically the case if both `PrimitiveTypes` refer to the same semantics objects.

In general, semantics of `PrimitiveTypes` are compatible if and only if:

1. They refer to compatible `Unit` definitions, or neither of them has an associated `Unit`.
2. They contain identical conversion methods `compuPhysToInternal` from physical to internal values, or neither of them associates such a method.
3. They contain identical conversion methods `compuInternalToPhys` from internal to physical values, or neither of them associates such a method.
4. They contain (if applicable) the same `invalidValue`.

Identical methods refers to conversion methods where all attributes are identical.

Two `Unit` definitions are compatible if and only if:

1. They have identical `shortNames`.
2. They have identical attributes `factorSiToUnit` and `offsetSiToUnit`.
3. They either refer to identical definitions of `PhysicalDimension` or neither of them associates a `PhysicalDimension`.

Two `PhysicalDimension` definitions are identical if they have identical `shortNames` and attributes.

3.4.3 Compatibility of Data Element Prototypes

Although `DataElementPrototypes` can only exist in the context of a `SenderReceiverInterface`, they are discussed separately.

Two `DataElementPrototypes` are compatible if and only if

1. They are typed by (read ”refer to”) compatible `Datatypes`.

2. The two `DataElementPrototypes` have identical `shortNames`. This is required to map `DataElementPrototypes` in unordered `SenderReceiverInterfaces`.
3. For each such pair, the values of their `isQueued` attributes are equal.

3.4.4 Compatibility of Mode Declaration Groups

`ModeDeclarationGroups` are compatible if and only if

1. They have identical `ModeDeclarations`.
2. They refer to identical `initialModes`.

3.4.5 Compatibility of Sender Receiver Interfaces

Please note that this compatibility requirement only satisfies static correctness, which means that logical consistency is not assured (e.g. that a receiver must process a certain data value to correctly interpret the following values).

3.4.5.1 Connection of required and provided Port via AssemblyConnectorPrototype

The compatibility of `SenderReceiverInterfaces` is considered for connecting of `PortPrototypes` with an `AssemblyConnectorPrototype`. `PortPrototypes` of different `SenderReceiverInterfaces` are compatible if and only if

1. For each `DataElementPrototype` defined in the context of the `SenderReceiverInterface` of the required `PortPrototype` a compatible `DataElementPrototype` exists in the `SenderReceiverInterface` of the provided `PortPrototype`. The `shortNames` of `DataElementPrototypes` are used to identify the pair.
2. For each `ModeDeclarationGroupPrototype` defined in the context of the `SenderReceiverInterface` of the required `PortPrototype` a compatible `ModeDeclarationGroupPrototype` exists in the `SenderReceiverInterface` of the provided `PortPrototype`. The `shortNames` of the `ModeDeclarationGroupPrototypes` are used to identify the pair.
3. For each such pair, the values of their `isService` attributes are identical.

3.4.5.2 Connection of inner and outer Port via DelegationConnectorPrototype

The compatibility of `SenderReceiverInterfaces` is considered for connecting of `PortPrototypes` with a `DelegationConnectorPrototype`. `PortPrototypes` of different `SenderReceiverInterfaces` are compatible if and only if

1. For each `DataElementPrototype` defined in the context of the `SenderReceiverInterface` of the required inner `PortPrototype` a compatible `DataElementPrototype` exists in the `SenderReceiverInterface` of the required outer `PortPrototype`. The `shortNames` of `DataElementPrototypes` are used to identify the pair.
2. For each `ModeDeclarationGroupPrototype` defined in the context of the `SenderReceiverInterface` of the required inner `PortPrototype` a compatible `ModeDeclarationGroupPrototype` exists in the `SenderReceiverInterface` of the required outer `PortPrototype`. The `shortNames` of the `ModeDeclarationGroupPrototypes` are used to identify the pair.
3. For at least one `DataElementPrototype` defined in the context of the `SenderReceiverInterface` of the provided inner `PortPrototype` a compatible `DataElementPrototype` exists in the `SenderReceiverInterface` of the provided outer `PortPrototype`. The `shortNames` of `DataElementPrototypes` are used to identify the pair.
4. For at least one `ModeDeclarationGroupPrototype` defined in the context of the `SenderReceiverInterface` of the provided inner `PortPrototype` a compatible `ModeDeclarationGroupPrototype` exists in the `SenderReceiverInterface` of the provided outer `PortPrototype`. The `shortNames` of the `ModeDeclarationGroupPrototypes` are used to identify the pair.
5. For each such pair, the values of their `isService` attributes are identical.

3.4.6 Compatibility of Argument Prototypes

Two `ArgumentPrototypes` are compatible if and only if

1. They are typed by compatible `Datatypes`.
2. They have the same `direction` (`in`, `out` or `inout`).

3.4.7 Compatibility of Application Errors

Two `ApplicationErrors` are compatible if and only if

1. They have the same `shortName`.

2. They have the same attributes. Especially the `errorCode` must be identical in both `ApplicationErrors`.

3.4.8 Compatibility of Operation Prototypes

Two `OperationPrototypes` are compatible if their signatures match. In particular, they are compatible if and only if

1. They have the same number of `OperationArguments`.
2. The n-th arguments of both `OperationPrototypes` are compatible. This implies ordering of `OperationArguments`.
3. They have the same `shortName` (again allows for mapping in `PortInterfaces`).
4. The required `OperationPrototype` specifies a compatible `ApplicationError` for each `ApplicationError` that is possibly raised by the provided `OperationPrototype`, maybe more.

3.4.9 Compatibility of Client Server Interfaces

Please note that this compatibility requirement only satisfies static correctness, which means that logical consistency is not assured (e.g. that a client must call a certain operation to allow the server to work correctly).

3.4.9.1 Connection of required and provided Port via AssemblyConnectorPrototype

`ClientServerInterfaces` are compatible if and only if

1. For each `OperationPrototype` defined in the context of the `ClientServerInterface` of the required `PortPrototype` a compatible `OperationPrototype` exists in the `ClientServerInterface` of the provided `PortPrototype`. The `shortNames` of `OperationPrototypes` are used to identify the pair.
2. For each such pair, the values of their `isService` attributes are identical.

3.4.9.2 Connection of inner and outer Port via DelegationConnectorPrototype

`ClientServerInterfaces` are compatible if and only if

1. For each `OperationPrototype` defined in the context of the `ClientServerInterface` of the required inner `PortPrototype` a com-

patible `OperationPrototype` exists in the `ClientServerInterface` of the required outer `PortPrototype`. The `shortNames` of `OperationPrototypes` are used to identify the pair.

2. For at least one `OperationPrototype` defined in the context of the `ClientServerInterface` of the provided inner `PortPrototype` a compatible `OperationPrototype` exists in the `ClientServerInterface` of the provided outer `PortPrototype`. The `shortNames` of `OperationPrototypes` are used to identify the pair.
3. For each such pair, the values of their `isService` attributes are identical.

3.4.10 Entire delegation of a provided Port Prototype

The delegation of an provided outer PortPrototype is entire defined, if following criteria are fulfilled:

1. For each `DataElementPrototype` with attribute `isQueued` = `TRUE` present in the `SenderReceiverInterface` of the provided outer `PortPrototype`, there exists at least one connection via `DelegationConnectorPrototype` to a provided inner `PortPrototype` with a compatible `DataElementPrototype` in the `SenderReceiverInterface` of the provided inner `PortPrototype`. The `shortNames` of `DataElementPrototype` are used to identify the pair.
2. For each `DataElementPrototype` with attribute `isQueued` = `FALSE` present in the `SenderReceiverInterface` of the provided outer `PortPrototype`, there exists exactly one connection via `DelegationConnectorPrototype` to a provided inner `PortPrototype` with a compatible `DataElementPrototype` in the `SenderReceiverInterface` of the provided inner `PortPrototype`. The `shortNames` of `DataElementPrototype` are used to identify the pair.
3. For each `ModeDeclarationGroupPrototype` present in the `SenderReceiverInterface` of the provided outer `PortPrototype`, there exists exactly one connection via `DelegationConnectorPrototype` to a provided inner `PortPrototype` with a compatible `ModeDeclarationGroupPrototype` in the `SenderReceiverInterface` of the provided inner `PortPrototype`. The `shortNames` of `ModeDeclarationGroupPrototype` are used to identify the pair.
4. For each `OperationPrototype` present in the `ClientServerInterface` of the provided outer `PortPrototype`, there exists exactly one connection via `DelegationConnectorPrototype` to a provided inner `PortPrototype` with a compatible `OperationPrototype` in the `ClientServerInterface` of the provided inner `PortPrototype`. The `shortNames` of `OperationPrototype` are used to identify the pair.

3.4.11 Split and merge of Data Element Prototypes

With the define Compatibility Rules in chapter 3.4.5 and 3.4.9 it is possible to split and distribute data from a `PortPrototype` of type of a `PortInterface` containing the superset of `DataElementPrototypes` to `PortPrototypes` of type of `PortInterfaces` containing subsets of `DataElementPrototypes`.

The examples showing the relationship between the usage of `DelegationConnectorPrototypes` in different configurations and the `DelegatedPortAnnotation`. Please consider that the `DelegatedPortAnnotation` is usually defined before the internal structure of a `CompositionType` is fully defined. Afterward it has to be consistent or can be removed. But showing it together simplifies the understanding of the mean of the `DelegatedPortAnnotation`.

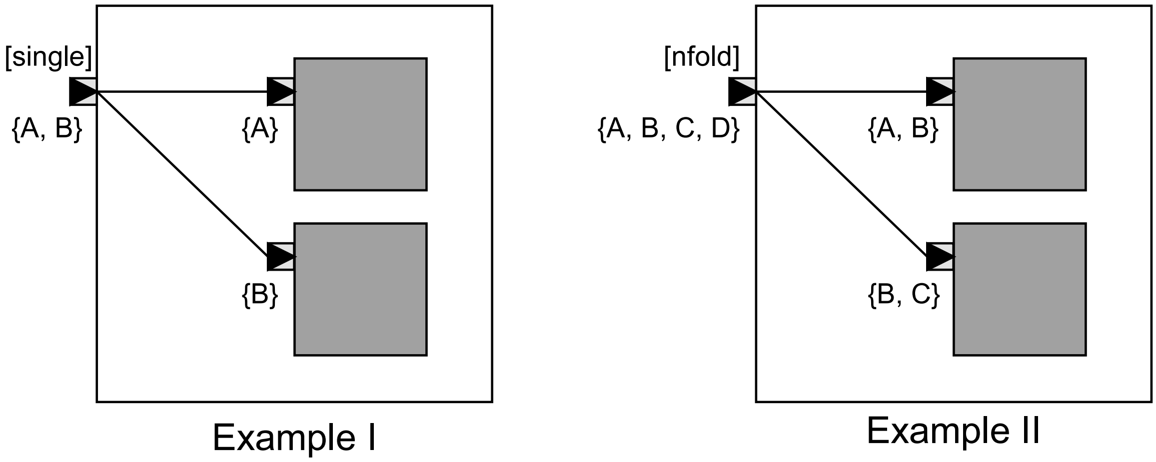


Figure 3.5: Delegation Connector Example I and II

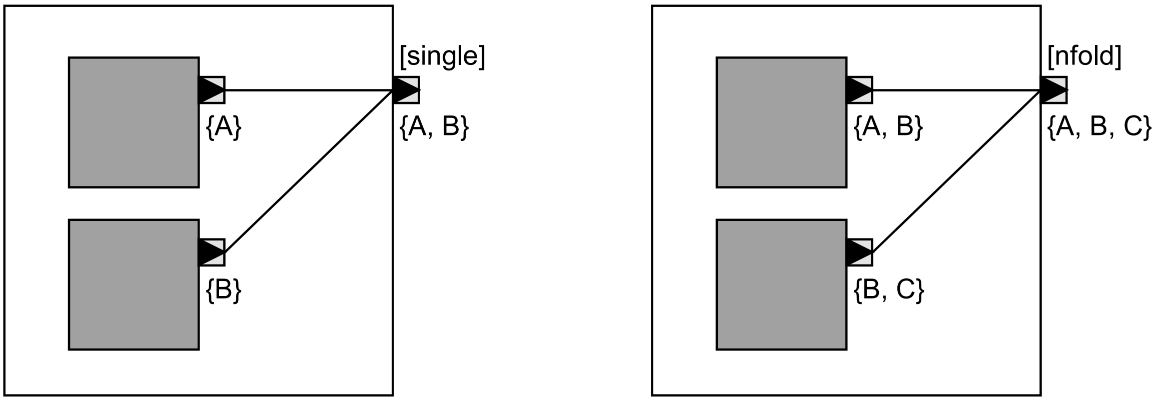
Example I

The required outer `PortPrototype` contains the superset of `DataElementPrototypes` {A ,B}. The two required inner `PortPrototypes` of the `ComponentPrototypes` contain the subsets of `DataElementPrototypes` {A} and {B}. In this case the resulting communication pattern on the `VFB` would be x:1, whereas x can be 1 to n. This would fulfill the criteria of a `DelegatedPortAnnotation` value `single`.

Example II

The required outer `PortPrototype` contains the superset of `DataElementPrototypes` {A ,B, C, D}. The two required inner `PortPrototypes` of the `ComponentPrototypes` contain the subsets of `DataElementPrototypes` {A, B} and {B, C}. In this case the resulting communication pattern on the `VFB` for B would be 1:n. This would require a `DelegatedPortAnnotation` value `nfold`. The data of `DataElementPrototypes` {D} isn't used.

In addition the Compatibility Rules for `DelegationConnectorPrototypes` in chapter 3.4.5.2 and 3.4.9.2 enable merging and collecting of data from `PortPrototypes` of type of `PortInterfaces` containing subsets of `DataElementPrototypes` to a `PortPrototype` of type of a `PortInterface` containing the superset of `DataElementPrototypes`.



Example III

Example IV

Figure 3.6: Delegation Connector Example III and IV

Example III

The provided outer `PortPrototype` contains the superset of `DataElementPrototypes` {A ,B}. The two provided inner PortPrototypes of the `ComponentPrototypes` contain in each case a subset of one `DataElementPrototypes` {A} and {B}. In this case the resulting communication pattern on the `VFB` would be 1:x, whereas x can be 0 to n. This would fulfill the criteria of a `DelegatedPortAnnotation` value single. All `DataElementPrototypes` of the provided outer `PortPrototypes` are provided by exactly one provided inner `PortPrototype`. Therefore the criteria of `entire delegation` defined in chapter 3.4.10 are fulfilled.

Example IV

The provided outer `PortPrototype` contains the superset of `DataElementPrototypes` {A ,B, C}. The two inner `PortPrototypes` of the `ComponentPrototypes` contain the subsets of `DataElementPrototypes` {A, B} and {B, C}. In this case the resulting communication pattern on the `VFB` for {B} would be n:1. This would require a `DelegatedPortAnnotation` value `nfold`. All `DataElementPrototypes` of the provided outer `PortPrototype` are provided by at least on provided inner `PortPrototype`. Therefore the criteria of `entire delegation` defined in chapter 3.4.10 are fulfilled.

3.5 Port Annotation

3.5.1 Introduction

In addition to the formal specification required to implement the communication via ports, a `PortPrototype` can carry so-called `Port Annotations` (please find a summary in figure 3.7). They do not directly influence the signature of calls via this port, but contain further information useful for the application developers of the components on both sides of the connection.

Besides formally specified attributes it is also possible to place textual information as provided in `GeneralAnnotaion`.

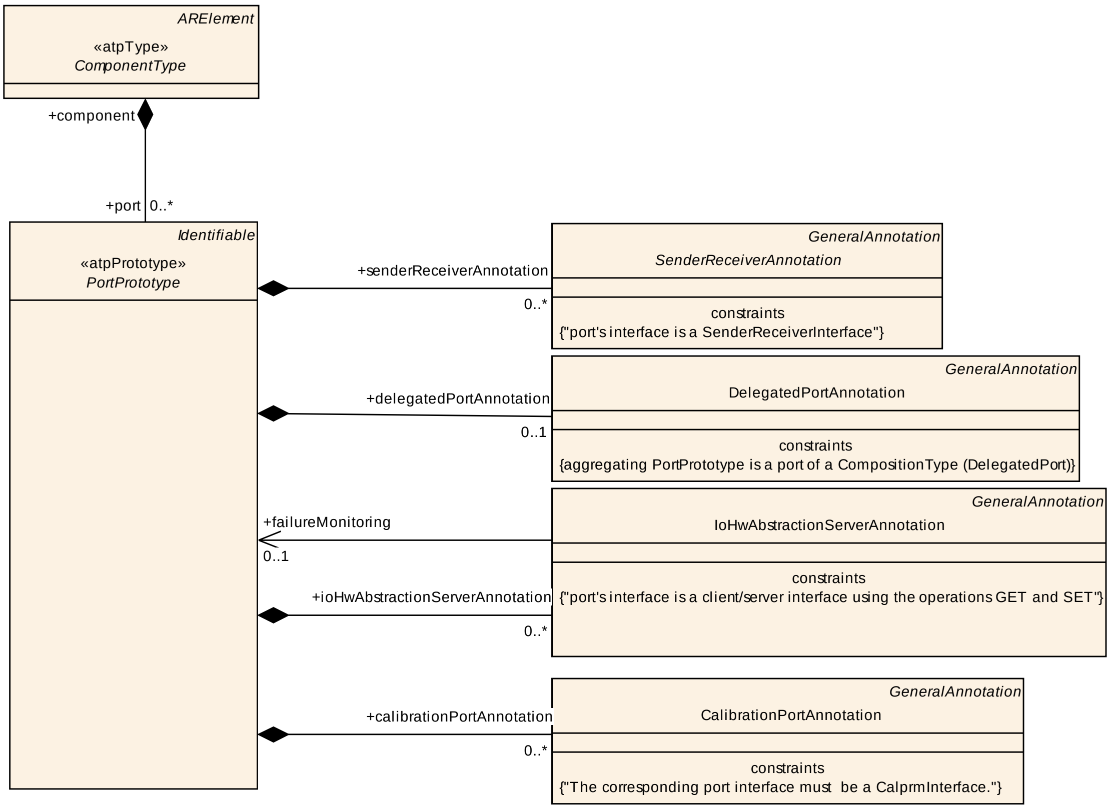


Figure 3.7: Application Level Port Annotations Overview

3.5.2 SenderReceiverAnnotation

Embedded automotive software is used to implement open-loop and closed-loop control-algorithms. Therefore, a software component description has to accommodate typical control engineering description means which have only indirect influence of the embedded software itself. Especially, from the embedded software point of view, these annotations are not reflected by different configuration of the VFB.

However, these annotations give the (function-) developer a direct indication whether a certain software-component is appropriate for the control-algorithm to be designed. A typical annotation is the signal quality, which is characterized by several properties. Each of the property is an annotation in its own.

Typical annotations for sender/receiver communication are:

- Signal Age: The attribute signal age expresses that the associated software-component will only work correctly given that the propagation of the signal from a sensor to a consumer can be finished within a particular time-limit. Of course, this cannot be identified on component or role level, but has to take into account the instance view as well as the actual ECU- and bus-scheduling.
- Raw: A raw signal is typically taken directly from the basic software modules of the ECU abstraction layer. In particular, no sensor software-component has filtered its original value. A `DataElementPrototype` in an `RPortPrototype` of a `ComponentType` using this annotation indicates to the control engineer (who develops a control-algorithm for this component) that the signal has to be filtered (This relationship holds for `SenderReceiverInterfaces`).
- Filtered: The attribute `filtered` indicates that a raw signal has been manipulated by some application software components by using a certain filter.
- Computed: This attribute shows that this signal is not measured directly, but calculated from tentatively several other measured or calculated signals. In a vehicle, there might be alternative signals to be used from other components having a better quality, e.g. a raw signal.
- Min: This annotation indicates that the signal carries a minimum value. If, for example, a reference value computed in the software-component is below that value some dedicated actions (e.g. failure-mode) might have to be taken.
- Max: This annotation indicates that the signal carries a maximum value. If, for example, a reference value computed in the software-component is above that value some dedicated actions (e.g. failure-mode) might have to be taken.

In the meta-model this aspect is implemented by the abstract meta-class `SenderReceiverAnnotation` which represents the base class of both `SenderAnnotation` and `ReceiverAnnotation`. This relationship is depicted in figure 3.8.

Class	⟨⟨**atpObject**⟩⟩ **SenderReceiverAnnotation (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Desc.	Annotation of the data elements in a port that realizes a sender/receiver interface.			
Base Class(es)	GeneralAnnotation			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
computed	Boolean	1	aggregation	Flag whether this data element was not measured directly but instead was calculated from possibly several other measured or calculated values.

dataElement	DataElement Prototype	1	reference	The instance of data element annotated.
limitKind	LimitKind	1	aggregation	This min or max has not to be mismatched with the min- and max for data-value in a compu-method. For example, this annotation shows when the result of the calculation performed in a RunnableEntity owned by one AtomicSoftwareComponentType is transmitted to another AtomicSoftwareComponentType whose RunnableEntity will use this value as a limit, e.g. the max.power which can be used by that software-component, or the current min. slip.
processing Kind	Processing Kind	1	aggregation	

Table 3.10: SenderReceiverAnnotation

Class	⟨⟨**atpObject**⟩⟩ **SenderAnnotation**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Desc.	Annotation of a sender port, specifying properties of data elements that don't affect communication or generation of the RTE.			
Base Class(es)	SenderReceiverAnnotation			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.11: SenderAnnotation

Class	⟨⟨**atpObject**⟩⟩ **ReceiverAnnotation**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Desc.	Annotation of a receiver port, specifying properties of data elements that don't affect communication or generation of the RTE. The given attributes are requirements on the required data.			
Base Class(es)	SenderReceiverAnnotation			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
signalAge	Float	1	aggregation	The maximum allowed age of the signal since it was originally read by a sensor. This is a requirement specified on the receiver side.

Table 3.12: ReceiverAnnotation

The Min and Max annotations are valid for a certain amount of time. The value is likely to change to another valid value while the ECU is running. E.g. the maximal torque which can be requested from an engine is a typical use-case.

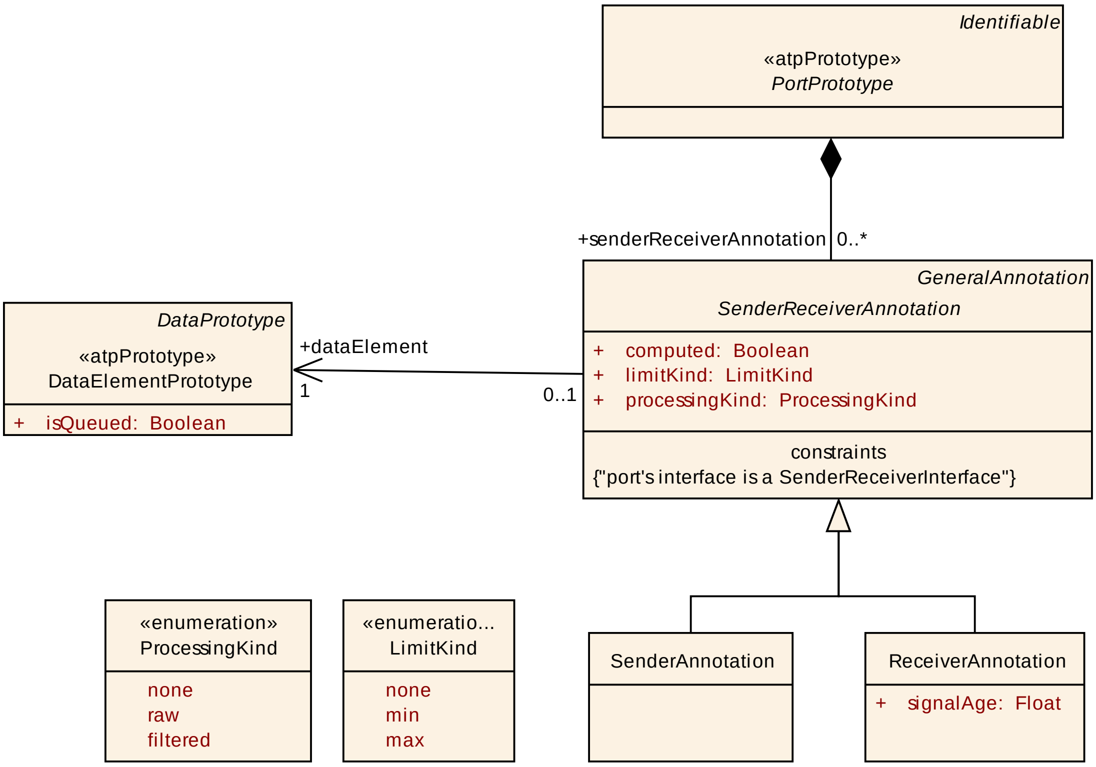


Figure 3.8: `SenderReceiverAnnotation`

This value might vary depending on e.g. the status of the climate control system. Therefore, these annotations must not be mismatched with the min and max attributes of `CompuMethods`.

The application level port annotations for sender/receiver communication have to be associated to each `DataElementPrototype` in a `PortPrototype`, e.g. there might be a ”raw” `DataElementPrototype` and a ”filtered” `DataElementPrototype` in the same `PortPrototype`!

Furthermore, if two `DataElementPrototypes` use the same application-level `PortAnnotation`, a reference from the annotation to the `DataElementPrototypes` will be established by an appropriate tool.

As shown in figure 3.8 the `PortAnnotations` for sender/receiver communication are grouped into

- processing type, indicating to some extend the direct quality of the signal,
- computed, which is just a flag or,
- limit type, showing the component expects an actual limit.

In the case of an `RPortPrototype`, the signal age of the value, carried by the associated `ConnectorPrototype`, can be specified. Each of these groups can be interpreted as a property of the signal-quality.

3.5.3 Annotation for the I/O Hardware Abstraction Layer

The attributes `BswRangeMin`, `BswRangeMax`, `BswResolution` and Unit of physical signals are currently being described by attributes of meta-class `IoHwAbstractionServerAnnotation`[4].

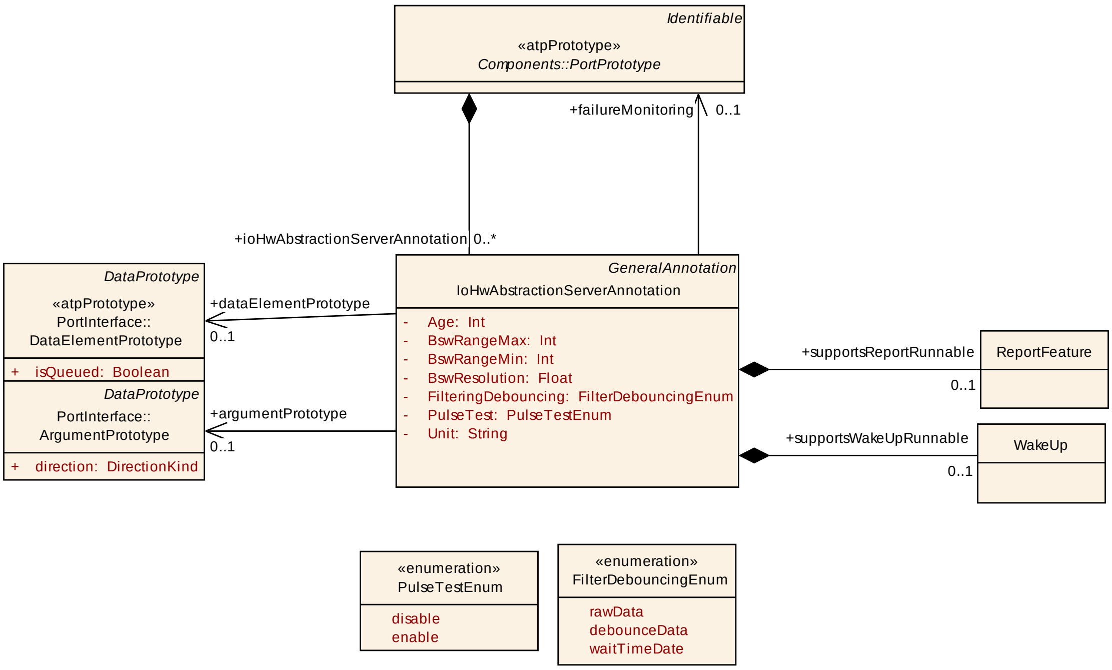


Figure 3.9: `IoHwAbstractionServerAnnotation`

Class	⟨⟨**atpObject**⟩⟩ **IoHwAbstractionServerAnnotation**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Desc.	The IoHwAbstractionPort Annotation will only be used from a sensor- or an actuator component while interacting with the IoHwAbstraction layer			
Base Class(es)	GeneralAnnotation			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
Age	Integer	1	aggregation	In case of a SET operation, the age will be interpreted as Delay while in a GET operation (input) it specifies the Lifetime of the signal within the IoHwAbstraction Layer
BswRange Max	Integer	1	aggregation	Specifies the maximum value of the Range the ECU-Signal is supposed to have
BswRange Min	Integer	1	aggregation	Specifies the maximum value of the Range the ECU-Signal is supposed to have.

---

[4] In future versions of the document, this should be expressed more in alignment to the rest of the Software Component Template by assigning `SwDataDefProps` to the `PrimitiveType` representing the physical signal that is to be exchanged over the `IoHardwareAbstraction` interface.

BswResolution	Float	1	aggregation	This value is determined by an appropriate combination of the range, the unit as well as the data-elements type, i.e. (BswRangeMax-BswRangeMin) / (2ˆ datatypelength - 1)
Filtering Debouncing	FilterDebouncing Enum	1	aggregation	This attribute is used to indicate what kind of filtering/debouncing has been put to the signal in the IoHwAbstraction layer. rawData means that no modification of the signal has been applied.This is the default value debounceData means that the signal is a mean value waitTimeData means that the signal is delivered by a GET operation after a certain amount of time
PulseTest	PulseTest Enum	1	aggregation	This attribute indicates to the connected SensorActuatorSoftwareComponentType whether the DataElementPrototype can be used to generate pulse test sequences using the IoHwAbstraction layer
Unit	String	1	aggregation	These are either electrical units like Volts (V) or time units like milliseconds (ms). The unit is set according to the ECU Input signal class which is either analogue or modulation
argument Prototype	Argument Prototype	0..1	reference	Reference to the corresponding ArgumentPrototype. The IoHwAbstractionServerAnnotation can be applied either to sender-receiver or to client-server communication. This association only applies in the latter case
dataElement Prototype	DataElement Prototype	0..1	reference	Reference to the corresponding DataElementPrototype. The IoHwAbstractionServerAnnotation can be applied either to sender-receiver or to client-server communication. This association only applies in the former case
failureMonitoring	PortPrototype	0..1	reference	This is only applicable in SET operations. If it is enabled, the IoHwAbstraction layer will monitor the result of the operation and issue an diagnostic signal. This means especially, that an additional client-server port has to be created. Tools can use this information to cross-check whether for each data-element in a SET operation with FailureMonitoring enabled an additional port is created The referenced port monitors a failure in the to be monitored data-element of the IoHwAbstraction layer. The referenced port has to be another port of the same Actuator or Sensor Component.

supports Report Runnable	Report Feature	0..1	aggregation	
supports WakeUp Runnable	WakeUp	0..1	aggregation	

Table 3.13: IoHwAbstractionServerAnnotation

Enumeration	**FilterDebouncingEnum**
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes
Enum Desc.	This element indicates to the connected Actuator Software component whether the data-element can be used to generate pulse test sequences using the IoHwAbstraction layer
Literal	***Description***
rawData	means that no modification of the signal has been applied.This is the default value
debounce Data	The signal is a mean value
waitTimeDate	The signal is delivered by a GET operation after a certain amount of time

Enumeration	**PulseTestEnum**
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes
Enum Desc.	
Literal	***Description***
disable	Disables the pulse test
enable	Enables the pulse test

This way, the `Range` and `Unit` attributes will be expressed by ordinary Datatype semantics as detailed in chapter 4.5.

Within the ECU-Abstraction Layer there are ECU-signals defined. These signals represent the electrical signals as they arrive in the microcontroller peripheral and are fetched from the registers via the MCAL. Access to the I/O Hardware Abstraction Layer is done via service interfaces, i.e. the I/O Hardware Abstraction Layer provides GET- and SET-operations at the specified service ports of a `SensorActuatorSoftwareComponentType`.

The `OperationPrototypes` provide an `ArgumentPrototype` where several annotations can be assigned to. They are depicted in the `IoHwAbstractionServerAnnotation` meta-class in figure 3.9.

A detailed description of the attributes can be found in the IoHwAbstraction Layer software specification document [13]. For example, the signal age has a very dedicated meaning in this particular interface w.r.t. a register whereas the signal age in the

`SenderReceiverAnnotation` is more generic. Especially, there is no relationship with the microcontroller peripherals.

3.5.4 Calibration Port Annotation

The `CalibrationPortAnnotation` can be used to provide more information with respect to calibration parameter prototypes of the port. The data provided at the `PortPrototype` is calibration parameters. The `CalibrationPortAnnotation` provides a reference to a particular `CalprmElementPrototype`.

Class	⟨⟨**atpObject**⟩⟩ **CalibrationPortAnnotation**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Desc.	Annotation to a port used for calibration regarding a certain CalprmElement.			
Base Class(es)	GeneralAnnotation			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
calprm Element	Calprm Element Prototype	1	reference	The instance of calprm element annotated.

Table 3.14: CalibrationPortAnnotation

The main use-case is to allow easy access to the information which calibration parameters influence the data on the `PortPrototype`.

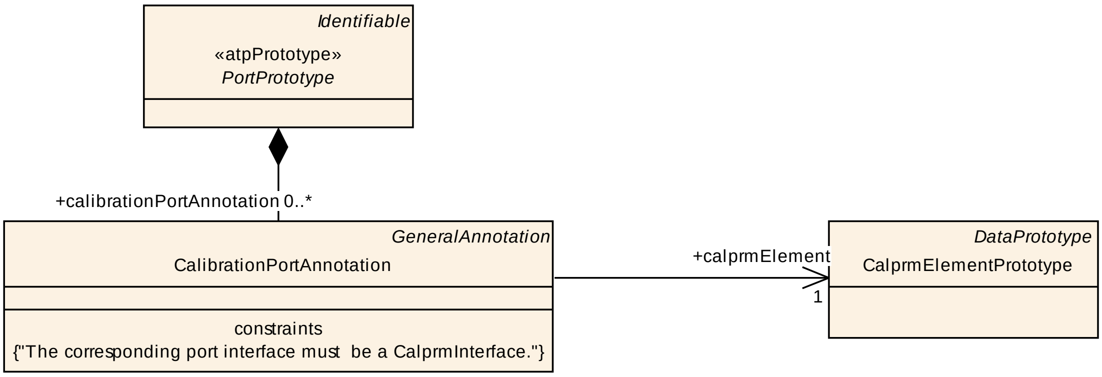


Figure 3.10: `CalibrationPortAnnotation`

3.5.5 Delegated Port Annotations

The `DelegatedPortAnnotation` is used to define the Signal Fan In or Signal Fan Out inside the `CompositionType`. This information is used to predefine and pre-check resulting communication patterns in the VFB (1:n, n:1, 1:1) if empty `CompositionTypes` are used as interface definition for sub-systems. The

`DelegatedPortAnnotation` guides either the system designer in connecting the empty `CompositionType` or the sub system designer in applying communication pattern (1:n, n:1, 1:1) inside of the `CompositionType`.

Class	⟨⟨**atpObject**⟩⟩ **DelegatedPortAnnotation**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Desc.	Annotation to a ”delegated port” to specify the Signal Fan In or Signal Fan Out inside the Composition Type.			
Base Class(es)	GeneralAnnotation			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
signalFan	SignalFan Enum	1	aggregation	Specify the Signal Fan In or Signal Fan Out inside the Composition Type

Table 3.15: DelegatedPortAnnotation

The attribute values have following definition:

- single: the internal connections in the `CompositionType` via `DelegationConnectorPrototypes` and `AssemblyConnectorPrototypes` are defined in a way that each `DataElementPrototype` present in the `SenderReceiverInterfaces` or `OperationPrototype` in the `ClientServerInterfaces` of the outer `PortPrototype` is involved in a 1:1 communication pattern only.
- nfold: The internal connections in the `CompositionType` via `DelegationConnectorPrototypes` and `AssemblyConnectorPrototoypes` are defined in a way that at least one `DataElementPrototype` present in the `SenderReceiverInterfaces` or one `OperationPrototype` in the `ClientServerInterfaces` of the outer `PortPrototype` is involved in a 1:n or n:1 communication pattern.

3.5.6 General Annotation

Besides formally specified attributes it is also possible to place textual information as provided in the abstract `GeneralAnnotation` (see figure 3.11 for an overview).

Class	⟨⟨**atpObject**⟩⟩ **GeneralAnnotation (abstract)**
Package	M2::AUTOSARTemplates::GenericStructure::CommonPatterns::Annotation
Class Desc.	This class represents textual comments (called annotations) which relate to the object in which it is aggregated. These are intended for use during the development process, to transfer information from one stage of the development process to the next one. The approach is similar to the ”yellow pads ...” This abstract class can be specialized in order to add some further formal properties.
Base Class(es)	ARObject

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
annotation Origin	String	1	aggregation	This element identifies the origin of the annotation. It is an arbitrary string since it can be an individual's name as well as the name of a tool or even the name of a process step.
annotation Text	Remark	1	aggregation	This is the text of the annotation.
label	MlData4	1	aggregation	label is used as a long designator (similar to longName) for objects which cannot be referenced.

Table 3.16: GeneralAnnotation

Class	⟨⟨**atpMixed**⟩⟩ **Remark**			
Package	M2::AUTOSARTemplates::GenericStructure::CommonPatterns::Annotation			
Class Desc.	<remark> is used for comments e.g. on the specific calibration state. The remark can be a regular paragraph or a preformatted text.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
p	MlData1	1	aggregation	Use <p> to create a paragraph for continuous texts.
verbatim	MlData5	1	aggregation	<verbatim> is a paragraph in which white-space (in particular blanks and line feeds) is obeyed. This enables basic preformatting to be carried out, which can even be displayed on simple devices. Behavior is the same as PRE in HTML .

Table 3.17: Remark

3.6 Communication of Runnables

In this section we describe the communication properties of an `AtomicSoftwareComponentType` from the point of view of a `RunnableEntity` (the concept of a `RunnableEntity` is introduced in chapter 5.2).

3.6.1 Communication Attributes

The highest level of description of information exchanged between components in an AUTOSAR system is the `PortInterfaces`, as shown in earlier sections. Such an interface however, only describes structure and does not include information about whether communication needs to be done reliably, or whether an init value exists in case the real data is not yet available.

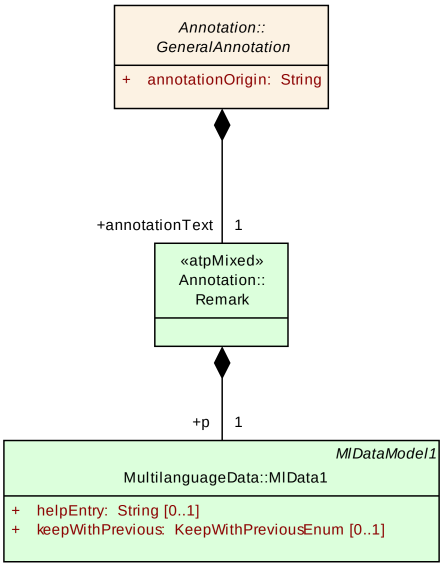


Figure 3.11: textual information in annotations

This kind of information is known only within the particular scenario the interface is used and also frequently differs depending on whether an interface is required or provided. Therefore, most communication relevant attributes are related to the ports of a component. The communication attributes are organized in a so-called communication specification (in terms of the meta-model: `ComSpec`) classes.

The model distinguishes three basic classes depending on the role (R-, P-Port or connector) as detailed below. Certain communication specifications are indirectly part of a composition: within a composition, multiple components are put to use (in form of component prototypes) and connected through assembly connectors.

Only in this particular context the assignment of the rather instance-specific communication attributes is relevant. Therefore, these ComSpec classes are attached to the assembly connectors.

Other ComSpec classes which are rather required on component type level are attached to the `PortPrototype` declarations, which in turn are part of the definition of a `ComponentType`. Nevertheless the usage of `ComSpec`s is **not** restricted to the ports of `AtomicSoftwareComponentType`.

`ComSpec`s attached to a `PortPrototype` owned by an `AtomicSwComponentType` have a direct impact on the generation of the RTE. The RTE Generator, on the other hand, does not consider the existence of `CompositionType`s.

Nevertheless, there are some cases where the definition of a `ComSpec` attached to a `PortPrototype` owned by a `CompositionSwComponentType` does make sense.

That is, in case an OEM wants to submit the definition of a `CompositionType` to a supplier for adding more details and implementing the behavior the OEM might want to point out that from the OEM's point of view `initValue`s apply for the elements of `PortInterface`s used to type the delegation `PortPrototype`s.

The idea is that the supplier takes over the `initValue`s attached to the delegation `PortPrototype`s and *copies* them to the `PortPrototype`s owned by `ComponentPrototype`s of the `CompositionType`.

The RTE Generator would still *only* take the initial values of the `PortPrototype`s of `AtomicSoftwareComponentType`s and ignore the `initValue`s at the delegation `PortPrototype`s.

Therefore, the `initValue`s of the delegation `PortPrototype` would be taken as *mere templates* for the detailing of `PortPrototype`s connected to the delegation `PortPrototype`s.

It is not required that the `initValue`s of delegated `PortPrototype` and a `PortPrototype` connected by means of a `DelegationSwConnector` match.

Although this would certainly make sense in many cases it is eventually still left to the supplier to decide on the specific `initValue`s applicable inside the `CompositionSwComponentType`.

On the other hand, a requirement that the `initValue`s defined on the surface of `CompositionType` and the inside of the `CompositionType` must be consistent in any case might effectively prevent the reuse of existing `AtomicSwComponentType`s. Sections 3.6.2 and 3.6.3 then explain the sender-receiver and client-server communication patterns with respect to the RTE, the RTE events and the corresponding communication attributes.

3.6.1.1 Communication Specification of an R-Port

Figure 3.12 shows the model of the communication attributes relevant for an R-Port.

The ComSpec attributes are collected depending on the kind of data transmitted, which means they may differ depending on whether data elements are exchanged (sender-receiver), operations are called (client-server), or even depending on whether the data-elements represent queued or non-queued data.

This is expressed in the inheritance tree of `ComSpec` classes. Each of these classes may then carry the specific attributes. An `RPortPrototype` may aggregate many `ComSpec`, possibly one for each interface element (data element or operation) the associated interface contains.

Granted, the definition of a `ComSpec` for `CalprmElementPrototype`s looks strange on first sight. A `CalprmElementPrototype` owned by a `PPortPrototype` typed by a `CalprmInterface` is not actually transmitted over any communication medium. Therefore, the term *communication* should in this case be taken with a grain of salt.

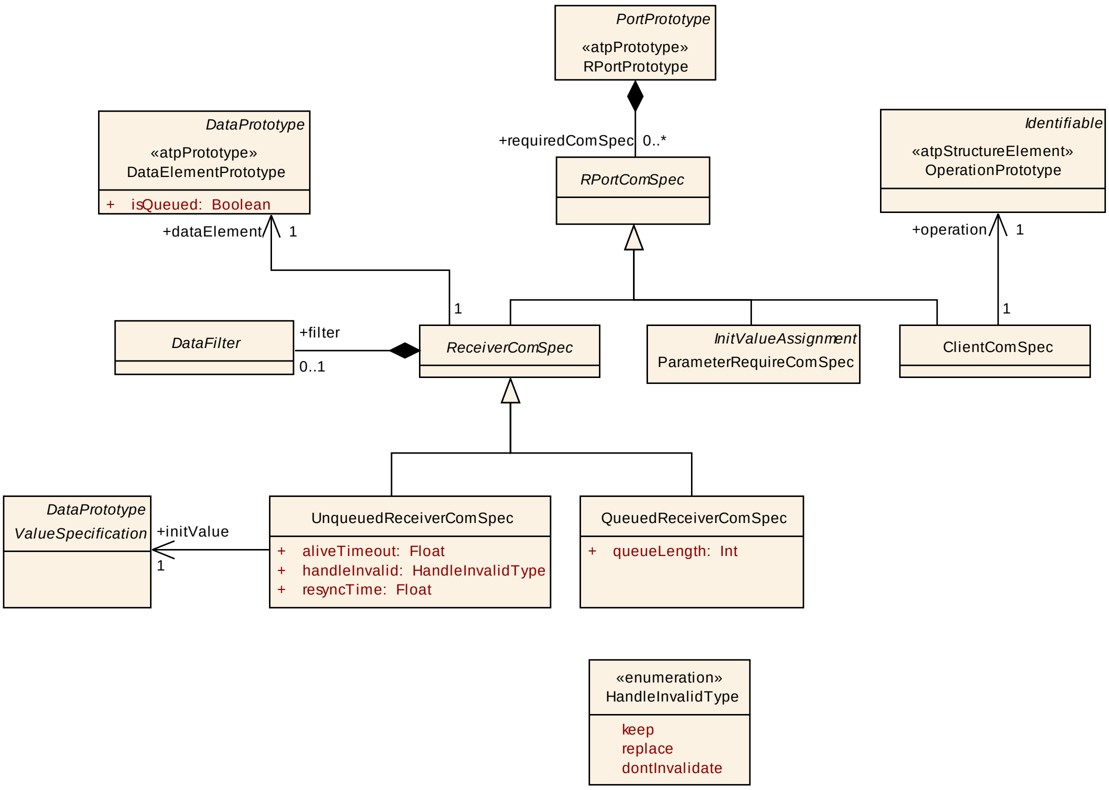


Figure 3.12: Communication attributes of `RPortPrototype`.

However, it is generally necessary to be able to define role-specific initial values for `CalprmElementPrototype`s aggregated in a `CalprmInterface`. In other words, the actual problem closely resembles the definition of initial values in the case of sender-receiver communication.

Therefore, it is only reasonable to apply the existing and well-known pattern to the definition of initial values for `CalprmElementPrototype`s aggregated in a `CalprmInterface`. The actual modeling is sketched in Figure 3.16 for provided `ParameterDataPrototype`s and in Figure 3.13 for required `ParameterDataPrototype`s. Please note that the abstract meta-class `InitValueAssignment` has been introduced to allow for the application of the same initialization mechanism to `CalprmElementPrototype`s owned by `InternalBehavior`.

Class	**⟨⟨atpObject⟩⟩ InitValueAssignment (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior:: ComponentLocalCalprm			
Class Desc.	This represents the ablity to assign an initial value to a calibration parameter.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

initValue	Value Specification	1	reference	This is the init value.
parameter	Calprm Element Prototype	1	reference	This is the parameter for which the initial value applies.

Table 3.18: InitValueAssignment

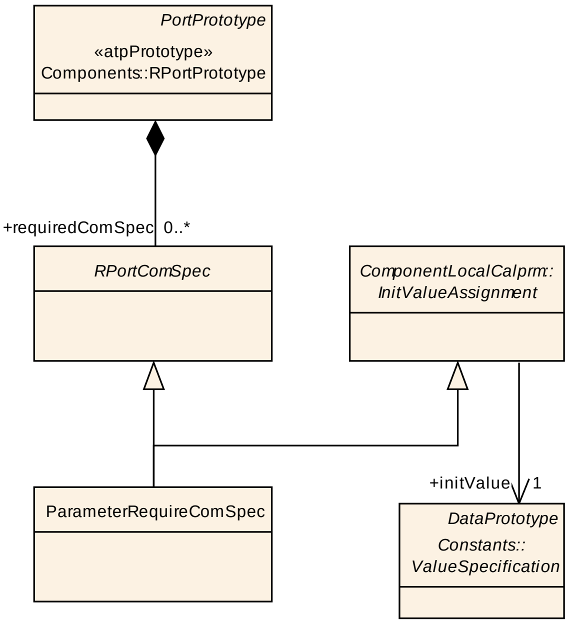


Figure 3.13: Communication attributes for calibration parameters.

The meaning of the attributes shown above is explained in the following class tables. Classes that have no attributes are not listed here.

Class	⟨⟨**atpObject**⟩⟩ **ReceiverComSpec (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Receiver specific communication attributes (R-Port and sender-receiver interface).			
Base Class(es)	RPortComSpec			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
dataElement	DataElement Prototype	1	reference	Data element these attributes belong to.
filter	DataFilter	0..1	aggregation	

Table 3.19: ReceiverComSpec

Enumeration	**HandleInvalidType**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication
Enum Desc.	Strategies of handling the reception of invalidValue.
Literal	***Description***
keep	Keep a received invalidValue. This allows handling of Signal Invalidation on RTE API level either by DataReceiveErrorEvent or return of an error code on on read access.
replace	Replace a received invalidValue. The replacement value is specified by the initValue.
dontInvalidate	Invalidation is switched off.

Class	⟨⟨**atpObject**⟩⟩ **UnqueuedReceiverComSpec**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Communication attributes specific to unqueued receiving.			
Base Class(es)	ReceiverComSpec			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
aliveTimeout	Float	1	aggregation	Specify the amount of time (in seconds) after which the software component (via the RTE) needs to be notified if the corresponding data item have not been received according to the specified timing description.
handle Invalid	Handle InvalidType	1	aggregation	Specifies strategy of handling the reception of invalidValue.
initValue	Value Specification	1	reference	Initial value to be used in case the sending component is not yet initialized. If the sender also specifies an init value the receiver's value will be used.
resync Time	Float	1	aggregation	Time allowed for resynchronization of data values after current data is lost, e.g. after an ECU reset.

Table 3.20: UnqueuedReceiverComSpec

Class	⟨⟨**atpObject**⟩⟩ **QueuedReceiverComSpec**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Communication attributes specific to queued receiving.			
Base Class(es)	ReceiverComSpec			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
queue Length	Integer	1	aggregation	Length of queue for received events.

Table 3.21: QueuedReceiverComSpec

Class	⟨⟨**atpObject**⟩⟩ **ClientComSpec**

Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Client specific communication attributes (R-Port and client-server interface).			
Base Class(es)	RPortComSpec			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
operation	Operation Prototype	1	reference	Operation these attributes belong to.

Table 3.22: ClientComSpec

3.6.1.2 Communication Specification of Data Filters

Figure 3.14 shows the model of the communication attributes relevant for defining data filters. For every r-port with sender-receiver semantics a data filter can be defined. Depending on the chosen filter, the filter specific attributes have to be defined.

The fifteen filter algorithms that are listed in the meta-model are taken from OSEK COM 3.0.2 specification that is referenced by the RTE specification. This OSEK specification states that ”filtering is only used for messages that can be interpreted as C language unsigned integer types (characters, unsigned integers and enumerations).” Therefore, filters can only be applied to values with integer datatype.

Class	**⟨⟨atpObject⟩⟩ DataFilter (abstract)**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Base class for data filters.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.23: DataFilter

Class	**⟨⟨atpObject⟩⟩ Always**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	No filtering is performed so that the message always passes.			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.24: Always

Class	**⟨⟨atpObject⟩⟩ Never**
Package	M2::AUTOSARTemplates::CommonStructure::Filter
Class Desc.	The filter removes all messages.

Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.25: Never

Class	⟨⟨**atpObject**⟩⟩ **MaskedNewEqualsX**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass messages whose masked value is equal to a specific value x (new_value&mask) == x new_value: current value of the message			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
mask	Integer	1	aggregation	mask for the new Value
x	Integer	1	aggregation	Value to compare with

Table 3.26: MaskedNewEqualsX

Class	⟨⟨**atpObject**⟩⟩ **MaskedNewDiffersX**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass messages whose masked value is not equal to a specific value x (new_value&mask) != x new_value: current value of the message			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
mask	Integer	1	aggregation	mask for the new Value
x	Integer	1	aggregation	Value to compare with

Table 3.27: MaskedNewDiffersX

Class	⟨⟨**atpObject**⟩⟩ **NewIsEqual**
Package	M2::AUTOSARTemplates::CommonStructure::Filter
Class Desc.	Pass messages which have not changed. newValue == oldValue new_value: current value of the message old_value: last value of the message (initialised with the initial value of the message, updated with new_value if the new message value is not filtered out)
Base Class(es)	DataFilter

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.28: NewIsEqual

Class	⟨⟨**atpObject**⟩⟩ **NewIsDifferent**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass messages which have changed. newValue != oldValue new_value: current value of the message old_value: last value of the message (initialised with the initial value of the message, updated with new_value if the new message value is not filtered out)			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.29: NewIsDifferent

Class	⟨⟨**atpObject**⟩⟩ **MaskedNewEqualsMaskedOld**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass messages where the masked value has not changed. (new_value&mask) ==(old_value&mask) new_value: current value of the message old_value: last value of the message (initialised with the initial value of the message, updated with new_value if the new message value is not filtered out)			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
mask	Integer	1	aggregation	mask for old and new value

Table 3.30: MaskedNewEqualsMaskedOld

Class	⟨⟨**atpObject**⟩⟩ **MaskedNewDiffersMaskedOld**
Package	M2::AUTOSARTemplates::CommonStructure::Filter
Class Desc.	Pass messages where the masked value has changed. (new_value&mask) !=(old_value&mask) new_value: current value of the message old_value: last value of the message (initialised with the initial value of the message, updated with new_value if the new message value is not filtered out)
Base Class(es)	DataFilter

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
mask	Integer	1	aggregation	mask for old and new value

Table 3.31: MaskedNewDiffersMaskedOld

Class	⟨⟨**atpObject**⟩⟩ **NewIsWithin**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass a message if its value is within a predefined boundary. min <= new_value <= max			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
max	Integer	1	aggregation	Value to specify the upper boundary
min	Integer	1	aggregation	Value to specify the lower boundary

Table 3.32: NewIsWithin

Class	⟨⟨**atpObject**⟩⟩ **NewIsOutside**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass a message if its value is outside a predefined boundary. (min > new_value) OR (new_value > max)			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
max	Integer	1	aggregation	Value to specify the upper boundary
min	Integer	1	aggregation	Value to specify the lower boundary

Table 3.33: NewIsOutside

Class	⟨⟨**atpObject**⟩⟩ **NewIsGreater**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass a message if its value has increased. new_value > old_value new_value: current value of the message old_value: last value of the message (initialised with the initial value of the message, updated with new_value if the new message value is not filtered out)			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.34: NewIsGreater

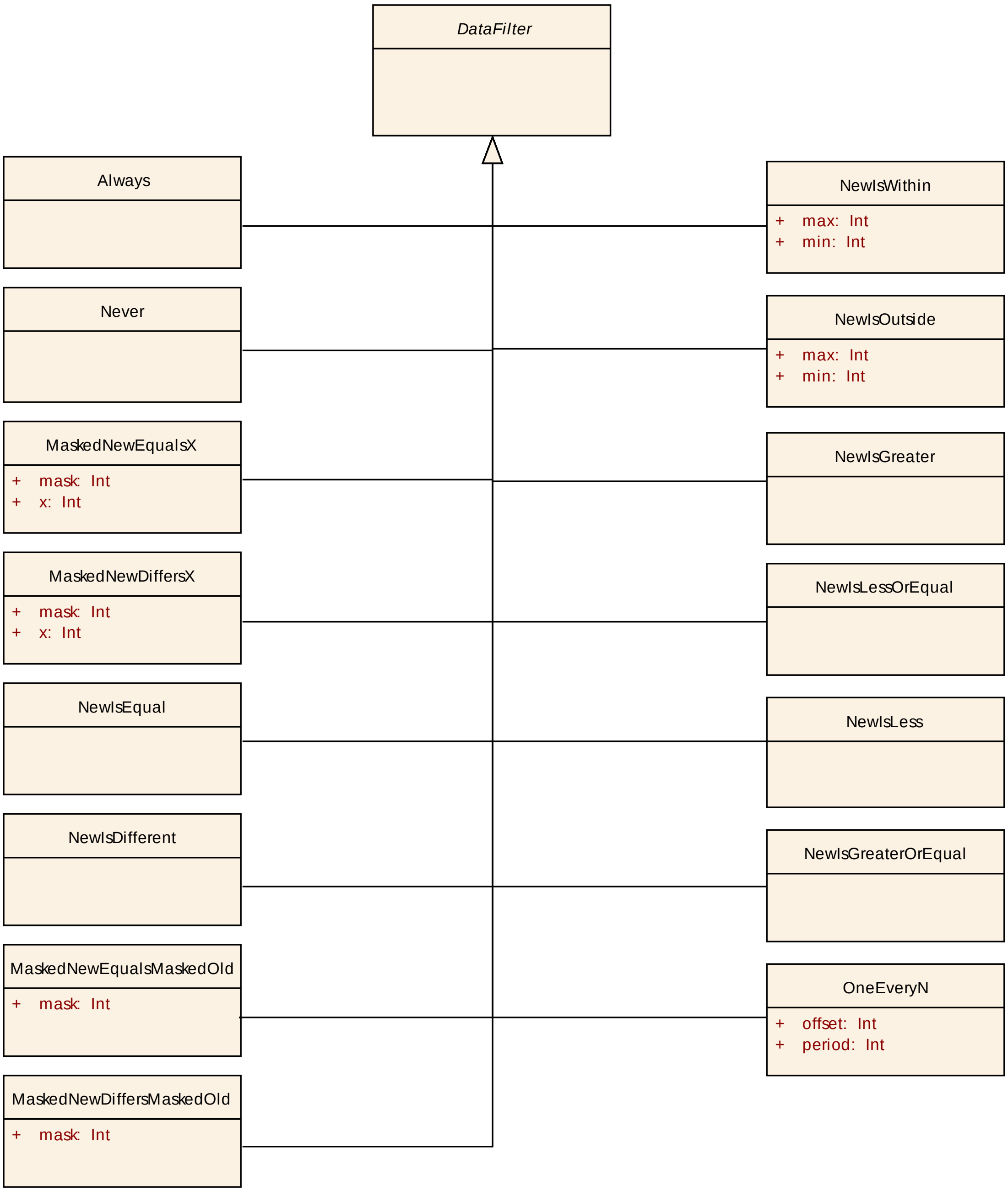


Figure 3.14: `DataFilter` and its communication attributes.

Class	⟨⟨**atpObject**⟩⟩ **NewIsLessOrEqual**
Package	M2::AUTOSARTemplates::CommonStructure::Filter
Class Desc.	Pass a message if its value has not increased. new_value <= old_value new_value: current value of the message old_value: last value of the message (initialised with the initial value of the message, updated with new_value if the new message value is not filtered out)

Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.35: NewIsLessOrEqual

Class	⟨⟨**atpObject**⟩⟩ **NewIsLess**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass a message if its value has decreased. new_value < old_value new_value: current value of the message old_value: last value of the message (initialised with the initial value of the message, updated with new_value if the new message value is not filtered out)			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.36: NewIsLess

Class	⟨⟨**atpObject**⟩⟩ **NewIsGreaterOrEqual**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass a message if its value has not decreased. new_value >= old_value new_value: current value of the message old_value: last value of the message (initialised with the initial value of the message, updated with new_value if the new message value is not filtered out)			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.37: NewIsGreaterOrEqual

Class	⟨⟨**atpObject**⟩⟩ **OneEveryN**			
Package	M2::AUTOSARTemplates::CommonStructure::Filter			
Class Desc.	Pass a message once every N message occurrences. Algorithm: occurrence % period == offset Start: occurrence = 0. Each time the message is received or transmitted, occurrence is incremented by 1 after filtering. Length of occurrence is 8 bit (minimum).			
Base Class(es)	DataFilter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

offset	Integer	1	aggregation	specifies the initial number of messages to occur before the first message is passed
period	Integer	1	aggregation	specifies number of messages to occur before the message is passed again

Table 3.38: OneEveryN

3.6.1.3 Communication Specification of a P-Port

In analogy to the previous section, figure 3.15 shows the attribute classes relevant for a P-Port.

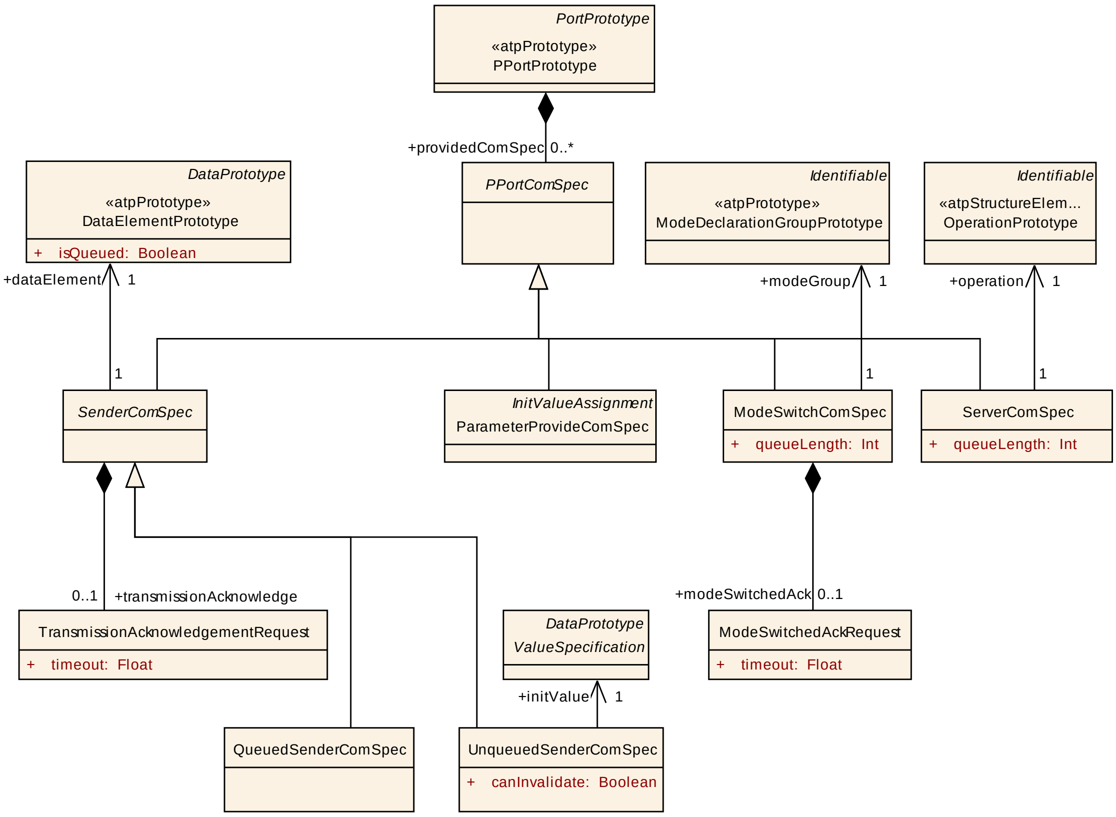


Figure 3.15: Communication attributes of `PPortPrototype`.

The same concept is applied here: a tree of ComSpec classes allows specification of such attributes on the different abstraction layers. Here are the new classes.

Class	**⟨⟨atpObject⟩⟩ SenderComSpec (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Communication attributes for a sender port (P-Port and sender-receiver interface).			
Base Class(es)	PPortComSpec			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

dataElement	DataElement Prototype	1	reference	Data element these quality of service attributes apply to.
transmissionAcknowledge	TransmissionAcknowledgement Request	0..1	aggregation	Requested transmission acknowledgement for data element.

Table 3.39: SenderComSpec

Class	⟨⟨**atpObject**⟩⟩ **TransmissionAcknowledgementRequest**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Requests transmission acknowledgement that data has been sent successfully. Success/failure is reported via a SendPoint of a Runnable.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
timeout	Float	1	aggregation	Number of seconds before an error is reported or in case of allowed redundancy, the value is sent again.

Table 3.40: TransmissionAcknowledgementRequest

Class	⟨⟨**atpObject**⟩⟩ **UnqueuedSenderComSpec**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Communication attributes specific to distribution of data (P-Port, sender-receiver interface and data element carries ”data” opposed to carrying an ”event”).			
Base Class(es)	SenderComSpec			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
canInvalidate	Boolean	1	aggregation	Flag whether the component can actively invalidate data.
initValue	Value Specification	1	reference	Init value to be sent if sender component is not yet fully initialized, but receiver needs data already.

Table 3.41: UnqueuedSenderComSpec

Class	⟨⟨**atpObject**⟩⟩ **QueuedSenderComSpec**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Communication attributes specific to distribution of events (P-Port, sender-receiver interface and data element carries an ”event”).			
Base Class(es)	SenderComSpec			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.42: QueuedSenderComSpec

Class	⟨⟨**atpObject**⟩⟩ **ServerComSpec**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Communication attributes for a server port (P-Port and client-server interface).			
Base Class(es)	PPortComSpec			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
operation	Operation Prototype	1	reference	Operation these communication attributes apply to.
queue Length	Integer	1	aggregation	Length of call queue on the server side. The queue is implemented by the RTE.

Table 3.43: ServerComSpec

Class	⟨⟨**atpObject**⟩⟩ **ModeSwitchComSpec**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Communication attributes for both sender /server port (P-Port and sender-receiver interface).			
Base Class(es)	PPortComSpec			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
mode Group	ModeDeclaration Group Prototype	1	reference	Mode Declaration Group (of the same Port Interface) to which these communication attributes apply.
mode Switched Ack	Mode Switched AckRequest	0..1	aggregation	
queue Length	Integer	1	aggregation	Length of call queue on the server side. The queue is implemented by the RTE. The value must be greater or equal to 0. Setting the value of queueLength to 0 implies non-queued communication.

Table 3.44: ModeSwitchComSpec

Class	⟨⟨**atpObject**⟩⟩ **ModeSwitchedAckRequest**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Requests acknowledgements that a mode switch has been proceeded successfully			
Base Class(es)	ARObject			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
timeout	Float	1	aggregation	Number of seconds before an error is reported or in case of allowed redundancy, the value is sent again.

Table 3.45: ModeSwitchedAckRequest

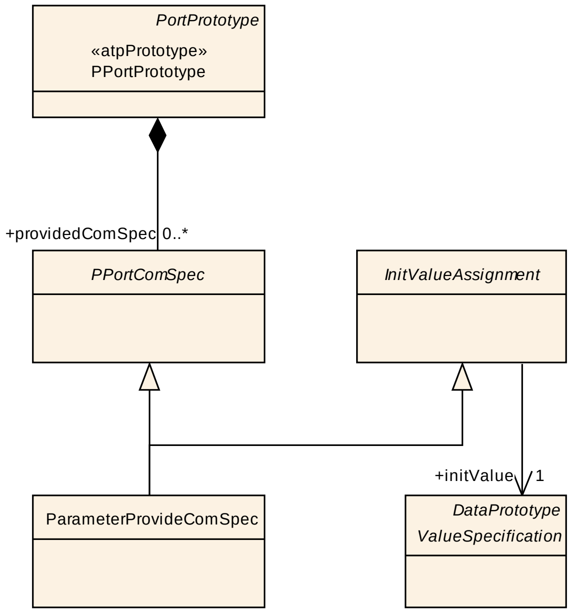


Figure 3.16: Communication attributes for calibration parameters.

3.6.2 Runnables and Sender Receiver Communication

This section describes the sender-receiver communication relevant attributes of a software-component, which influence the behavior and API of the AUTOSAR RTE. Furthermore, the possible interaction patterns for application of the sender-receiver paradigm are explained, namely:

1. Data-access in a cat. 1 `RunnableEntity`,
2. explicit sending,
3. the `DataSendCompletedEvent`: dealing with the success/failure of an explicit send, and
4. the `DataReceivedEvent`: responding to the reception of data
5. the `DataReceiveErrorEvent`: notifying an error concerning the reception of data.

3.6.2.1 Terminology

The AUTOSAR meta-model foresees two different approaches for sender-receiver communication. These are described in detail in chapters 3.6.2.2 and 3.6.2.3. However, it turned out that it is rather cumbersome to discuss issues of communication approaches directly on the basis of meta-classes and their attributes.

Therefore, it seems appropriate to introduce a dedicated terminology for this purpose. The approach eventually selected was originally introduced by the contributors to the RTE specification.

This terminology proposes to use the term ”implicit” for communication based on Data-Access (for more information about details of this approach please consult chapter 3.6.2.2) and ”explicit” for communication based on Data-Points (please refer to chapter 3.6.2.3).

The motivation for the differentiation between ”implicit” and ”explicit” was originally the characteristics of the RTE specification that foresaw an API for handling a `DataSendPoint` or `DataReceivePoint` in contrast to the Data-Access that was supposed to be part of the function signature (therefore, no API was required) of a specific `RunnableEntity`.

Although the specification of the RTE changed in the meantime (and the original motivation no longer applies) it turned out that the terminology based on ”implicit” and ”explicit” communication” was already widely used within AUTOSAR.

As no consensus could be reached over alternative proposals this terminology approach is taken over by this document as well.

3.6.2.2 Data Access

The `InternalBehavior` may specify that a `RunnableEntity` needs read-access (respectively write-access) to the `DataElementPrototypes` of an `RPortPrototype` (respectively `PPortPrototype`). The usage of this access mechanism to the `DataElementPrototypes` is appropriate for cat. 1 `RunnableEntities` only, which guarantees finite response time (opposed to waiting for data for instance).

Please note that from the formal point of view read-access is implemented by means of the meta-class `DataReadAccess` while the write-access is defined by means of the corresponding meta-class `DataWriteAccess`. This aspect is depicted in figure 3.17.

Class	⟨⟨**atpObject**⟩⟩ **DataReadAccess**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Data Elements			
Class Desc.	The presence of a DataReadAccess implies that a RunnableEntity needs access to a DataElementPrototype in an RPortPrototype. The RunnableEntity will not modify the contents of the data but only read the information. The RunnableEntity expects that the contents of this data does NOT change during the entire duration of its execution.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
dataElement	DataElementPrototype	1	instanceRef	The data element that is going to be read by this runnable.

Table 3.46: DataReadAccess

Class	⟨⟨**atpObject**⟩⟩ **DataWriteAccess**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Data Elements			
Class Desc.	The presence of a DataWriteAccess means that the RunnableEntity will potentially modify the DataElementPrototype in the PPortPrototype. The RunnableEntity has free access to the DataElementPrototype while it is running. The RunnableEntity has the responsibility to make sure that the DataElementPrototype is in a consistent state when it returns. When using DataWriteAccess the new values of the DataElementPrototype is not made available via the communication infrastructure before the RunnableEntity returns (exits the "Running" state).			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
dataElement	DataElement Prototype	1	instanceRef	The data element that is going to be written to by this runnable.

Table 3.47: DataWriteAccess

3.6.2.3 Explicit Sending and Receiving

A `RunnableEntity` can also have `DataSendPoints`. Using an `instanceRef` association, these eventually reference a `DataElementPrototype` in the context of a `PPortPrototype`, owned by the `AtomicSoftwareComponentType` associated with the `RunnableEntity`.

More precisely, as the `RunnableEntity` is owned by an `InternalBehavior` referencing an `AtomicSoftwareComponentType`, the `PPortPrototype` in the `instanceRef.context` needs to be owned by this specific `AtomicSoftwareComponentType`, and the `DataElementPrototype` in the `instanceRef.target` needs to be owned by the `SenderReceiverInterface` being implemented by the `PPortPrototype`.

As opposed to the `DataWriteAccess`:

- Using the `DataSendPoint`, the `RunnableEntity` needs to explicitly "send" through an API; when using a `DataWriteAccess`, the `RunnableEntity` only needs to modify the value of certain variables.
- Using `DataSendPoint`, the Runnable can decide to "send" an arbitrary number of times; when using `DataWriteAccess` the new values of the `DataElementPrototype` is not made available before the `RunnableEntity` returns (exits the "Running" state).
- The presence of a `DataSendPoint` per definition lets the corresponding `RunnableEntity` attain cat. 1B.

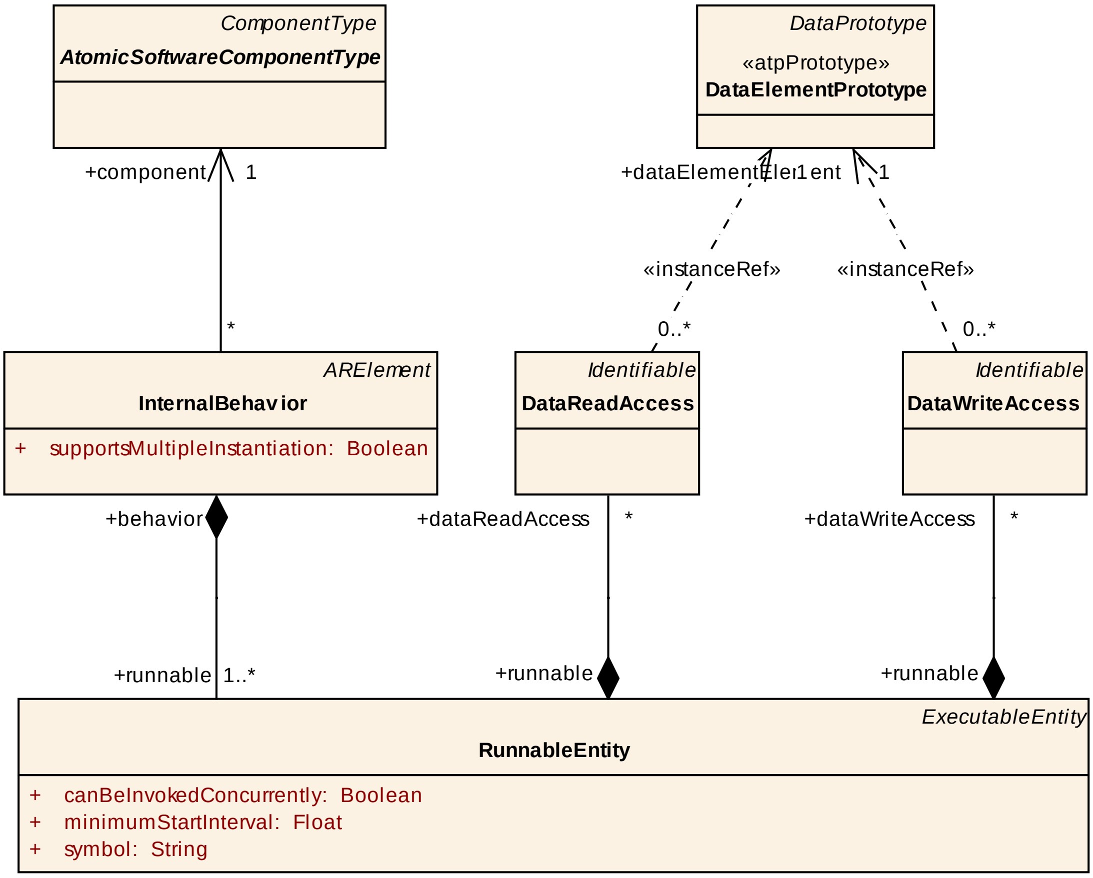


Figure 3.17: DataReadAccess and DataWriteAccess

Class	⟨⟨**atpObject**⟩⟩ **DataSendPoint**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Data Elements			
Class Desc.	A DataSendPoint specifies that a RunnableEntity explicitly sends a certain DataElementPrototype.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
dataElement	DataElementPrototype	1	instanceRef	The data element that is sent by this runnable.

Table 3.48: DataSendPoint

In analogy to explicitly sending data it is also possible to define explicit polling for new available data through a `DataReceivePoint` as shown in figure 3.19.

By using a `DataReceivePoint` instead of `DataReadAccess` the constraining access to the referenced data element - other `RunnableEntities` must not change the `DataElementPrototype` during the read execution - is limited to a short, well-defined amount of time.

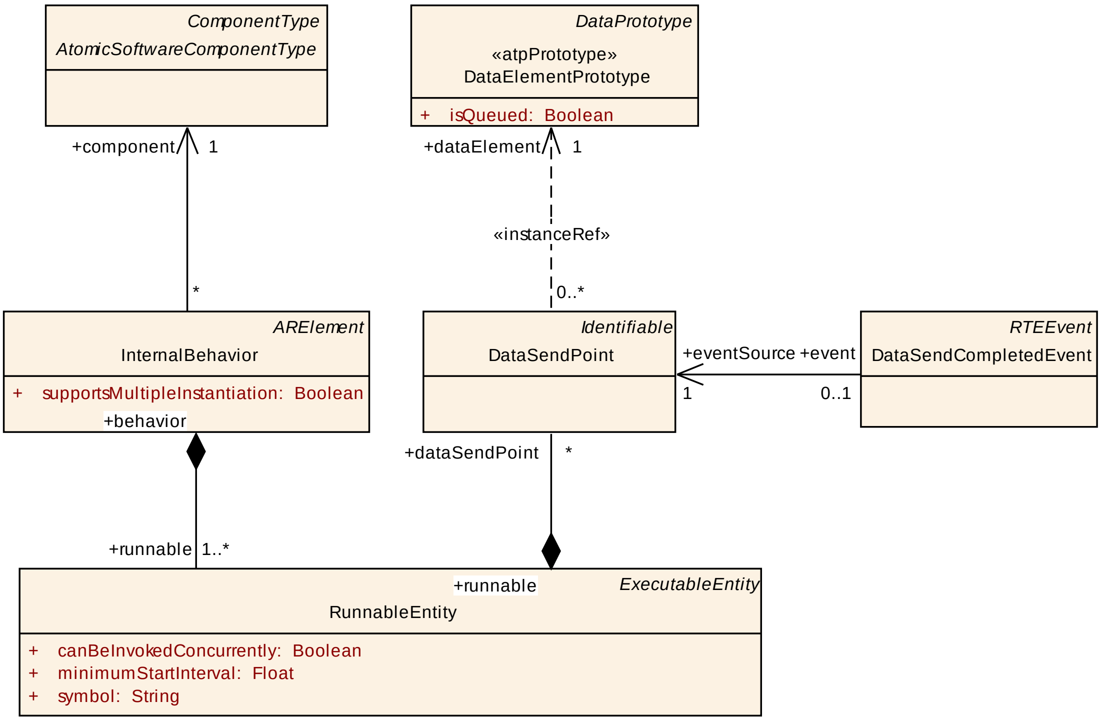


Figure 3.18: DataSendPoint

Therefore, category 1 `RunnableEntities` may also have `DataReceivePoints` and consequently become `RunnableEntities` of category 1B.

Class	⟨⟨**atpObject**⟩⟩ **DataReceivePoint**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Data Elements			
Class Desc.	A DataReceivePoint allows a RunnableEntity to explicitly query for received information, thereby blocking write access to the same information only for a very brief period.			
Base Class(es)	Identifiable			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
dataElement	DataElementPrototype	1	instanceRef	The data element to be explicitly read.

Table 3.49: DataReceivePoint

Please note that it would in general be possible to combine a `DataReceivePoint` with a `WaitPoint` in the scope of a particular `RunnableEntity`. This would allow for a call to a blocking receive routine implemented by the RTE. The `timeout` attribute of meta-class `WaitPoint` can be used to specify the time until the blocking call expires.

Please note however, that in this case (in response to the presence of a `WaitPoint`) the `RunnableEntity` becomes category 2.

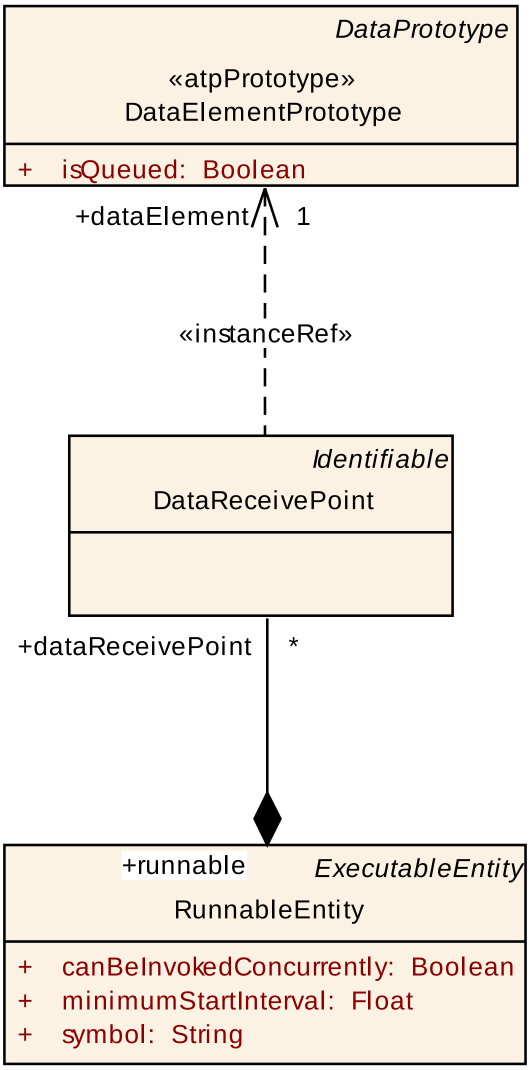


Figure 3.19: Definition of an explicit request to receive data

3.6.2.4 DataSendCompletedEvent

The `DataSendPoint` also allows for the definition of a `DataSendCompletedEvent`, as shown in figure 3.18. This event occurs when the data has been sent successfully or when an error has occurred during sending.

This feature can only be used, when the `AtomicSoftwareComponentType` describes the meaning of success or failure of the send operation.

In particular, via a `ComSpec` class different acknowledgment requests (in this case: successful transmission) can be attached to a `PPortPrototype`, as is shown in the left part of figure 3.15.

This will configure the RTE that when data is sent, it will try to obtain the specified acknowledgment, possibly by waiting a certain timeout period.

Class	⟨⟨**atpObject**⟩⟩ **DataSendCompletedEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	The event is raised when the referenced data elements have been sent or an error occurs.			
Base Class(es)	RTEEvent			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
event Source	DataSend Point	1	reference	Data send point that triggers the event.

Table 3.50: DataSendCompletedEvent

3.6.2.5 DataReceivedEvent

Similarly, a receiver is notified through the same event mechanism when a `DataElementPrototype` is received. As shown in figure 3.20, the `DataReceivedEvent` is directly associated with the corresponding data element.

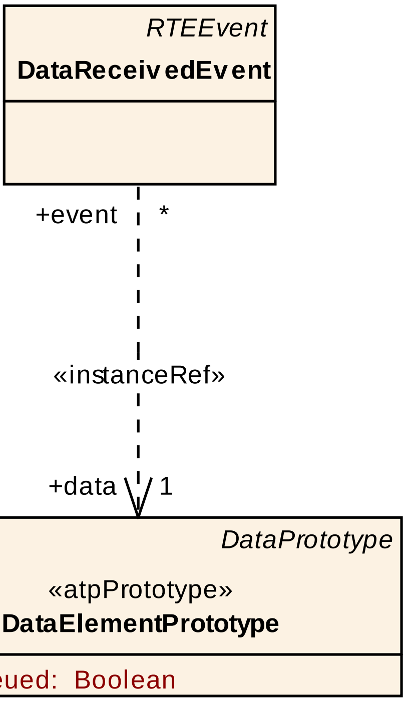


Figure 3.20: Receiver is notified by an event when new data has arrived

Class	⟨⟨**atpObject**⟩⟩ **DataReceivedEvent**
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events
Class Desc.	The event is raised when the referenced data elements are received.

Base Class(es)	RTEEvent			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
data	DataElement Prototype	1	instanceRef	Data element referenced by event

Table 3.51: DataReceivedEvent

3.6.2.6 DataReceiveErrorEvent

A receiver is notified of `DataReceiveErrorEvent` through the activation of its `RunnableEntity` which is referenced by this `RTEEvent`. A `DataReceiveErrorEvent` includes a reference to a `DataElementPrototype` and is raised by the RTE when an error concerning the reception of the referenced data is detected by the COM [5] layer. The following cases present some situations which will cause the RTE to raise a `DataReceiveErrorEvent`:

- the RTE receives a signal-outdated notification from the COM layer when a monitored periodic signal is not received in time. The COM layer monitors the validity of the signal's value based on the value of the `aliveTimeout` attribute of `ReceiverComSpec` referencing the `DataElementPrototype` associated with the signal. If the time elapsed since the last update of a signal's value exceeds its `aliveTimeout` then the COM layer notifies the RTE of a signal outdated error.
- The RTE receives a signal invalid notification from the COM layer when this latter detects that an incoming signal has the predefined 'invalid' value.

This `RTEEvent` is used by the RTE to activate `RunnableEntities` which handle the above-mentioned errors. The error code will be made available to the activated `RunnableEntity` through the appropriate RTE API function.

This `RTEEvent` cannot be associated with a `WaitPoint`. It can only be used for the receiver component in a sender-receiver communication and in release 2.0 (and newer) its data reference is restricted to `DataElementPrototypes` with their `isQueued` attribute set to false.

As shown in figure 3.21, the `DataReceiveErrorEvent` is directly associated with the corresponding `DataElementPrototype` and references the `RunnableEntity` that is activated due to the occurrence of this `RTEEvent`.

Class	⟨⟨**atpObject**⟩⟩ **DataReceiveErrorEvent**
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events
Class Desc.	This event is raised by the RTE when the Com layer detects and notifies an error concerning the reception of the referenced data element.

---

[5] In case of internal communication the RTE is not enforced to use the COM layer. It is also possible to implement the required behavior directly in the RTE [1].

Base Class(es)	RTEEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
data	DataElement Prototype	1	instanceRef	Data element referenced by event

Table 3.52: DataReceiveErrorEvent

3.6.3 Runnables and Client Server Communication

3.6.3.1 Invoking an Operation

A `RunnableEntity` invokes an operation via an `RPortPrototype` of the enclosing `ComponentPrototype` typed by a particular `AtomicSoftwareComponentType`. Note that the operation itself can be invoked either ”synchronously” or ”asynchronously”.

In the majority of cases the operation will be invoked at a different `ComponentPrototype` but in general it would be possible to invoke an operation on the very same `ComponentPrototype` as well. The decision whether a specific operation is called synchronously or asynchronously needs to be specified in the formal description of the corresponding `AtomicSoftwareComponentType`, namely in the context of an `InternalBehavior` (see figure 3.22 for more details).

In case of a synchronous operation invocation the particular `RunnableEntity` merely needs a `SynchronousServerCallPoint` (see figure 3.22). The other case is a bit more complex because it is necessary to specify how to respond to a notification about the completion of the corresponding operation.

This is done using the generic `RTEEvent` mechanism: the notification about an asynchronously executed operation being complete is implemented as an `AsynchronousServerCallReturnsEvent`. Therefore, if an `AsynchronousServerCallReturnsEvent` is raised the RTE can either trigger the execution of a specific `RunnableEntity` or the `AtomicSoftwareComponentType` can implement a `WaitPoint` that blocks the execution of the calling runnable until the `AsynchronousServerCallReturnsEvent` is recognized.

For example, let’s consider the case of an asynchronous call to a remote operation where the RTE is supposed to trigger a specific `RunnableEntity` when the operation completes. The description of the corresponding AtomicSoftwareComponentType would typically contain the following elements:

1. The `AtomicSoftwareComponentType` contains an `RPortPrototype` ’myPort’ typed by a `PortInterface` that in turn contains the definition of an `OperationPrototype` ’remoteOperation’.

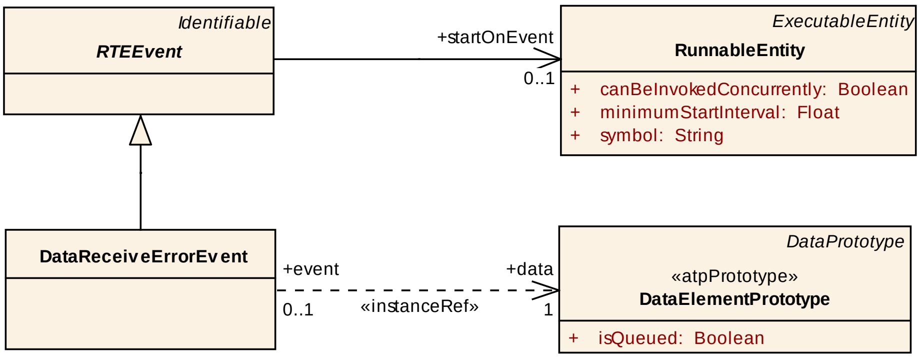


Figure 3.21: DataReceiveErrorEvent references a Runnable and a DataElementPrototype

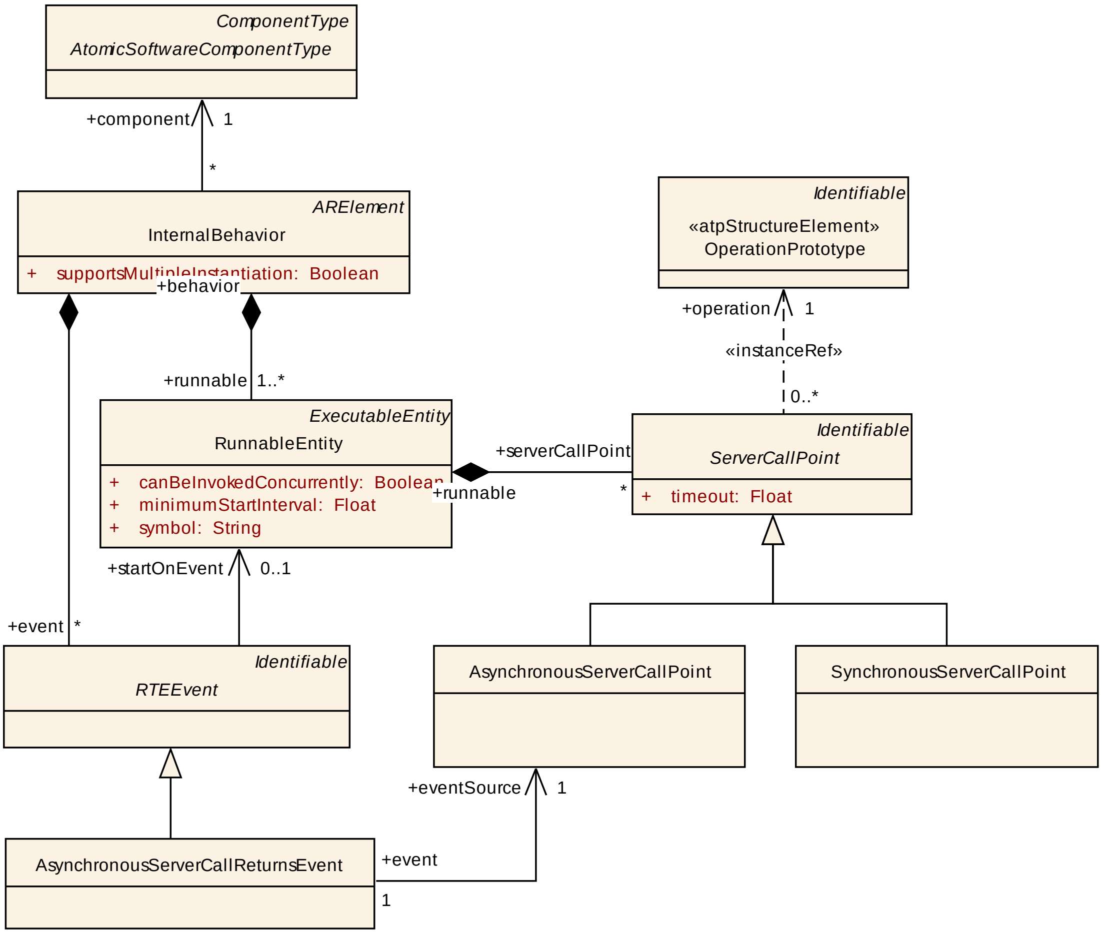


Figure 3.22: Model of a server call point.

2. The `AtomicSoftwareComponentType`'s `InternalBehavior` contains at least two `RunnableEntities`: the `RunnableEntity` 'main' is supposed to invoke the operation; the `RunnableEntity` 'callback' is the one that should be called when the operation completes.

3. The description of the `RunnableEntity` 'main' contains an `AsynchronousServerCallPoint` 'invokeMyOperation' referencing the respective `OperationPrototype` in the `PortInterface` used to type the `PortPrototype` 'myPort'. This implies that the `RunnableEntity` is allowed to invoke this operation asynchronously.
4. The description of the `AtomicSoftwareComponentType` includes an `AsynchronousServerCallReturnsEvent` 'myOperationReturns' which references the previously defined `AsynchronousServerCallPoint` 'invokeMyOperation' out of `RunnableEntity` 'main'.
5. The description of the `AsynchronousServerCallReturnsEvent` 'myOperationReturns' references the `RunnableEntity` 'callback', indicating that the RTE should trigger the execution of this Runnable when 'myOperationReturns' is raised.

Class	⟨⟨**atpObject**⟩⟩ **ServerCallPoint (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::ServerCall			
Class Desc.	When a runnable has a serverCallPoint, it has the possibility to invoke any of the operations of a specific rport of the component.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
operation	Operation Prototype	1	instanceRef	The operation that is called by this runnable.
timeout	Float	1	aggregation	Time in seconds before the server call times out and returns with an error message. It depends on the call type (synchronous or asynchronous) how this is reported.

Table 3.53: ServerCallPoint

Class	⟨⟨**atpObject**⟩⟩ **AsynchronousServerCallPoint**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::ServerCall			
Class Desc.	An asynchronous server call-point is used for asynchronous invocation of an operation prototype. It is associated with AsynchronousServerCallReturnsEvent, this RTEEvent notifies the completion of the required operation or a timeout, this event can be waited for or it can lead to the invocation of a runnable. IMPORTANT: a server-call-point cannot be used concurrently. Once the client runnable has made the invocation, the server-call-point cannot be used until the call returns (or an error occurs!) at which point the server call-point becomes available again...			
Base Class(es)	ServerCallPoint			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.54: AsynchronousServerCallPoint

Class	⟨⟨**atpObject**⟩⟩ **SynchronousServerCallPoint**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::ServerCall			
Class Desc.	This means that the runnable will block for a response from the server.			
Base Class(es)	ServerCallPoint			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**

Table 3.55: SynchronousServerCallPoint

Class	⟨⟨**atpObject**⟩⟩ **AsynchronousServerCallReturnsEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	This event is raised when an asynchronous server call is finished.			
Base Class(es)	RTEEvent			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
event Source	Asynchronous ServerCall Point	1	reference	The referenced server call point

Table 3.56: AsynchronousServerCallReturnsEvent

3.6.3.2 Providing an Implementation of an Operation

A software-component can define an `OperationInvokedEvent` for each operation inside one of the server P-Ports. This way a Runnable may respond to such an invocation through the generic event handling mechanisms described above (as formally expressed in figure 3.23).

Class	⟨⟨**atpObject**⟩⟩ **OperationInvokedEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	The OperationInvokedEvent references the OperationPrototype invoked by the client.			
Base Class(es)	RTEEvent			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
operation	Operation Prototype	1	instanceRef	The operation to be executed as the consequence of the event.

Table 3.57: OperationInvokedEvent

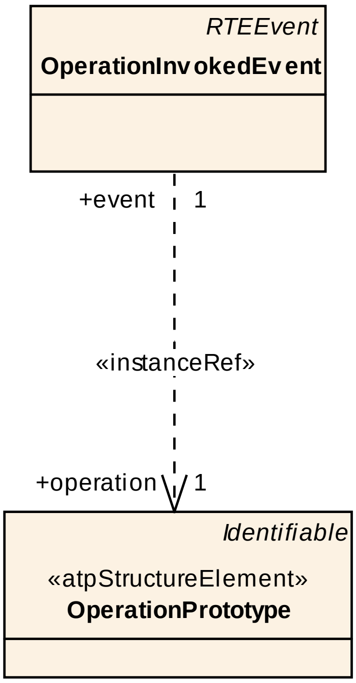


Figure 3.23: The `OperationInvokedEvent` references the operation that was called by a client.

4 Data Types and Data Semantics

4.1 Introduction

In the context of defining data types, the AUTOSAR concept distinguishes between different levels of abstraction as depicted in figure 4.1.

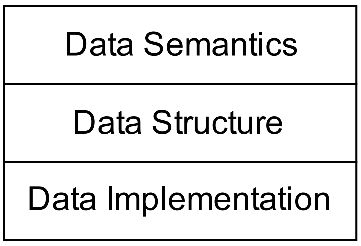


Figure 4.1: Levels of abstraction

The abstraction level called *Data Structure* is the common level at which Software Interface Definition Languages (like OMG IDL) specify a data type. Typically, a set of primitive data types (such as *int* and *floats*) is defined. On top of this, it is usually possible to build various structures with these primitive types.

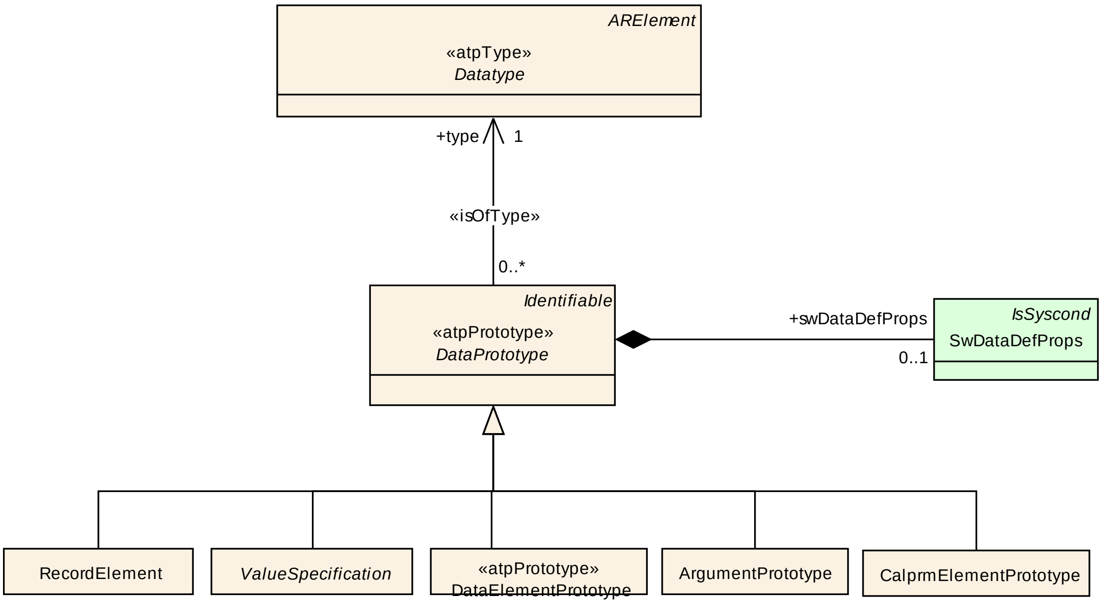


Figure 4.2: Data type usage

The *Data-Implementation* level is the implementation of Data-Structures on bits and bytes. The mapping of a given Data-Structure on a Data-Implementation depends on the medium on which the data is transported. For example, a typical 16-bit unsigned integer might look very different when sent over CAN, when seen by a software-component on a *big-endian* 32-bit machine or as seen by a software-component on a *little-endian* 16-bit processor.

Conversion between several Data-Implementations of the same Data-Structure might be necessary in case of communication between components on different ECUs. AUTOSAR COM [14] is responsible for this. It implies that the configuration depends on the exact Data-Structures that are transmitted between components.

AUTOSAR COM might need to convert a 16-bit integer between *little-endian* and *big-endian* representations; whereas an array of 16 bits does not need to be swapped even if the endianness changes. In case of intra-ECU communication byte order conversion is not necessary, since the software-components reside on the same machine.

The *Data-Semantics* finally are an additional layer of information that at least partly also has an impact on the RTE. For example, data-semantics describe how the numerical values stored in the data-structure can be mapped onto physical quantities. This is not expected to be of relevance for the RTE. On the other hand, data-semantics also defines signal invalidation that directly impacts the RTE implementation.

The description of the *Data Structure* level is contained in chapter 4.4. It explains what kinds of `Datatype` are available at this level within AUTOSAR and how new data types can be constructed.

The following chapter 4.5 deals with the optional *Data-Semantics* used to describe the correct interpretation of the values stored in the *Data-Structures*.

The *Data Implementation* level is not necessarily described in the scope of this document but depends on the medium on which the *Data-Structure* is used. Note that in particular for measurement and calibration this can be specified using the meta-class `BaseType`.

4.2 About Meta-Model Data Types

The representation of the concept of a data type within the AUTOSAR concept is implemented by means of the meta-class `Datatype`. It is taken as the base class for mainly two specializations, `PrimitiveType` and `CompositeType`. The latter, however, are taken as base classes for an even finer breakdown of the data type diversity.

Class	**⟨⟨atpType⟩⟩ Datatype (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	Abstract base class for user defined (and AUTOSAR predefined) datatypes.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.1: Datatype

Class	**⟨⟨atpType⟩⟩ PrimitiveType (abstract)**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes

Class Desc.	A primitive datatype consists of a set of allowed values.			
Base Class(es)	Datatype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
swDataDef Props	SwData DefProps	0..1	aggregation	

Table 4.2: PrimitiveType

Class	**⟨⟨atpType⟩⟩ CompositeType (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	Abstract base class for all data types composed of other data types.			
Base Class(es)	Datatype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.3: CompositeType

Please note, however, that all these flavors of `Datatype` exist on meta-level M2 (as depicted in figure 4.3), i.e. they can be taken as the basis for defining specific data types on the M1 meta-level. On the other hand, it is not possible to directly use e.g. `IntegerType` directly in an M1 model.

To ensure compatibility between communicating software components, not only the data types involved in the transactions must match. Even if sender and receiver exchange a velocity as 8-bit integer between 0 and 255, the sender may provide this velocity in miles per hours with a resolution of 0.1 mph, while the receiver expects meters per second with a resolution of 1 m/s.

Since the RTE will not implement automatic type conversion on this level, the compatibility of provider and consumer need to be ensured - among other things - through the compatibility of the so-called data-semantics. Data-semantics specify how to convert between physical values (including the physical unit) and the corresponding representation of a computer system. In section 4.5 these two representations are referred to as *physical* and *internal*.

4.3 Usage of Data Types in the Meta-Model

Figure 4.2 sketches some of the usages of a `Datatype` in the AUTOSAR meta-model. In particular, `Datatype` is used to define

- `RecordElements` within the scope of a `RecordType`,
- `Constant`,

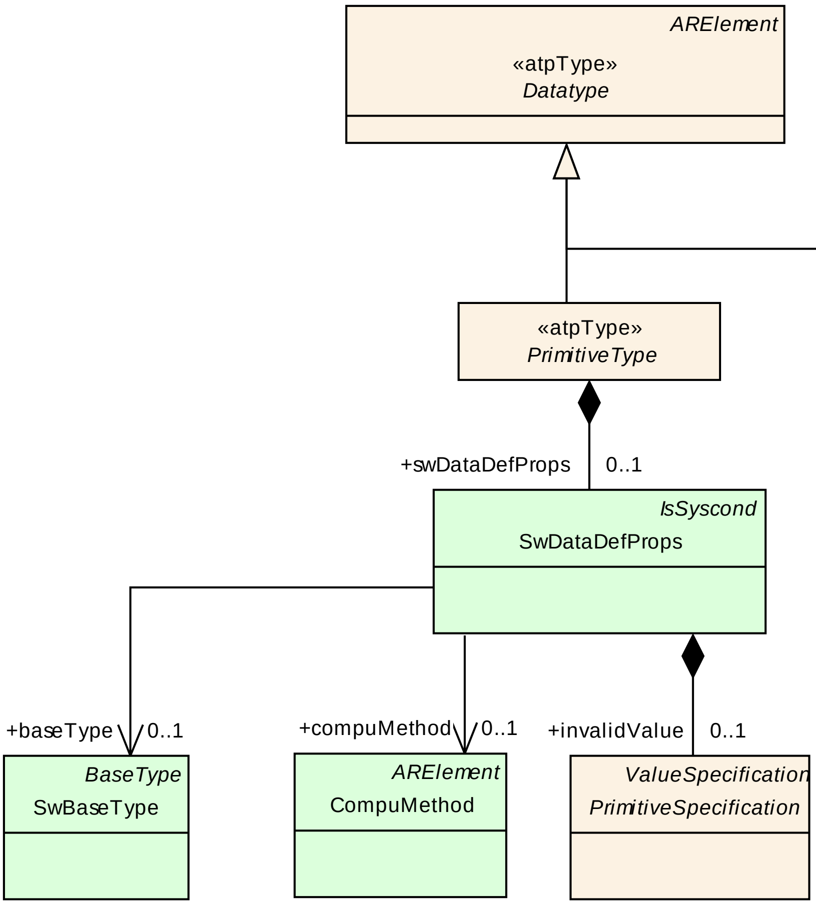


Figure 4.3: Summary of data types on the M2 level

- `DataElementPrototypes` inside a `SenderReceiverInterface`, or
- `ArgumentPrototypes` for the `OperationPrototypes` in a `ClientServerInterface`.

Note that a `Datatype` does not contain any information on the evolution of the values in the `DataType` over time. For example: when a data type types a data-element inside a sender-receiver interface, the data type defines the structure (and semantics) of a specific value (snapshot) of the data; it does not describe any aspects related to its value changing over time.

4.4 Data Type Details

In general, a data type is a set of values characterized by properties of those values and by operations on those values. Primitive data types cannot be decomposed in other data types.

In *low-level* programming languages primitive data types are implemented with respect to the natural data sizes (typically 8, 16, 32, 64 bits) and the operations available in a CPU (for example arithmetic operations for integer and floating-point numbers).

In *higher-level* programming languages data types like integer and float with arbitrary precision, lists, stacks, hash tables and others are provided as primitive data types. For these programming languages resource consumption of time and memory play a minor role.

However in AUTOSAR, resource consumption of time and memory are very important and the exchange of data between software-components must be as efficient as possible. Therefore, the primitive AUTOSAR data types must allow an efficient mapping to programming languages like *C*.

On networks with low bandwidth and small package sizes, like typical automotive CAN, the signals inside the frames mostly are of a much finer granularity: they are not limited to the power-of-2 data-sizes found in software, but can be of arbitrary bit-size. It is common to find a 4-bit or a 12-bit unsigned integer.

At the *Data-Structure* level, the AUTOSAR data types

1. are limited to a small and simple set (and could be extended later by more complex primitive types)
2. support the "arbitrary" bit-sizes needed for a compact representation on networks

Note that it is important to keep in mind the distinction between the structural and the Implementation level. A 12-bit unsigned integer will probably take exactly 12 bits inside a CAN-frame but will probably be mapped onto a 16-bit integer inside the software.

The conversion between both representations is done by the COM layer, which in turn is utilized by the RTE. To ensure the relocatability of software-components, the AUTOSAR standard needs to define a fixed mapping between the structural data types and their implementations in a specific programming language.

4.4.1 Range

When defining a `Datatype`, it is often necessary to specify an open or closed range of values. Semantically, the range represents all real numbers defined by:

range = $\{x \in \Re || LOWER - LIMIT.VALUE < x < UPPER - LIMIT.VALUE\}$
$\bigcup \{LOWER - LIMIT.VALUE\} iff LOWER - LIMIT.INTERVAL - TYPE == CLOSED$
$\bigcup \{UPPER - LIMIT.VALUE\} iff UPPER - LIMIT.INTERVAL - TYPE == CLOSED$

Class	**⟨⟨atpObject⟩⟩ Range (abstract)**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes
Class Desc.	Abstract class for specifying a range from lower limit to upper limit.

Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
lowerLimit	ARLimit	1	aggregation	This element specifies the lower limit of a closed, half-open or open interval. It can also be set to infinity by setting the attribute INTERVAL-TYPE to INFINITE. No value has to be set in the case of an infinite interval.
upperLimit	ARLimit	1	aggregation	This element specifies the upper limit of a closed, half-open or open interval. It can also be set to infinity by setting the attribute INTERVAL-TYPE to INFINITE. No value has to be set in the case of an infinite interval.

Table 4.4: Range

Enumeration	**IntervalTypeEnum**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::LocalConstraints
Enum Desc.	
Literal	***Description***
closed	
infinite	
open	

4.4.2 Primitive Data Types

The following sections describes the primitive types (see figure 4.4) on M2 level in AUTOSAR.

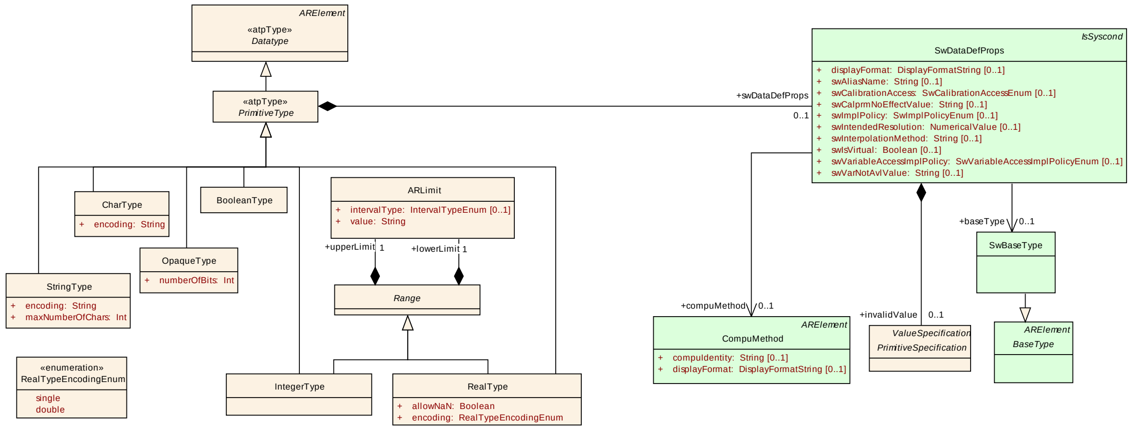


Figure 4.4: Summary of `PrimitiveType`

4.4.2.1 Boolean Type

Class	⟨⟨**atpType**⟩⟩ **BooleanType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	This datatype represents a set containing the logical value true and false			
Base Class(es)	PrimitiveType			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**

Table 4.5: BooleanType

4.4.2.2 Opaque Type

Class	⟨⟨**atpType**⟩⟩ **OpaqueType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	This Datatype represents an array of exactly numberOfBits bits. It is called ”opaque” because this array of bits should be transported ”as is” by the AUTOSAR RTE.			
Base Class(es)	PrimitiveType			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
numberOfBits	Integer	1	aggregation	The number of bits that are used to make up the opaque type.

Table 4.6: OpaqueType

4.4.2.3 Integer Type

`IntegerType` inherits from both `Range` (see section 4.4.1) and `PrimitiveType`. Therefore the attributes `upperLimit` and `lowerLimit` are defined implicitly.

Class	⟨⟨**atpType**⟩⟩ **IntegerType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	This data-type are the integers in the interval defined by the Range.			
Base Class(es)	PrimitiveType , Range			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.7: IntegerType

Semantically a range of type `IntegerType` is the intersection of the range of real numbers as defined section 4.4.1 and the numbers that can be expressed by the data type integer. For example, the following values of the `IntegerType` attributes define a (M1) data type containing the integers 0, 1, 2 and 3.

```
lowerLimit = 0
lowerLimit.INTERVAL-TYPE = CLOSED
upperLimit = 4
upperLimit.INTERVAL-TYPE = OPEN
```

4.4.2.4 Real Type

When attribute `encoding` is set to `Single` or `Double`, the values in this data type are the real numbers that can be represented by the IEC 60559 (IEEE 754) standard for single-precision resp. double-precision numbers and that lie in the interval defined by the `Range`.

Class	⟨⟨**atpType**⟩⟩ **RealType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	This represents a range of reals that can be represented by either the IEEE 754 ”Single Precision” (encoding is ”Single”) or IEEE 754 ”Double Precision” (encoding is ”Double”) arithmetic. Note that these standards include representations for +infinity, -infinity, QNaN and SNaN. When defining a RealType, one must indicate whether these special values are allowed or not.			
Base Class(es)	PrimitiveType , Range			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
allowNaN	Boolean	1	aggregation	Denotes whether this data type permits for ”not a number” being represented by the type

encoding	RealType Encoding Enum	1	aggregation	Denotes the precision of the RealType

Table 4.8: RealType

In other words: A range of type `RealType` is the intersection of the range of real numbers as defined section 4.4.1 and the numbers that can be expressed by the floating point representation defined by the attribute `encoding`.

For example, a `RealType` with the following attributes defines the entire range of values that can be represented as a common IEC 60559 single-precision float, including the special values infinity and NaN (Not-a-Number).

```
encoding = "Single"
lowerLimit = -INF
lowerLimit.INTERVAL-TYPE = CLOSED
upperLimit = +INF
upperLimit.INTERVAL-TYPE = CLOSED
allowNaN = TRUE
```

It might be possible to extend this format to allow for other floating-point formats (for example, special formats used by specific digital signal processors).

4.4.2.5 Char Type

For the definition of the attribute `encoding` of `CharType` and `StringType` the names described in table 4.10 shall be used. The table shows a list of frequently used encodings and is based on the Character Sets document of the Internet Assigned Numbers Authority. That document describes *The official names of character sets that may be used in the Internet* and references to the definitions and standardizations of these character sets.

Class	⟨⟨**atpType**⟩⟩ **CharType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	This represents a character belonging to the character-set specified in the encoding. The semantics are built-in into this datatype.			
Base Class(es)	PrimitiveType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
encoding	String	1	aggregation	Specification of character encoding, e.g. ISO-8859-1

Table 4.9: CharType

The table was created by

Name of Encoding	*Description*
US-ASCII	American standard code for information interchange
UTF-8	Eight-bit Unicode transformation format
UTF-16	Sixteen-bit Unicode Transformation Format, byte order specified by a mandatory initial byte-order mark
ISO-8859-1	Latin alphabet No. 1
ISO-8859-2	Latin alphabet No. 2
ISO-8859-3	Latin alphabet No. 3
ISO-8859-4	Latin alphabet No. 4
ISO-8859-5	Latin/Cyrillic alphabet
ISO-8859-6	Latin/Arabic alphabet
ISO-8859-7	Latin/Greek alphabet
ISO-8859-8	Latin/Hebrew alphabet
ISO-8859-9	Latin alphabet No. 5

Table 4.10: Character encodings

1. choosing the name or alias of a character set which is marked as *preferred MIME name*
2. or by choosing the name if no *preferred MIME name* is defined

If table 4.10 needs to be extended the same rules shall be applied.

4.4.2.6 String Type

4.4.2.7 About enumerations

In the AUTOSAR meta-model, an enumeration is not implemented by means of `PrimitiveType`. Instead, a range of integer numbers can be used as a structural description. The mapping of the integer numbers on *labels* in the scope of the definition of an enumeration is part of the *Data-Semantics* level and therefore not part of the structural description.

Class	⟨⟨**atpType**⟩⟩ **StringType**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes
Class Desc.	This represents a string of characters out of the character-set specified by the given encoding. The maxNumberOfChars is the maximal number of characters which can be stored within the String. The actual number of bytes that is required to represent the string can be calculated out of maxNumberOfChars and the encoding: bytes required to represent the string = maxNumberOfChars * (max bytes per character using the given encoding) + 1 (terminating null)
Base Class(es)	PrimitiveType

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
encoding	String	1	aggregation	Specification of character encoding, e. g. ISO-8859-1.
maxNumberOfChars	Integer	1	aggregation	The maxNumberOfChars is the maximum number of characters that can be stored in the string.

Table 4.11: StringType

4.4.3 Composite Data Types

The meta-classes `ArrayType` and `RecordType` (details are depicted in figure 4.5) provide the means to define composite data types. It is possible to use a combination of `ArrayType` and `RecordType`, so that an `ArrayType` could be defined as `RecordElement` of a `RecordType` and in the same manner a `RecordType` could be used as the base type of an `ArrayType`. The creation of nested `CompositeTypes` is also possible.

4.4.3.1 ArrayType

An `ArrayType` may contain `maxNumberOfElements` `ArrayElements`. Each of these ArrayElements must have the same type. When referring to an element of an array within the software-component descriptions, the element-index runs from 0 to the value of `maxNumberOfElements`-1.

Class	**⟨⟨atpType⟩⟩ ArrayType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.				
Base Class(es)	CompositeType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
element	ArrayElement	1	aggregation	

Table 4.12: ArrayType

Class	**⟨⟨atpPrototype⟩⟩ ArrayElement**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.				
Base Class(es)	DataPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

maxNumberOf Elements	Integer	1	aggregation	The maximum number of elements that the array can contain.

Table 4.13: ArrayElement

4.4.3.2 RecordType

A declaration of `RecordType` describes a nonempty set of objects, each of which has a unique identifier with respect to the `RecordType` and a `Datatype`. The `shortName` of each `RecordElement` within the scope of an `RecordType` must be unique.

Class	⟨⟨**atpType**⟩⟩ **RecordType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.				
Base Class(es)	CompositeType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
element (ordered)	Record Element	1..*	aggregation	

Table 4.14: RecordType

Class	⟨⟨**atpPrototype**⟩⟩ **RecordElement**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Desc.	An element in a record.			
Base Class(es)	DataPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.15: RecordElement

4.4.4 Constant

The AUTOSAR standard allows the utilization of constant values in two ways:

1. by referencing a publicly defined `ConstantSpecification`
2. or through an inline aggregation of a constant value (meta-class `ValueSpecification`).

Class	⟨⟨**atpObject**⟩⟩ **ConstantSpecification**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants

Class Desc.	Specification of a constant that can be part of a package, i.e. it can be defined stand-alone.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
value	ValueSpecification	1	aggregation	Specification of an expression leading to a value of a given datatype.

Table 4.16: ConstantSpecification

Class	⟨⟨**atpPrototype**⟩⟩ **ValueSpecification (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	Description of a constant of a modeled datatype (M1 datatype).			
Base Class(es)	DataPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.17: ValueSpecification

Class	⟨⟨**atpPrototype**⟩⟩ **PrimitiveSpecification (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	A constant of a primitive datatype.			
Base Class(es)	ValueSpecification			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.18: PrimitiveSpecification

Class	⟨⟨**atpPrototype**⟩⟩ **ArraySpecification**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	A constant array, which refers to its elements by index.			
Base Class(es)	ValueSpecification			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
element (ordered)	ValueSpecification	*	aggregation	Elements of array.

Table 4.19: ArraySpecification

Class	⟨⟨**atpPrototype**⟩⟩ **RecordSpecification**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants
Class Desc.	

Base Class(es)	ValueSpecification			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
element (ordered)	ValueSpecification	*	aggregation	Elements of the record.

Table 4.20: RecordSpecification

The structure of a `ValueSpecification` is defined by its `Datatype`. Specialized subclasses of `ValueSpecification` allow for the definition of values for the different kinds of `Datatype`, e.g. `BooleanValue` specifies the value for a `BooleanType` and an `ArraySpecification` does the same for an `ArrayType`. This relationship is formally expressed in figure 4.6.

Class	⟨⟨**atpPrototype**⟩⟩ **BooleanLiteral**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	Boolean constant expression.			
Base Class(es)	PrimitiveSpecification			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
value	Boolean	1	aggregation	The Boolean value.

Table 4.21: BooleanLiteral

Class	⟨⟨**atpPrototype**⟩⟩ **OpaqueLiteral**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	An opaque literal.			
Base Class(es)	PrimitiveSpecification			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
value	String	1	aggregation	The string encodes an array of bytes in the following syntax ”ae:05:fe”

Table 4.22: OpaqueLiteral

Class	⟨⟨**atpPrototype**⟩⟩ **IntegerLiteral**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	Constant integer value.			
Base Class(es)	PrimitiveSpecification			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
value	Integer	1	aggregation	The value.

Table 4.23: IntegerLiteral

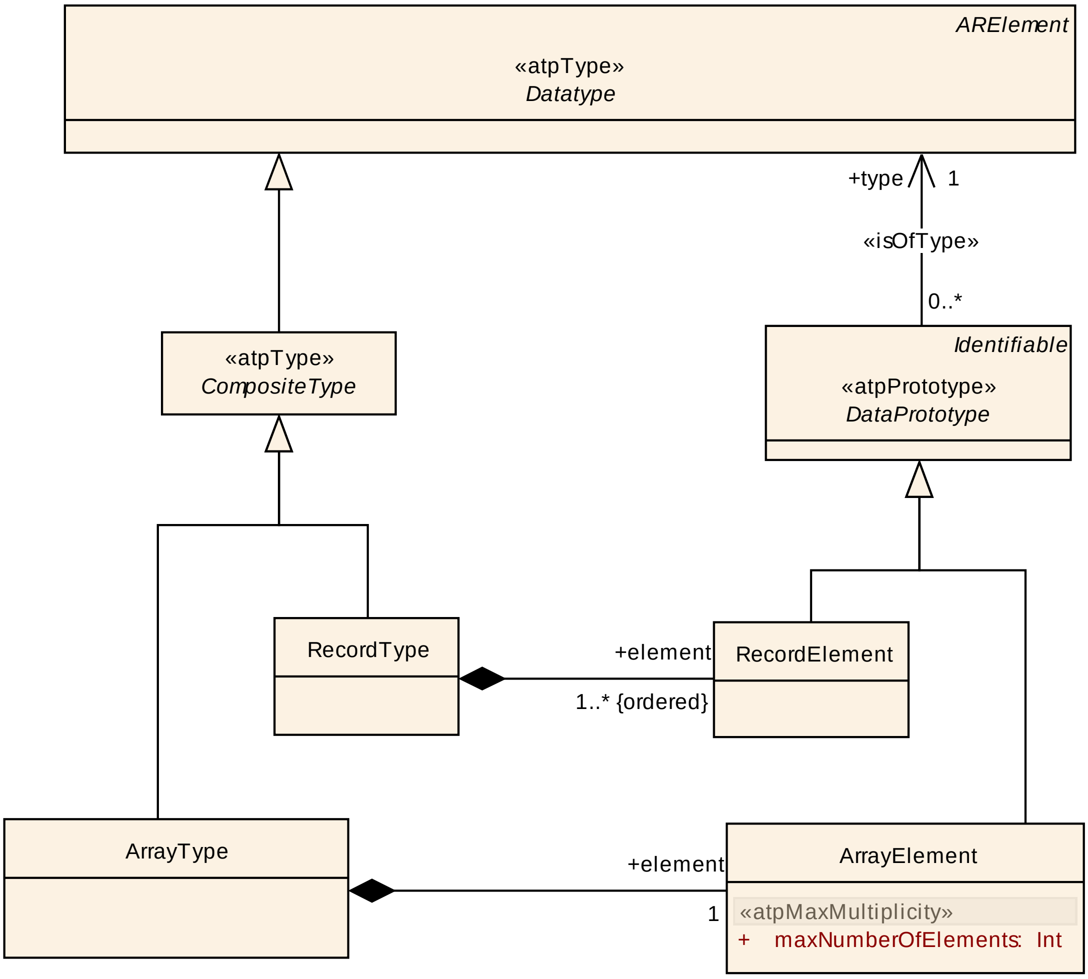


Figure 4.5: Summary of `CompositeType`

Class	⟨⟨**atpPrototype**⟩⟩ **RealLiteral**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	Constant description for real values.			
Base Class(es)	PrimitiveSpecification			
Attribute	**_Datatype_**	**_Mul._**	**_Link Type_**	**_Description_**
value	Float	1	aggregation	The numeric value itself.

Table 4.24: RealLiteral

Class	⟨⟨**atpPrototype**⟩⟩ **CharLiteral**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	Character constant description.			
Base Class(es)	PrimitiveSpecification			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
value	String	1	aggregation	The character value (a string of length 1).

Table 4.25: CharLiteral

Class	⟨⟨**atpPrototype**⟩⟩ **StringLiteral**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	A constant string.			
Base Class(es)	PrimitiveSpecification			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
value	String	1	aggregation	The string itself.

Table 4.26: StringLiteral

A specific `ValueSpecification` is the `ConstantReference`: it passes the definition of the constant value on to another `ConstantSpecification` that is defined as part of an AUTOSAR `Package`.

Class	⟨⟨**atpPrototype**⟩⟩ **ConstantReference**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Constants			
Class Desc.	Instead of defining this constant inline, another constant is referenced.			
Base Class(es)	ValueSpecification			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
constant	ConstantSpecification	1	reference	The referenced constant.

Table 4.27: ConstantReference

4.5 Datatypes with Semantics

It does not make sense to specify semantics and therefore a physical meaning to all of the data types explained in the previous section. More precisely, data semantics may be assigned to `PrimitiveTypes` only.

Class	⟨⟨**atpObject**⟩⟩ **SwDataDefProps**
Package	M2::AUTOSARTemplates::CommonStructure::DataDefProperties

Class Desc.	This class is a collection of properties relevant for data objects under various aspects. One could consider this class as a ”pattern of inheritance by aggregation”. The properties can be applied to all objects of all classes in which SwDataDefProps is agrregated. Note that not all of the attributes or associated elements are useful all of the time. Hence, the process definition (e.g. expressed with an OCL or a Document Control Instance) MSR-DCI has the task of implementing limitations. SwDataDefProps covers various aspects: * Structure of the data element, is it a single value, a curve, or a map, but also the recordLayouts which specify, how such elements are mapped/converted to the DataTypes in the programming language (or in Autosar). This is mainly expressed by properties like swRecordLayout and swCalprmAxisSet * Implementation policy, mainly expressed by swImplPolicy, swVariableAccessImplPolicy, swAddrMethod * Access policy for the MDC system, mainly expressed by swCalibrationAccess * Semantics of the data element, mainly expressed by compuMethod and/or unit, dataConstr * Code generation policy provided by swCodeSyntax			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
annotation	Annotation	*	aggregation	This aggregation allows to add annotations (yellow pads ...) related to the current data object.
baseType	SwBase Type	0..1	reference	Base type associated with the value axis of this data object.
compu Method	Compu Method	0..1	reference	Computation method associated with the semantics of this data object.
dataConstr	DataCon-str	0..1	reference	Data constraint for this data object.
display Format	Display Format String	0..1	aggregation	This property describes how a number is to be rendered e.g. in documents or in a measurement and calibration system.
invalid Value	Primitive Specifica-tion	0..1	aggregation	Optional value to express invalidity of the actual data element. If given, the owning component has the API to set this data element invalid, otherwise it does not.
swAddr Method	SwAddr Method	0..1	reference	Addressing method related to this data object.
swBitRep-resentation	SwBitRep-resentation	0..1	aggregation	Description of the binary representaion in case of a bit variable.
swCalibra-tionAccess	SwCalibra-tionAccess Enum	0..1	aggregation	Specifies the read or write access by MCD tools for this data object.

swCalprm AxisSet	SwCalprm AxisSet	0..1	aggregation	This specifies the properties of the axes in case of a curve or map etc. This is mainly applicable to calibration parameters.
swCode Syntax	SwCode Syntax	0..1	reference	Coding policy for this data object expressed as a reference to a Code syntax to be applied.
swDataDependency	SwData Dependency	0..1	aggregation	If the data object is virtual - that means it is not directly in the ecu, then this property describes how the ”virtual variable” can be computed from the real ones.
swHost Variable	SwVariable Ref	0..1	aggregation	Contains a reference to a variable, which serves as a host-variable for a bit variable. Only applicable to bit objects.
swImpl Policy	SwImpl Policy Enum	0..1	aggregation	Implementation policy for this data object.
swPointer	SwPointer	0..1	aggregation	Specifies that the containing data object is a pointer to another data object.
swRecord Layout	SwRecord Layout	0..1	reference	Record layout for this data object.
swText Props	SwText Props	0..1	aggregation	the specific properties if the data object is a text object.
swValue BlockSize	SwArraysize	0..1	aggregation	Specifies the size in case the data object is an VAL_BLK. It is there for compatibility reasons, where value blocks were introduced as a kind of an array.
swVariable Access ImplPolicy	SwVariable Access ImplPolicy Enum	0..1	aggregation	In case of a swImplPolicy set to ”message” the access policy can be refined here.
unit	Unit	0..1	reference	Physical unit associated with the semantics of this data object. This attribute applies, if no compuMethod is specified. If buth units (this as well as via compuMethod is specified,the units ust be the same.

Table 4.28: SwDataDefProps

Class	⟨⟨**atpObject**⟩⟩ **CompuMethod**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
compu InternalTo Phys	Compu	0..1	aggregation	

compu PhysTo Internal	Compu	0..1	aggregation	
display Format	Display Format String	0..1	aggregation	This property specifies, how the physical value shall be displayed e.g. in documents or measurement and calibration tools.
unit	Unit	0..1	reference	This is the physical unit of the Physical values for which the CompuMethod applies.

Table 4.29: CompuMethod

A `CompositeType` cannot be given a particular semantic meaning besides the one occasionally specified for the contained primitive data elements.

Since `PrimitiveTypes` with specified semantics may often be reused, it is possible to assign additional properties to a `PrimitiveType` using `swDataDefProps`. The actual semantics class is called `CompuMethod`, due to compatibility with *MSR-SW*.

The diagram also shows that in addition to the semantics defined through the `CompuMethod` (explained below), also an `invalidValue` can be specified. This is a requirement of the VFB [3], allowing to express which specific value in a given data range is used to indicate invalidation.

The `PrimitiveType` allows to specify a constant value for this purpose. Of course, the constant value also needs to be a primitive value again. More specific, it even needs to be of the same type as the original `PrimitiveType` (not shown in diagram). Please note that `Constants` are explained in section 4.4.4.

The following section explains the usage of the class `CompuMethod` in order to allow specification of the data semantics of a `PrimitiveType`.

4.5.1 Computation Methods

This meta-class was actually taken from the *ASAM* standard's *harmonized data objects*. This is also indicated by the green color of the meta-classes in the diagram.

`CompuMethods` (see figure 4.8) are used for the conversion of internal values into their physical representation and vice versa. The direction of the conversion depends on the origin of the value to be converted: If the value is provided by the ECU then the conversion direction is from internal to physical. Physical values that are provided by the tester are converted to internal values before they are sent to the ECU.

The preferred conversion direction depends on the use case. The physical-to-internal direction is suitable for calibration while the internal-to-physical direction is preferred for diagnostic purposes. A CompuMethod can be defined for each of these directions.

In the following section, the internal-to-physical conversion direction is used as the default. Usually a `CompuMethod` is defined for one conversion direction only even if it

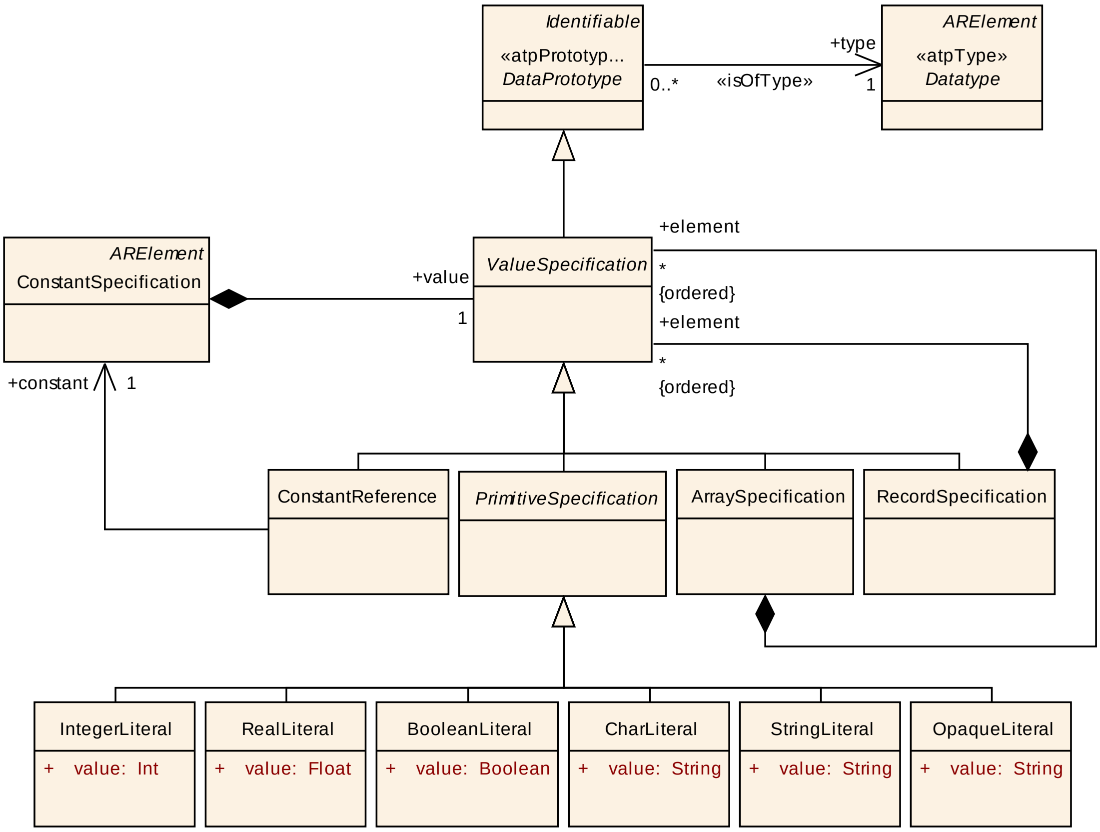


Figure 4.6: Summary of `Constant`

is used in both directions. For simple functions like identical or linear functions this is sufficient because the inverse function as well as the applicable limits can be derived quite easily from the defined function.

For more complex functions (e.g. rational functions) it is usually not possible to compute the inverse function automatically. More seriously, the inversion yields ambiguous results if the function is not monotonic. To deal with such possible ambiguities in a direct way an inverse value can be provided explicitly for the function or for each of its parts respectively. In case that both domains are specified in the compu-method, both shall have limits.

The `compuDefaultValue` is used to specify an invalid value and is specified in the internal domain. Additionally, the `compuDefaultValue` is not bound to the given upper- and lower-limits of an integer-type or of an associated compu-method.

As a `CompuMethod` specifies the conversion between the physical world and the numerical values, they must refer to a unit.

Figure 4.8 sketches a conceptual overview of `CompuMethod`. It consists of the following attributes:

- A physical unit (described in next section) to be associated with the `Datatype` to which the `CompuMethod` is associated. Note that quantities like ”%” are not

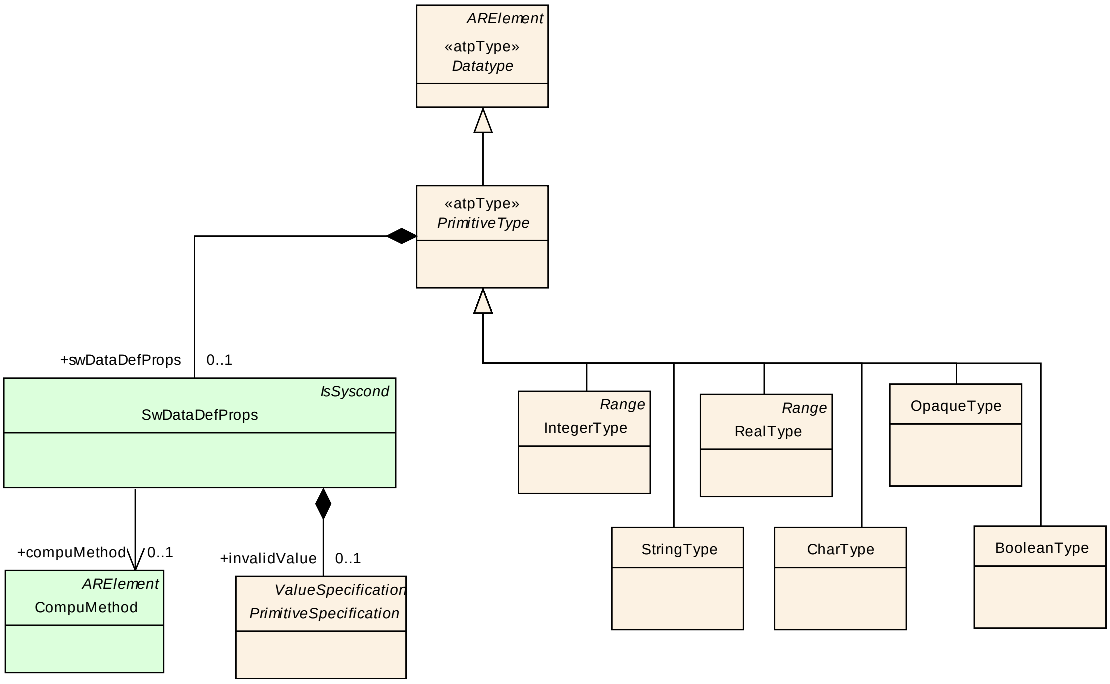


Figure 4.7: Data types with semantics

derived from SI units. However, they have a meaning in the physical world and need to be represented in form of datatypes. Therefore, a CompuMethod also applies in those cases.

- A conversion specification from internal to physical values, as well as the reverse conversion. Both of them in turn consist of an abstract `CompuContent`. Derived classes allow the specification of a conversion formula in two different ways. Within AUTOSAR only the stepwise definition (`CompuScales`) is used.
- `CompuScales` is a number of intervals (called `CompuScale`) within which a certain conversion applies. The respective interval is given in terms of upper and lower limit. Limits have already been explained in the data types chapter. Within each `CompuScale` we have the abstract `CompuScaleContent`. To deal with possible ambiguities in a direct way an inverse value can be provided explicitly for that particular scale (`compuInverseValue`).
- As the diagram shows, `CompuScaleContent` is an abstract meta-class. A number of derived meta-classes allow the specification of a conversion formula in a variety of ways, including:
  - mapping the whole interval to a constant (`CompuConst`)
  - providing rational coefficients of the conversion formula (`CompuRationalCoeffs`)
- The rational function is specified as rational coefficients for the numerator (`compuNumerator`) and the denominator (`compuDenominator`).

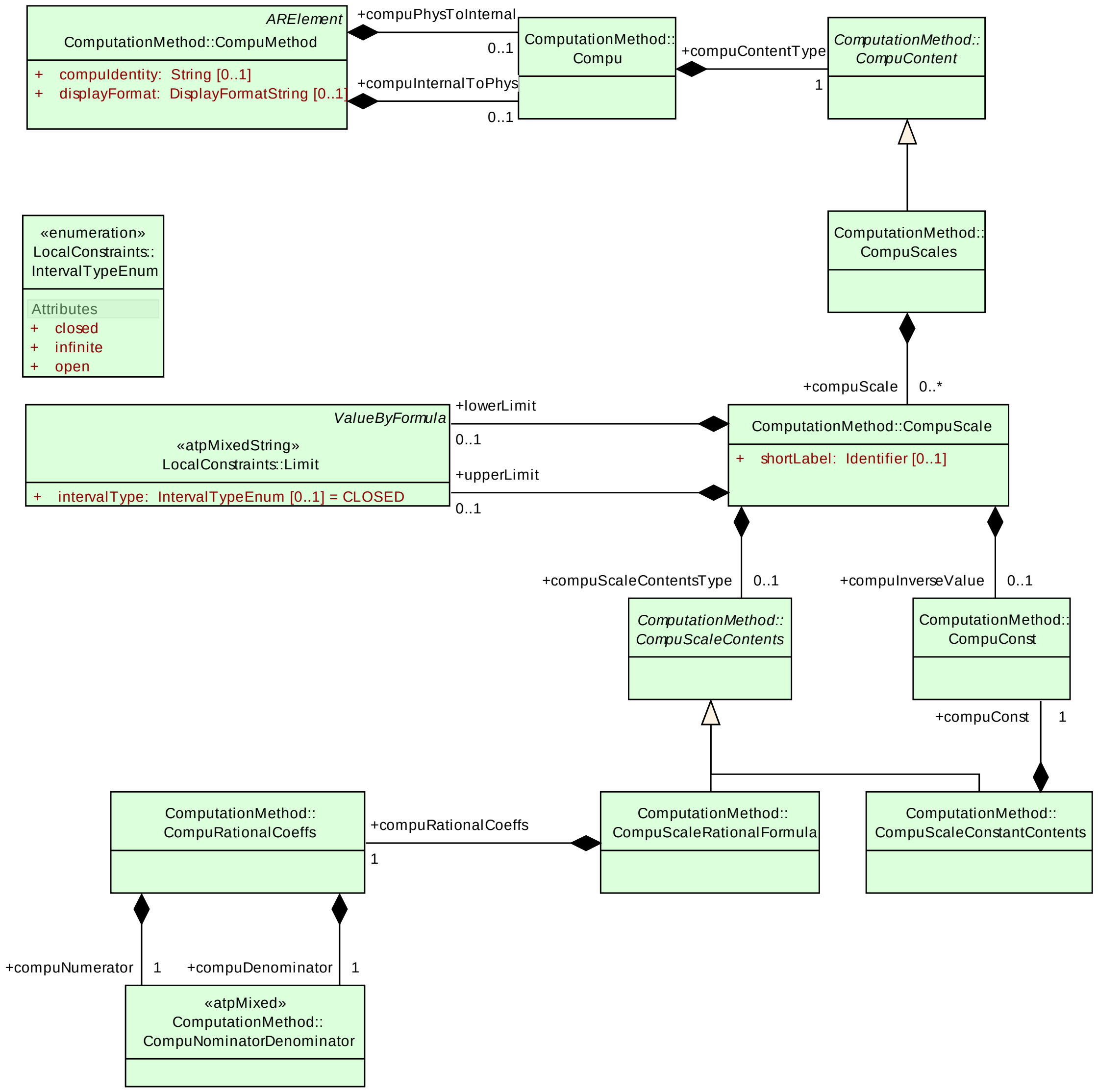


Figure 4.8: A `CompuMethod` and its attributes define data semantics

`CompuNominatorDenominator` can have as many *V* elements as needed for the rational function. The sequence of the values *V* carries the information for the exponents, that means the first *V* is the coefficient for x0, the second *V* is the coefficient for x1, etc. With this sequence the values of the exponents can be entirely represented.

Class	⟨⟨**atpObject**⟩⟩ **Compu**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

compu ContentType	Compu Content	1	aggregation	
compuDefaultValue	Compu Const	0..1	aggregation	This property can be used to specify an output value for a conversion formula, if the value to be converted lies outside the plausibility limit. Although this is possible for all conversion formulae, it is especially valid for variables with tabular conversion formulae.

Table 4.30: Compu

Class	⟨⟨**atpObject**⟩⟩ **CompuContent (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.31: CompuContent

Class	⟨⟨**atpObject**⟩⟩ **CompuScale**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
desc	MIData2	0..1	aggregation	<desc> represents a general but brief description of the object in question.
compu ScaleContentsType	Compu Scale Contents	0..1	aggregation	
lowerLimit	Limit	0..1	aggregation	This element specifies the lower limit of a closed, half-open or open interval. It can also be set to infinity by setting the attribute INTERVAL-TYPE to INFINITE. No value has to be set in the case of an infinite interval.
shortLabel	Identifier	0..1	aggregation	This element specifies a short name for the particular scale. The name can for example be used to derive a programming language identifier.
upperLimit	Limit	0..1	aggregation	This element specifies the upper limit of a closed, half-open or open interval. It can also be set to infinity by setting the attribute INTERVAL-TYPE to INFINITE. No value has to be set in the case of an infinite interval.

Table 4.32: CompuScale

Class	⟨⟨**atpObject**⟩⟩ **CompuScales**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	CompuContent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
compu Scale	Compu Scale	*	aggregation	

Table 4.33: CompuScales

Class	⟨⟨**atpObject**⟩⟩ **CompuScaleContents (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.34: CompuScaleContents

Class	⟨⟨**atpObject**⟩⟩ **CompuRationalCoeffs**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
compuDenominator	Compu Nominator Denominator	1	aggregation	
compu Numerator	Compu Nominator Denominator	1	aggregation	

Table 4.35: CompuRationalCoeffs

Class	⟨⟨**atpObject**⟩⟩ **CompuConst**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

compuConstContentType	CompuConstContent	1	aggregation	

Table 4.36: CompuConst

For a detailed description of `compuMethods`, please refer to the *ASAM MCD 2 Harmonized Data Objects*.

ASAM Category	*Meaning*	*Specific dataDefProps*
IDENTICAL	This `CompuMethod` just hands over the internal value with an optional unit.	Only the base elements are allowed and `UNIT-REF`, `PHY-SCONSTR` and `INTERNAL-CONSTR` are optional. This is the simplest type of a `CompuMethod`.
LINEAR	A linear conversion can be performed in two steps: The internal value is multiplied with a factor; after that, an offset is added to the result of the multiplication.	Exactly one `CompuScale`, with two V in `compuNominator` and on V in `compuDenominator`.
SCALE_LINEAR	Used for a piecewise linear conversion	more than one `COMPU-SCALE` can be defined. Additionally there have to be the `UPPER-LIMIT` and `LOWER-LIMIT` elements, which define the region of validity for the linear function. The boundaries of the regions must not overlap.
RAT_FUNC	The rational function type is similar to the linear type without the restrictions for the `COMPU-NUMERATORs` and `COMPU-DENOMINATORs`.	It can have as many `Velements` as needed for the rational function. The sequence of the values *V* carries the information for the exponents, that means the first *V* is the coefficient for x0, the second *V* is the coefficient for x1, etc. With this sequence the values of the exponents can be entirely represented. A rational function is only applicable for conversions in the direction that it is defined for, i.e. the automatic calculation of the inverse function is not supported by the MCD system.

ASAM Category	*Meaning*	*Specific dataDefProps*
SCALE_RAT_FUNC	Used for piecewise defined rational conversion.	
TEXTTABLE	The type TEXTTABLE is used for transformations of the internal value into textual elements.	UNIT-REF and PHYS-CONSTR are not allowed. COMPU-INTERNAL-TO-PHYS must exist with COMPU-SCALES consisting of UPPER-LIMIT and LOWER-LIMIT. The result is placed in the VT member of COMPU-CONST. The COMPU-DEFAULTVALUE is optional. If the reverse calculation is needed then for each scale the COMPU-INVERSE-VALUE can be used to define the reverse calculation result. If no inverse value is explicitly defined then the smallest possible value of the scale12 will be used as result of the reverse calculation.
TAB_NOINTP	Similar to TEXTTABLE but for numerical values.	The values per scale are defined in `compuConst`.

Table 4.37: ASAM Categories

Class	⟨⟨**atpObject**⟩⟩ **CompuScaleRationalFormula**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	CompuScaleContents			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
compu Rational Coeffs	Compu Rational Coeffs	1	aggregation	

Table 4.38: CompuScaleRationalFormula

Class	⟨⟨**atpObject**⟩⟩ **CompuScaleConstantContents**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	CompuScaleContents			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

compu Const	Compu Const	1	aggregation	

Table 4.39: CompuScaleConstantContents

Class	**⟨⟨atpMixed⟩⟩ CompuNominatorDenominator**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::ComputationMethod			
Class Desc.				
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
v	String	1	aggregation	Use <v> to enter a numerical value.
vf	Vf	1	aggregation	Value calculated via a system constant. This element is included in every case, where parameters should be generated from numerical values during compile time (not runtime!). Thus for example, the influence of the cylinder number on conversion formulae, can be introduced in a repeatable manner.

Table 4.40: CompuNominatorDenominator

4.5.1.1 Example for Enumeration

The following example illustrates how an enumeration is specified using `CompuMethod`.

```
<COMPU-METHOD>
    <SHORT-NAME>boolean</SHORT-NAME>
    <CATEGORY>TEXTTABLE</CATEGORY>
    <COMPU-INTERNAL-TO-PHYS>
        <COMPU-SCALES>
            <COMPU-SCALE>
                <LOWER-LIMIT INTEVAL-TYPE="CLOSED">0</LOWER-LIMIT>
                <UPPER-LIMIT INTEVAL-TYPE="CLOSED">0</UPPER-LIMIT>
                <COMPU-CONST>
                    <VT>false</VT>
                </COMPU-CONST>
            </COMPU-SCALE>
            <COMPU-SCALE>
                <LOWER-LIMIT INTEVAL-TYPE="CLOSED">1</LOWER-LIMIT>
                <UPPER-LIMIT INTEVAL-TYPE="CLOSED">1</UPPER-LIMIT>
                <COMPU-CONST>
                    <VT>true</VT>
                </COMPU-CONST>
            </COMPU-SCALE>
        </COMPU-SCALES>
    </COMPU-INTERNAL-TO-PHYS>
```

```
</COMPU-METHOD>
```

4.5.1.2 Example for linear conversion

The following example illustrates how a linear conversion is specified using `CompuMethod`.

$F_{[kmh]} = 30_{[kmh]} + 2_{[kmh]} * x$

```
<COMPU-METHOD>
    <SHORT-NAME>linear</SHORT-NAME>
    <CATEGORY>LINEAR</CATEGORY>
    <UNIT-REF>kmh</UNIT-REF>
    <COMPU-INTERNAL-TO-PHYS>
        <COMPU-SCALES>
           <COMPU-SCALE>
             <COMPU-RATIONAL-COEFFS>
               <COMPU-NUMERATOR>
                  <V>30</V>
                  <V>2</V>
               </COMPU-NUMERATOR>
               <COMPU-DENOMINATOR>
                  <V>1</V>
               </COMPU-DENOMINATOR>
             </COMPU-RATIONAL-COEFFS>
            <COMPU-SCALE>
        </COMPU-SCALES>
    </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
```

4.5.2 Physical Units

An important part of the semantics associated with a data type is its physical dimension. Units are used to augment the value with additional information like *m/s* or *liter*. That is necessary for a correct interpretation of the physical value for input and output processes.

The conversion of values into other units like *km/h* into *miles/h* is also possible. Therefore the unit involves information about its physical dimensions. The substructure of physical dimensions defines all used quantities in the SI-System [1] (e.g. velocity as length/time corresponds to m/s).

The unit references one physical dimension. If the physical dimensions of two units are identical, a conversion between them is possible. 4.9 depicts the concept how units are defined.

---

[1] For the definition of what SI units are, see http://physics.nist.gov/cuu/Units/

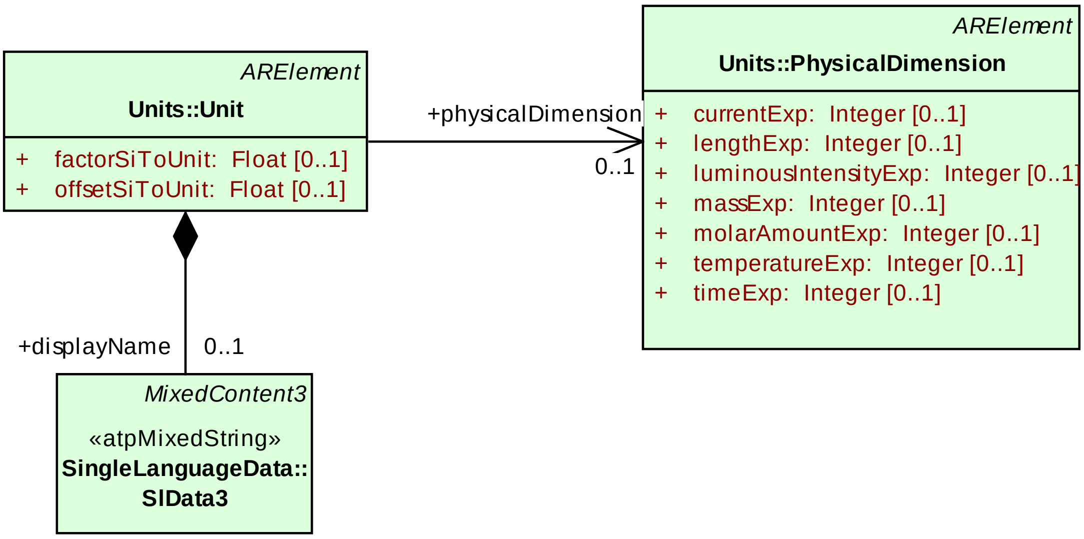


Figure 4.9: Definition of SI based units

For a detailed description of these elements please refer to the *ASAM MCD 2 Harmonized Data Objects*. Standard units are already predefined for AUTOSAR in form of a description file.

Class	⟨⟨**atpObject**⟩⟩ **Unit**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Units			
Class Desc.	This is a physical measurement unit. All units that might be defined should stem from SI units. In order to convert one unit into another factor and offset are defined. For the calculation from SI-unit to the defined unit the factor (factorSiToUnit) and the offset (offsetSiToUnit) are applied: unit = siUnit * factorSiToUnit + offsetSiToUnit For the calculation from a unit to SI-unit the reciprocal of the factor (factorSiToUnit) and the negation of the offset (offsetSiToUnit) are applied: siUnit = (unit - offsetSiToUnit) / factorSiToUnit			
Base Class(es)	ARElement			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
displayName	SIData3	0..1	aggregation	
factorSiToUnit	Float	0..1	aggregation	this is the factor for the convesion from and to siUnits.
offsetSiToUnit	Float	0..1	aggregation	this is the offset for the convesion from and to siUnits.
physicalDimension	PhysicalDimension	0..1	reference	

Table 4.41: Unit

For basing a new unit directly upon SI units an exponent for each of the seven fundamental dimensions and its corresponding SI unit needs to be specified. Negative exponents are allowed. Note that quantities like ”%” are not derived from SI units and therefore have no association to a physical dimension.

If a new unit is based on an existing unit that has been defined earlier, a factor and offset, which are applied to the referenced unit, need to be specified.

Class	⟨⟨**atpObject**⟩⟩ **PhysicalDimension**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Units			
Class Desc.	This class represents a physical dimension. If the physical dimension of two units is identical a conversion between them is possible. The conversion between units is related to the definition of the physical dimension.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
currentExp	Integer	0..1	aggregation	the exponent of the physical dimension ”electric current”
lengthExp	Integer	0..1	aggregation	The exponent of the physical dimension ”length”
luminous Intensity Exp	Integer	0..1	aggregation	The exponent of the physical dimension ”luminous intensity”
massExp	Integer	0..1	aggregation	The exponent of the physical dimension ”mass”
molar Amount Exp	Integer	0..1	aggregation	The exponent of the physical dimension ”quantity of substance”
temperature Exp	Integer	0..1	aggregation	The exponent of the physical dimension ”temperature”
timeExp	Integer	0..1	aggregation	The exponent of the physical dimension ”time”

Table 4.42: PhysicalDimension

4.5.3 Base Type

`BaseType` is used to specify in detail the Data Implementation level mentioned in chapter 4.1. For a detailed description of `BaseTypes`, please refer to the *ASAM MCD 2 Harmonized Data Objects*[2]. This information is necessary to create an *A2L*-File.

---

[2] The definition of *Harmonized Data Objects* can be retrieved from ASAM at www.asam.net. Access is limited to ASAM members

Class	⟨⟨**atpObject**⟩⟩ **BaseType (abstract)**			
Package	M2::AUTOSARTemplates::CommonStructure::BaseTypes			
Class Desc.				
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
baseTypeDefinitionType	BaseTypeDefinition	1	aggregation	

Table 4.43: BaseType

Class	⟨⟨**atpObject**⟩⟩ **BaseTypeDefinition (abstract)**			
Package	M2::AUTOSARTemplates::CommonStructure::BaseTypes			
Class Desc.				
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.44: BaseTypeDefinition

Class	⟨⟨**atpObject**⟩⟩ **BaseTypeDirectDefinition**			
Package	M2::AUTOSARTemplates::CommonStructure::BaseTypes			
Class Desc.	This BaseType is defined directly (as opposite to a derived BaseType)			
Base Class(es)	BaseTypeDefinition			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
baseTypeEncoding	BaseTypeEncodingString	1	aggregation	This specifies, how an object of the current BaseType is encode eg. in an ECU in a message sequence.
baseTypeSizeDefintionType	BaseTypeSizeDefinition	1	aggregation	This aggregation is necessary to specify the exact sequence of properties in the xml-file. It represents the size of the BaseType.
byteOrder	ByteOrder	0..1	aggregation	This element specifies the byte order of the parent element. The byte order is defined with the attribute TYPE. Possible values are: * MOST-SIGNIFICANT-BYTE-FIRST * MOST-SIGNIFICANT-BYTE-LAST
memAlignment	Integer	0..1	aggregation	describes the alignment of the memory object in bits. E.g. ”1” specifies, that the object in question is aligned to a byte while ”32” specifies that it is aligned four byte.

Table 4.45: BaseTypeDirectDefinition

Class	⟨⟨**atpObject**⟩⟩ **BaseTypeSizeDefinition (abstract)**			
Package	M2::AUTOSARTemplates::CommonStructure::BaseTypes			
Class Desc.	This abstract class represents the possible methods of defining the size of a BaseType.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 4.46: BaseTypeSizeDefinition

Class	⟨⟨**atpObject**⟩⟩ **BaseTypeAbsSize**			
Package	M2::AUTOSARTemplates::CommonStructure::BaseTypes			
Class Desc.	This is the absolute size of the basetype. In this case the BaseType is of fixed lenght.			
Base Class(es)	BaseTypeSizeDefinition			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
baseTypeSize	Integer	0..1	aggregation	Describes the length of the data type specified in the container in bits.

Table 4.47: BaseTypeAbsSize

Class	⟨⟨**atpObject**⟩⟩ **BaseTypeMaxSize**			
Package	M2::AUTOSARTemplates::CommonStructure::BaseTypes			
Class Desc.	This is the maximum size of a BaseType in case of a dynamic BaseType.			
Base Class(es)	BaseTypeSizeDefinition			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
maxBaseTypeSize	Integer	0..1	aggregation	Describes the maximum length of the BaseType in bits

Table 4.48: BaseTypeMaxSize

The properties of a `BaseType` are:

- For `CATEGORY` only the values `FIXED_LENGTH` and `VARIABLE_LENGTH` are supported. In case of `FIXED_LENGTH BaseTypeSize` is filled with content. In case of `VARIABLE_LENGTH BaseTypeMaxSize` is filled. In both cases the size is specified in bits.
- `baseTypeEncoding` specifies how the values of the base type are encoded. The Supported values for this member are:
  - `1C`: One's complement
  - `2C`: Two's complement

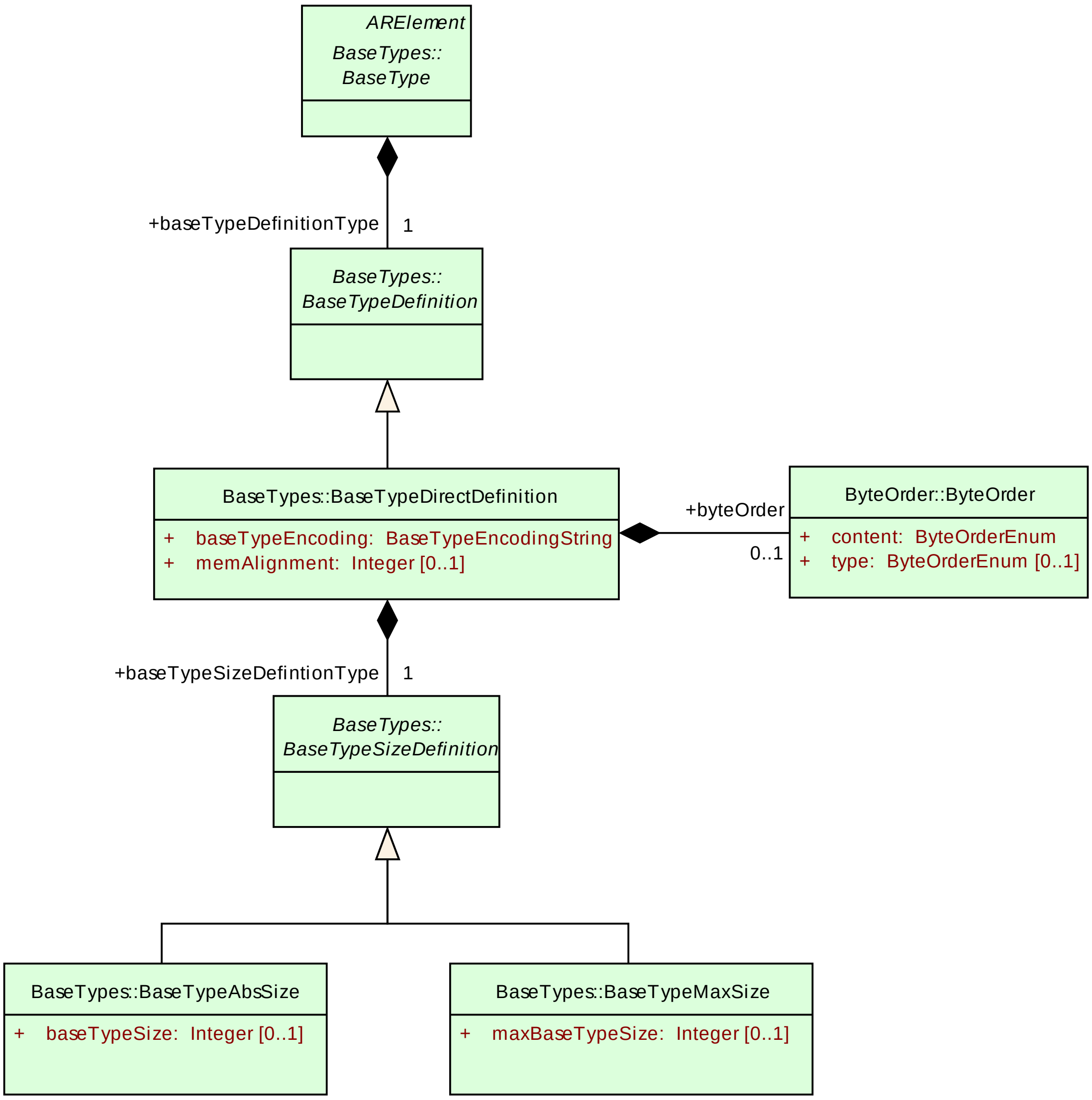


Figure 4.10: `BaseType`

- `BCD-P`: Packed Binary Coded Decimals
- `BCD-UP`: Unpacked Binary Coded Decimals
- `DSP-FRACTIONAL`: Digital Signal Processor
- `SM`: Sign Magnitude
- `IEEE754`: floating point numbers
- `ISO-8859-1`: ASCII-Strings
- `ISO-8859-2`: ASCII-Strings
- `WINDOWS-1252`: ASCII-Strings

  - `UTF-8`: UCS Transformation Format 8
  - `UCS-2`: Universal Character Set 2
  - `NONE`: Unsigned Integer

- `memAlignment` describes the alignment of the memory object in bits. For example, if `memAlignment` is set to 16, the data object in question is aligned to a memory address that can be divided by 2.
- `ByteOrder` specifies the ordering of bits in memory. Possible values are `MOST-SIGNIFICANT-BYTE-FIRST` and `MOST-SIGNIFICANT-BYTE-LAST`.

Class	⟨⟨**atpObject**⟩⟩ **ByteOrder**			
Package	M2::AUTOSARTemplates::CommonStructure::ByteOrder			
Class Desc.	This element specifies the byte order of the parent element. The byte order is defined with the attribute TYPE. Possible values are: * MOST-SIGNIFICANT-BYTE-FIRST * MOST-SIGNIFICANT-BYTE-LAST			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
content	ByteOrder Enum	1	aggregation	

Table 4.49: ByteOrder

5 Internal Behavior

5.1 Introduction

This chapter focuses on the description of the `InternalBehavior` meta-class and the various meta-classes it aggregates. An overview of the meta-class is sketched in figure 5.1.

Class	⟨⟨**atpObject**⟩⟩ **InternalBehavior**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior			
Class Desc.	The internal behavior of an atomic software component describes the RTE relevant aspects of a component, i.e. the runnable entities and the events they respond to.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
component	Atomic Software ComponentType	1	reference	The component this behavior is defined for.
event	RTEEvent	*	aggregation	
exclusive Area	Exclusive Area	*	aggregation	
initValue	LocalParameterInit ValueAssignment	*	aggregation	
inter Runnable Variable	Inter Runnable Variable	*	aggregation	
perInstance Calprm	Calprm Element Prototype	*	aggregation	the perInstanceCalprm is aggregated in the internal behavior, since it is read only. Therefore not protection mechanisms are necessary regardless which runnable performs the access
perInstance Memory	PerInstance Memory	*	aggregation	Defines a per-instance memory object needed by this software component.
portAPI Option	PortAPI Option	*	aggregation	Options for generating the signature of port-related calls from a runnable to the RTE and vice versa.
runnable	Runnable Entity	1..*	aggregation	
service Needs	Service Needs	*	aggregation	the requirements on an AUTOSAR Service defined by this InternalBehavior
shared Calprm	Calprm Element Prototype	*	aggregation	

supportsMultipleInstantiation	Boolean	1	aggregation	Indicate whether the corresponding software-component can be multiply instantiated on one ECU. In this case the attribute will result in an appropriate component API on programming language level (with or without instance handle).

Table 5.1: InternalBehavior

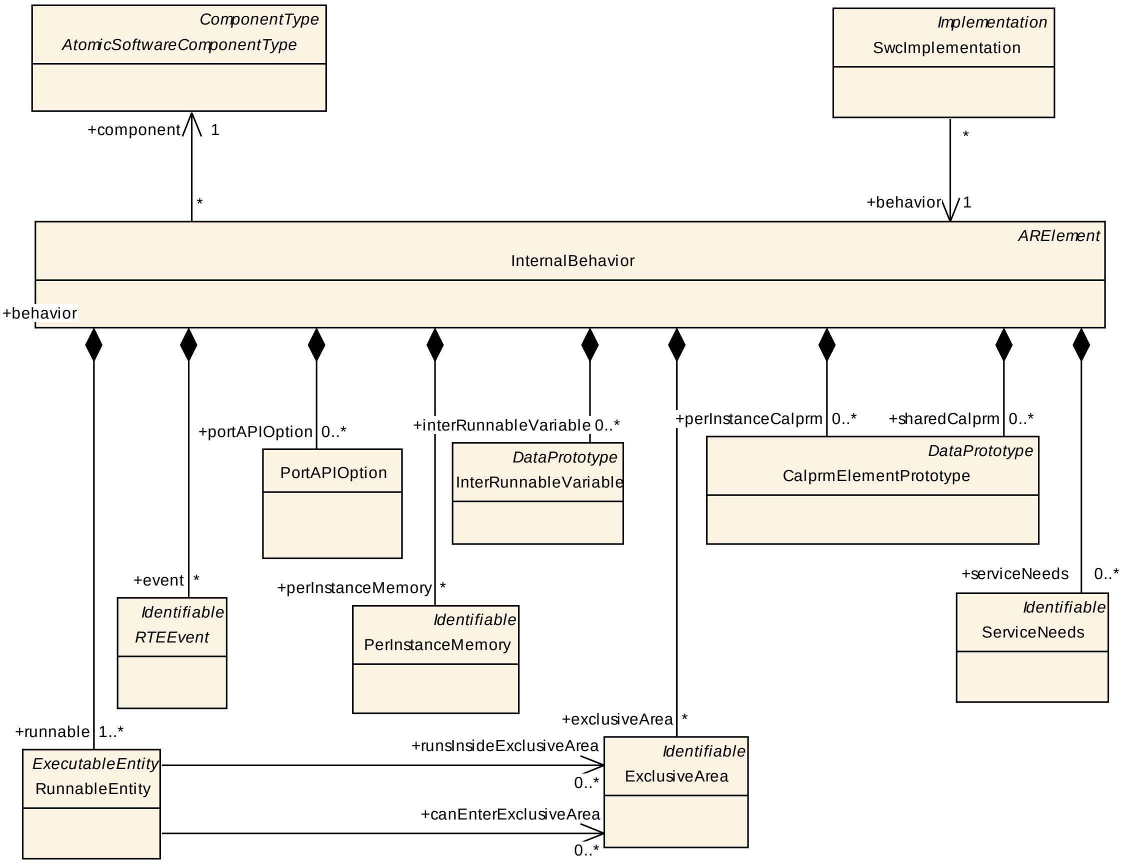


Figure 5.1: `InternalBehavior`

5.2 Runnable Entity

The concept of `RunnableEntity` (more details can be found in figure 5.2) is defined in the specification of the `Virtual Function Bus` [3]. `RunnableEntities` are the smallest code-fragments that are provided by the component and are (at least indirectly) a subject for scheduling by the underlying operating system.

Please note that it is intentionally not possible for `CompositionType` to be referenced by `InternalBehavior`. Consequently, `CompositionTypes` don't have

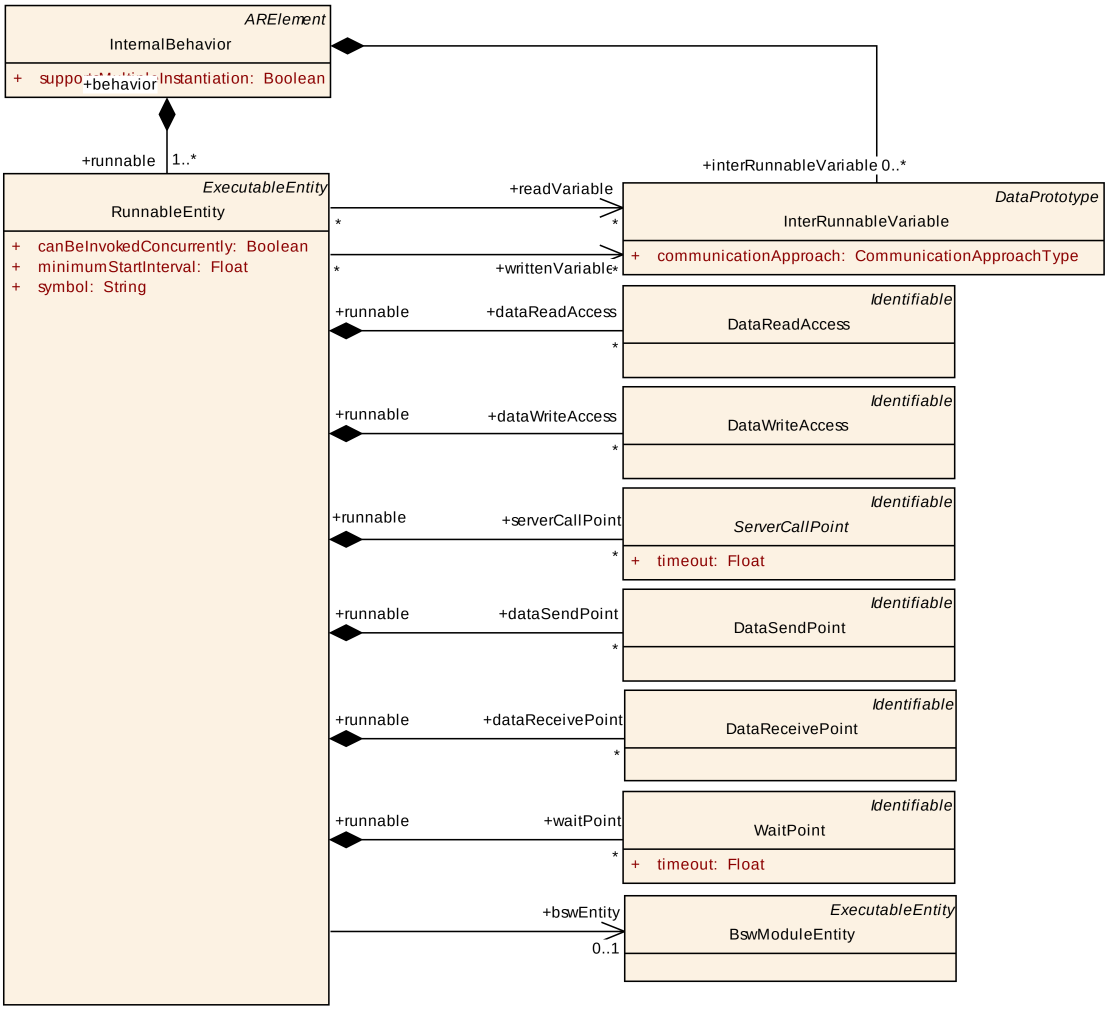


Figure 5.2: Details of `RunnableEntity`

`RunnableEntities` by themselves. Only the `AtomicSoftwareComponentType` that are populating a `CompositionType` in the role of `ComponentPrototypes` may have `RunnableEntities`. This correlation is depicted in Figure 5.3.

Please note that `RunnableEntities` exist in several categories that have different properties. Please find more explanation about categories of `RunnableEntities` in the specification document of the VFB [3]. Note further that this document emphasizes on `RunnableEntities` of category 1A, 1B, and 2.

Class	**⟨⟨atpObject⟩⟩ RunnableEntity**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior			
Class Desc.	The runnable entities are the smallest code-fragments that are provided by the component and are executed in the RTE. Runnables are for instance set up to respond to data reception or operation invocation on a server.			
Base Class(es)	ExecutableEntity			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

bswEntity	BswModuleEntity	0..1	reference	Optional reference to the corresponding BswModuleEntity in case the RunnableEntity is implemented as part of a BSW module (in the case of an AUTOSAR Service, a Complex Device Driver or an ECU Abstraction). It can be used by a tool to find relevant information on the behavior, e.g. whether the bswEntity shall be running in interrupt context.
calprm Access	Calprm Access	*	aggregation	
canBe Invoked Concurrently	Boolean	1	aggregation	Normally, this is FALSE. When this is TRUE, it is allowed that this runnable entity is invoked concurrently (even for one instance of the SW-C), which implies that it is the responsibility of the implementation of the runnable to take care of this form of concurrency.
canEnter Exclusive Area	Exclusive Area	*	reference	This means that the runnable can enter/leave the referenced exclusive area through explicit API calls.
dataRead Access	DataRead Access	*	aggregation	Runnable has read access to data element
dataReceivePoint	DataReceivePoint	*	aggregation	Data receive points of this runnable.
dataSend Point	DataSend Point	*	aggregation	The runnable has data send point.
dataWrite Access	DataWrite Access	*	aggregation	Runnable has write access to data element
minimum StartInterval	Float	1	aggregation	Specifies the time in seconds which two starts of a RunnableEntity are guaranteed to be separated.
mode Switch Point	Mode Switch Point	*	aggregation	The runnable has a mode switch point.
perInstance Calprm Access	Calprm Element Prototype	*	reference	
readVariable	Inter Runnable Variable	*	reference	Inter-runnable variables that are read by this Runnable.

runsInside Exclusive Area	Exclusive Area	*	reference	The runnable entity runs inside the referenced exclusive area
serverCall Point	ServerCall Point	*	aggregation	The runnable has server call point.
shared Calprm Access	Calprm Element Prototype	*	reference	
symbol	String	1	aggregation	The symbol describing this runnable's entry point. This is considered the API of the runnable and is required during the RTE contract phase.
waitPoint	WaitPoint	*	aggregation	The runnable has wait point.
written Variable	Inter Runnable Variable	*	reference	Inter-runnable variables that are written by Runnable.

Table 5.2: RunnableEntity

The attribute `minimumStartInterval` defines the time which the RTE will guarantee between two starts of this RunnableEntity.

Please note that the formal definition of the semantics of a `RunnableEntity` has strong relations to the specification of the AUTOSAR RTE [1]. The definition of the RTE semantics is not in the scope of this document. However, the formal definition requires some background discussion that can't be completely left out of this document. Otherwise the meaning of specific model elements could not be understood properly.

5.2.1 Concurrency and Reentrancy of a RunnableEntity that cannot be Invoked Concurrently

This section applies to the case that the attribute `canBeInvokedConcurrently` is `FALSE`. During runtime, each `RunnableEntity` of each instance of an `AtomicSoftwareComponentType` is (by being a member of an AUTOSAR OS task) in one of three states:

- `Suspended`: the initial state, when the `RunnableEntity` is passive and can be started
- `Enabled`: the `RunnableEntity` should run (because for example a message has been received on a `PortPrototype` of an `AtomicSoftwareComponentType` or a `TimingEvent` occurs).
- `Running`: the `RunnableEntity` is running within a running task. From this state, the `RunnableEntity` can either perform a transition to `Enabled` (if it has been preempted because the task has been preempted) or to `Suspended`.

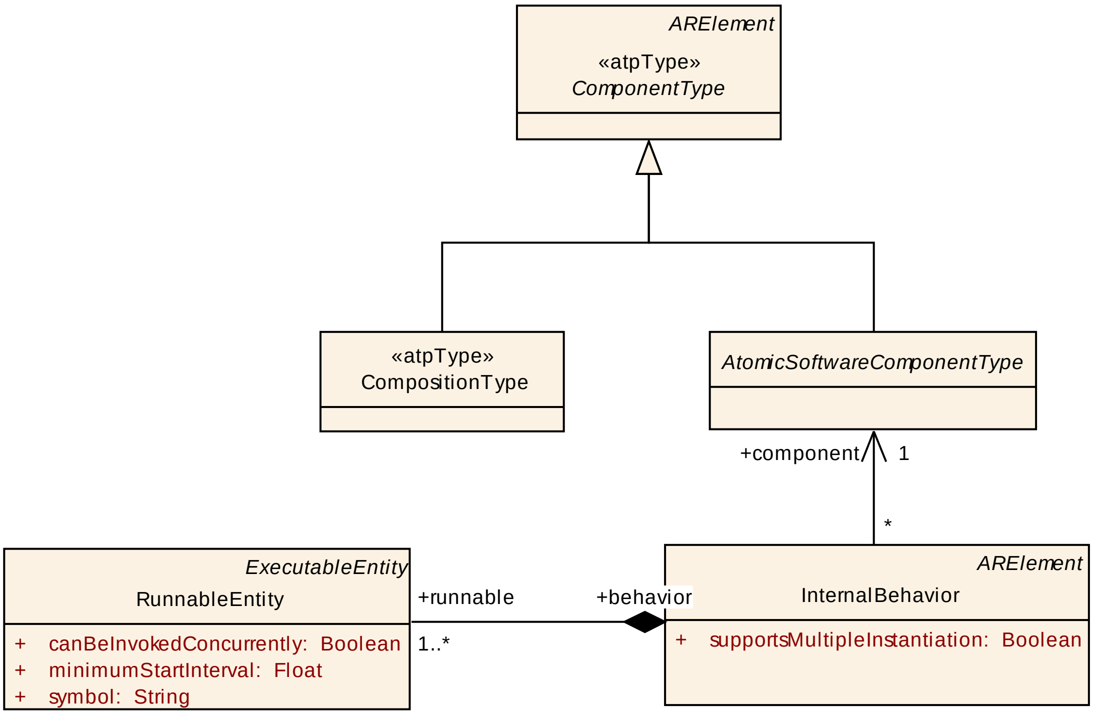


Figure 5.3: Only `AtomicSoftwareComponentTypes` may have `RunnableEntities`

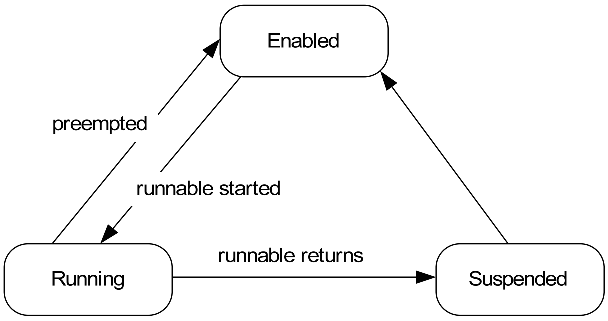


Figure 5.4: Task-derived run-time states of a `RunnableEntity`

The `InternalBehavior` describes for each `RunnableEntity`, when a transition from `Suspended` to `Enabled` should occur. This is done using the concept of an `RTEEvent`.

When a `RunnableEntity` is in state `Enabled`, the RTE can decide to start running the `RunnableEntity`. The delay between entering the state `Enabled` (e.g. a message has been received in response to which the `RunnableEntity` should run) and moving into the state `Running` (the first instruction of the `RunnableEntity` has been executed) depends on the scheduling strategy of the RTE, i.e. the mapping of `RunnableEntities` on AUTOSAR OS tasks.

The transition from the state `Running` into the state `Suspended` is in the hands of the `RunnableEntity`: the transition occurs when the `RunnableEntity` returns (thereby handing over control to the AUTOSAR OS [15]). Some `RunnableEntities` (like cat. 2 `RunnableEntities`) might never return to the ”Suspended” state once they entered the ”Running” state.

They might enter the ”Enabled” state when being preempted. The same applies if a `RunnableEntity` needs to wait for a `WaitPoint` to be unblocked.

Cat. 1A and 1B `RunnableEntities` will typically return after having executed a specific finite algorithm (the execution time of which might be provided).

In most cases `RunnableEntities` will not be scheduled individually but as parts of AUTOSAR OS tasks. Please note that the concept of runtime states as depicted in Figure 5.4 has been created along the example of the OSEK Operating System specification.

In case the internal behavior defines a `RunnableEntity` as one that cannot be invoked concurrently, it is the responsibility of the RTE to make sure that the `RunnableEntity` is never started concurrently (in, for example, two AUTOSAR OS tasks). This implies that the implementation of the `AtomicSoftwareComponentType` does not need to worry about concurrency issues.

For example: The internal behavior of an `AtomicSoftwareComponentType` MyComponentType describes a `RunnableEntity` R1, which should be enabled when an operation on a client-server p-port of the `AtomicSoftwareComponentType` is invoked. The `AtomicSoftwareComponentType` specifies that the `RunnableEntity` R1 cannot be invoked concurrently.

The `AtomicSoftwareComponentType` MyComponentType is instantiated on an ECU. When a call of the operation is received, the corresponding instance of the `RunnableEntity` R1 is enabled and the RTE will start executing the `RunnableEntity` (the `RunnableEntity` is in state `running`) in a task eventually managed by the AUTOSAR OS.

If another call of the operation is received while the `RunnableEntity` is in state `running`, it is not allowed that the RTE runs the `RunnableEntity` again in a second task. Rather, the RTE has to wait (and maybe queue the second incoming request) until the `RunnableEntity` has returned and has moved to the `Suspended` state.

5.2.2 Concurrency and Reentrancy of a RunnableEntity that can be Invoked Concurrently

This section applies to the case that the attribute `canBeInvokedConcurrently` is `TRUE`. In this case, it is allowed that the same `RunnableEntity` is running several times concurrently in different AUTOSAR OS tasks. This implies that the state machine defined in Figure 5.4 is not the state of the `RunnableEntity` any more, but can be cloned an arbitrary number of times.

Note that the software-component description itself does not put any bounds on the number of concurrent invocations of the `RunnableEntity` that are allowed. The software-component description only specifies whether the `RunnableEntity` can be invoked concurrently or not.

Allowing concurrent invocation of a `RunnableEntity` implies that the implementation of the `AtomicSoftwareComponentType` needs to take care of this additional form of concurrency.

For example: The internal behavior of a component-type MyComponentType describes a `RunnableEntity` R1, which should be enabled when an `OperationPrototype` on a `PPortPrototype` typed by a `ClientServerInterface` of the `AtomicSoftwareComponentType` is invoked.

The `AtomicSoftwareComponentType` specifies that the `RunnableEntity` R1 can be invoked concurrently. The `AtomicSoftwareComponentType` MyComponentType is instantiated on an ECU. When a call of the `OperationPrototype` is received, the corresponding instance of the `RunnableEntity` R1 is enabled and the RTE will start executing the `RunnableEntity` (the `RunnableEntity` is in state `running`) in a task eventually managed by the AUTOSAR OS.

If another call of the `OperationPrototype` is received, it is allowed that the same `RunnableEntity` is started again in a different task.

A typical use-case of concurrent `RunnableEntities` are the AUTOSAR services. The AUTOSAR services will typically take care of concurrency internally: several software-components can directly use the services in parallel. The ECU-integrator could then decide that the `RunnableEntity` implementing the AUTOSAR service runs directly in the context (in the task) of the `AtomicSoftwareComponentType` invoking the service.

This is a very efficient, direct coupling between the client and the server: the connector between the client and the server is reduced to a local function-call.

5.2.3 Additional Remarks and Clarifications

5.2.3.1 Reentrancy and Multiple Instantiation

Note that it is useful to consider the combinations of the attributes `supportsMultipleInstantiation` and `canBeInvokedConcurrently`.

supportsMultiple-Instantiation	canBeInvoked-Concurrently	Implication for an implementation of a `RunnableEntity`
`FALSE`	`FALSE`	This implies that the implementation of the `RunnableEntity` will never be invoked concurrently from several tasks. The implementation does not need to care about reentrancy issues and can typically use `static variables` to store state.
`TRUE`	`FALSE`	In case there are several instances of the same `AtomicSoftwareComponentType` on the local ECU, the implementation of the `RunnableEntity` can still be invoked concurrently from several tasks. However, there will be no concurrent invocations of the implementation with the same `instance handle`. To ensure that this is safe, the implementation will typically use per-instance memory.
`FALSE/TRUE`	`TRUE`	In this case the `RunnableEntity` can be invoked concurrently from several tasks, even with the same instance handle.

Note that the combination of `supportsMultipleInstantiation`=FALSE and `canBeInvokedConcurrently`=FALSE is only uncritical in case each `RunnableEntity` is implemented by its own `C`-function.

In case the `AtomicSoftwareComponentType` implementation decides to map several `RunnableEntities` to the same `symbol` there are reentrancy problems to be sorted out. However, this scenario is not supported by the RTE [1] anyway and must therefore be avoided.

5.2.3.2 Reentrancy and "Library Functions"

Note that all code that is called by different `RunnableEntities` (like e.g. library routines, etc.) must obviously be reentrant. A filter algorithm implemented in C, for example, is not allowed to store values from previous runs by means of static variables or variables with external binding.

5.2.4 Timed Activation of Runnable Entities

In many cases, `RunnableEntities` need to be activated in response to timing events rather than related to communication (e.g. the reception of a response to an asynchronous operation invocation). Many `RunnableEntities` will need to run cyclically with a fixed rate.

The approach taken in the software-component description is to define so-called `TimingEvents` (please find more details in figure 5.5) as special kinds of `RTEEvents`. So far, only one kind of timing-related `RTEEvent` has been defined: a simple periodic `TimingEvent`.

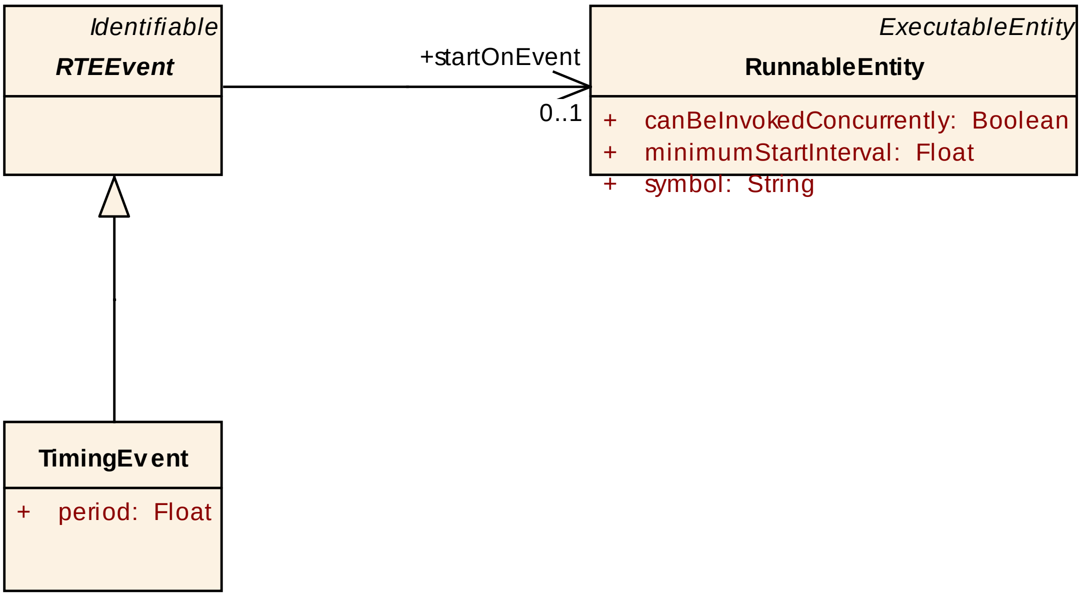


Figure 5.5: Periodic activation of RunnableEntities

Therefore, if the `InternalBehavior` of an `AtomicSoftwareComponentType` requires that the RTE executes certain `RunnableEntities` periodically, the description needs to define a `TimingEvent` with the desired period. This `TimingEvent` then contains a reference to the Runnable that needs to be executed with this period.

Class	⟨⟨**atpObject**⟩⟩ **TimingEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	TimingEvent references the runnable that need to be started in response to the TimingEvent			
Base Class(es)	RTEEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
period	Float	1	aggregation	Period of timing event in seconds.

Table 5.3: TimingEvent

5.3 RTEEvent

During execution, several `RTEEvents` will occur, such as the reception of a remote invocation of an `OperationPrototype` on a `PPortPrototype` or a timeout on an

`RPortPrototype` that is not receiving the `DataElementPrototypes` it expects to receive. Describing an `RTEEvent` includes two aspects:

1. defining an `RTEEvent`
2. defining how the RTE should deal with the `RTEEvent` when it occurs.

Class	⟨⟨**atpObject**⟩⟩ **RTEEvent (abstract)**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	Abstract base class for all RTE-related events			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
modeDependency	Mode Disabling Dependency	0..1	aggregation	Provides the means to describe the Modes this RTEEvent can be disabled by.
startOn Event	Runnable Entity	0..1	reference	Runnable starts when event occurs

Table 5.4: RTEEvent

Class	⟨⟨**atpObject**⟩⟩ **AsynchronousServerCallReturnsEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	This event is raised when an asynchronous server call is finished.			
Base Class(es)	RTEEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
event Source	Asynchronous ServerCall Point	1	reference	The referenced server call point

Table 5.5: AsynchronousServerCallReturnsEvent

Class	⟨⟨**atpObject**⟩⟩ **DataSendCompletedEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	The event is raised when the referenced data elements have been sent or an error occurs.			
Base Class(es)	RTEEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
event Source	DataSend Point	1	reference	Data send point that triggers the event.

Table 5.6: DataSendCompletedEvent

Class	⟨⟨**atpObject**⟩⟩ **DataReceivedEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	The event is raised when the referenced data elements are received.			
Base Class(es)	RTEEvent			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
data	DataElement Prototype	1	instanceRef	Data element referenced by event

Table 5.7: DataReceivedEvent

Class	⟨⟨**atpObject**⟩⟩ **DataReceiveErrorEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	This event is raised by the RTE when the Com layer detects and notifies an error concerning the reception of the referenced data element.			
Base Class(es)	RTEEvent			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
data	DataElement Prototype	1	instanceRef	Data element referenced by event

Table 5.8: DataReceiveErrorEvent

Class	⟨⟨**atpObject**⟩⟩ **OperationInvokedEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	The OperationInvokedEvent references the OperationPrototype invoked by the client.			
Base Class(es)	RTEEvent			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
operation	Operation Prototype	1	instanceRef	The operation to be executed as the consequence of the event.

Table 5.9: OperationInvokedEvent

Class	⟨⟨**atpObject**⟩⟩ **TimingEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events			
Class Desc.	TimingEvent references the runnable that need to be started in response to the TimingEvent			
Base Class(es)	RTEEvent			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
period	Float	1	aggregation	Period of timing event in seconds.

Table 5.10: TimingEvent

Class	⟨⟨**atpObject**⟩⟩ **ModeSwitchEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTEEvents			
Class Desc.	This event is listening to mode changes coming from the StateManager.			
Base Class(es)	RTEEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
activation	ModeActivationKind	1	aggregation	Specifies if the event is activated on entering or exiting the referenced Mode.
mode	ModeDeclaration	1	instanceRef	Reference to the Mode that initiates the Mode Switch Event.

Table 5.11: ModeSwitchEvent

Class	⟨⟨**atpObject**⟩⟩ **ModeSwitchedAckEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTEEvents			
Class Desc.	The event is raised when the referenced mode have been received or an error occurs.			
Base Class(es)	RTEEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
eventSource	ModeSwitchPoint	1	reference	Mode switch point that triggers the event.

Table 5.12: ModeSwitchedAckEvent

As described in the Virtual Functional Bus specification [3], the `RunnableEntities` of an `AtomicSoftwareComponentType` can interact with the occurrence of such `RTEEvents` in two ways:

- the RTE can be instructed to enable a specific `RunnableEntity` when the `RTEEvent` occurs
- the RTE can provide `WaitPoints`, that allow a `RunnableEntity` to block until an `RTEEvent` in a set of `RTEEvents` occurs.

5.3.1 Defining an Event

The description of the `InternalBehavior` includes a description of all `RTEEvents` that the `InternalBehavior` of the `AtomicSoftwareComponentType` relies on. This `RTEEvent` shows up as an ”abstract” base-class (see Figure 5.6) in the meta-model: the exact attributes of the `RTEEvent` depend on the specific sub-class of `RTEEvent` that is used for the purpose.

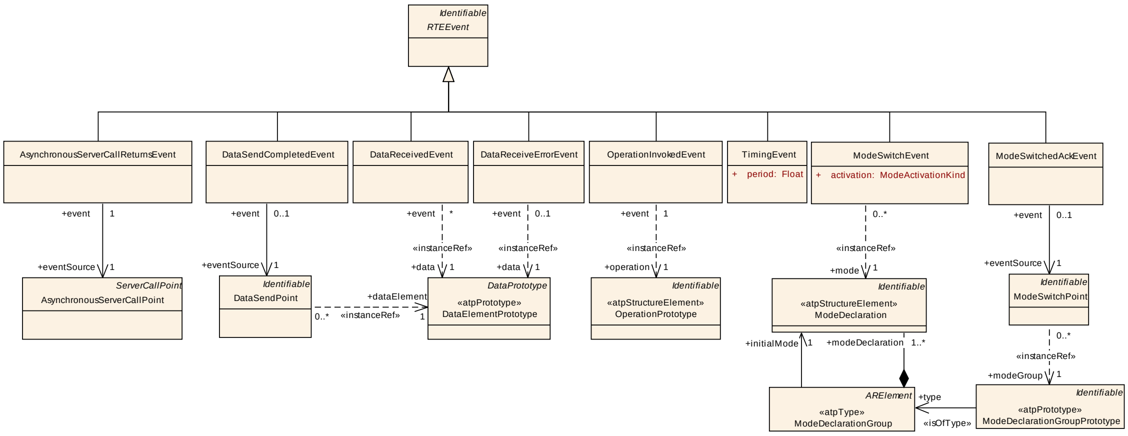


Figure 5.6: Kinds of `RTEEvents`

The details of the various kinds of concrete `RTEEvents` (such as the `TimingEvent`, `DataSendCompletedEvent`, etc.), is described in chapters 3.6.2, 3.6.3 and 5.2.4.

5.3.2 Defining how to Respond to an Event

If the software-component description contains a reference from an `RTEEvent` to a `RunnableEntity` it is the responsibility of the RTE to trigger the execution of the corresponding `RunnableEntity` when the `RTEEvent` occurs.

In case the `RunnableEntity` wants to block and wait for `RTEEvents` (which makes the `RunnableEntity` into a cat. 2 `RunnableEntity`), the description of the `RunnableEntity` may include the definition of a `WaitPoint`.

Such a `WaitPoint` (see Figure 5.7) contains a reference to all `RTEEvents` that can unblock the specific `WaitPoint`. In other words: the `WaitPoint` will block until one of the referenced `RTEEvents` occurs.

A single `RunnableEntity` can actually wait only at a single `WaitPoint` provided that the `RunnableEntity` can only be scheduled a single time[1]. On the other hand, it is in general possible that a single `RTEEvent` can be used to trigger `WaitPoints` in different `RunnableEntities`.

Class	⟨⟨**atpObject**⟩⟩ **WaitPoint**
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTE Events
Class Desc.	This defines a wait-point for which the runnable can wait.
Base Class(es)	Identifiable

---

[1] This constraint is valid at least in the OSEK standard where an extended task (that can have wait points) can only exist a single time in the context of the scheduler.

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
timeout	Float	1	aggregation	Time in seconds before the waitpoint times out and the blocking wait call returns with an error indicating the timeout.
trigger	RTEEvent	1	reference	Events this wait point is waiting for.

Table 5.13: WaitPoint

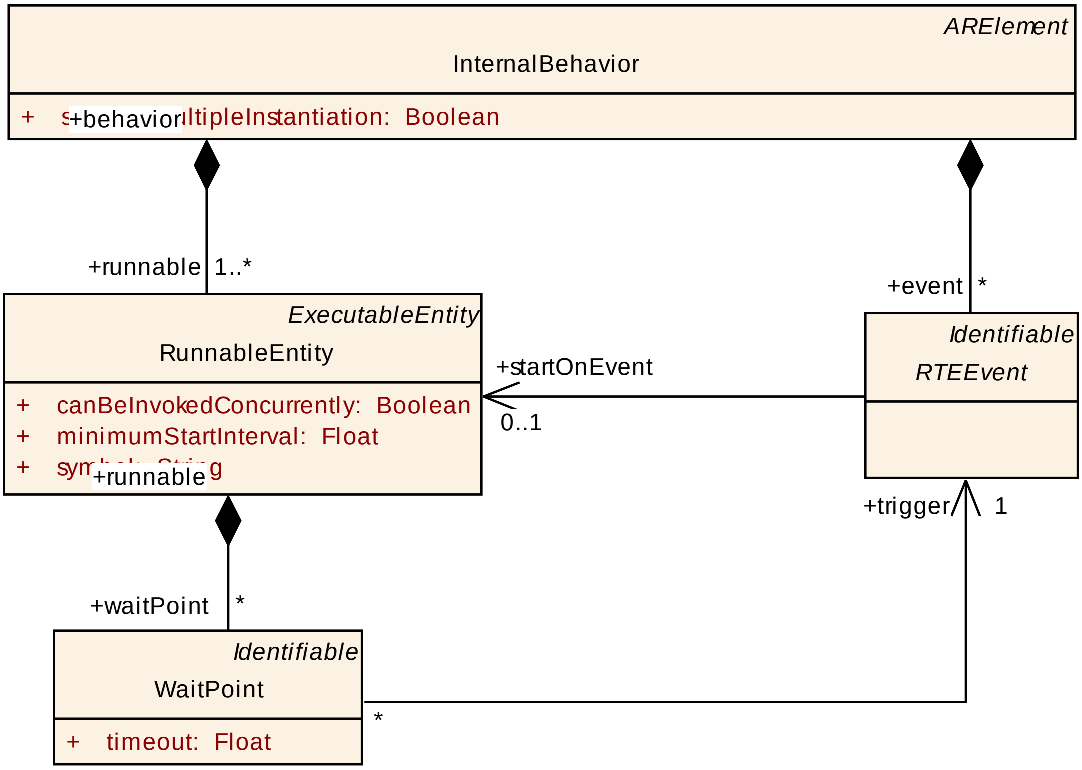


Figure 5.7: Description of the interaction between an `RTEEvent` and `RunnableEntities`

5.4 Communication among Runnable Entities

It is taken for granted that particular `RunnableEntities` within a specific `AtomicSoftwareComponentType` will need to communicate among each other. This implies that the RTE need to provide synchronization mechanisms to the `RunnableEntities` such that safe (in the multi-threading sense) exchange of data is possible.

Several concepts for implementing communication among `RunnableEntities` can be identified. As an introduction, this section first describes the various techniques that the RTE might use to provide efficient interaction between `RunnableEntities` within one `AtomicSoftwareComponentType`.

Next, two possible approaches for formal specification of this kind of communication are described:

- Specifying that several `RunnableEntities` belong in a specific `ExclusiveArea`
- Specifying the data exchanged between the `RunnableEntities`

5.4.1 Background: the Issues

This section gives some background information and lists possible strategies concerning the implementation of the `RunnableEntities` and the RTE w.r.t. efficient communication between the `RunnableEntities`.

The communication among `RunnableEntities` can very efficiently be implemented by means of ”sharing memory”[2].

This is technically feasible because it is always guaranteed that the `RunnableEntities` within an `AtomicSoftwareComponentType` are always gathered at a specific processing unit (in other words: distribution is not an option).

Note that the purpose of communication among the `RunnableEntities` is to establish a data flow scheme. The latter is a very popular pattern in the application of control theory to automotive embedded systems. So if ”global variables” are used for establishing internal communication among `RunnableEntities` they acquire the semantics of so called state-messages.

Nevertheless, directly sharing memory between `RunnableEntities` requires a serious problem to be solved: the guarantee of data consistency among communicating `RunnableEntities`. The `RunnableEntities` will indeed be mapped to tasks so that one `RunnableEntity` of an `AtomicSoftwareComponentType` may be preempted by a different `RunnableEntity` of the same `AtomicSoftwareComponentType`.

Please note that a purist approach to achieving data consistency not only applies to single accesses of concurrently accessed variables. Rather, it would not be permitted that the value of a concurrently accessed variable (with state-message semantics) is unintentionally changed during the runtime of a `RunnableEntity`.

The following paragraphs describe some common strategies that can be used to ensure the required data-consistency. We do not attempt to describe the pros or cons of these approaches.

5.4.1.1 Mutual Exclusion with Semaphores

Multi-threaded operating systems provide mutexes (mutual exclusion semaphores) that protect access to an exclusive resource that is used from within several tasks.

The RTE could use these OS-provided mutexes to make sure that the `RunnableEntities` sharing a memory-space would never run concurrently. The RTE would make sure the task running the `RunnableEntity` has taken an appropriate mutex before accessing the memory shared between the `RunnableEntities`.

---

[2] Please note that the term ”sharing memory” can be interpreted on different levels. It is e.g. in the C language possible to use variables with external linkage (a.k.a. ”global variables”, although this term is not officially defined by the C language) for the purpose of inter-Runnable communication.

5.4.1.2 Interrupt Disabling

Another alternative would be the disabling of interrupts during the run-time of `RunnableEntities` or at least for a period in time identical to the interval from the first to the last usage of a concurrently accessed variable in a `RunnableEntity`. This approach could lead to seriously non-deterministic execution timing.

5.4.1.3 Priority Ceiling

Priority ceiling allows for a non-blocking protection of shared resources. Provided that the priority scheme is static, the AUTOSAR OS is capable of temporarily raising the priority of a task that attempts to access a shared resource to the highest priority of all tasks that would ever attempt to access the resource.

By this means is technically impossible that a task in temporary possession of a resource is ever preempted by a task that attempts to access the resource as well.

5.4.1.4 Implicit Communication by Means of Variable Copies

Another alternative is the usage of copies of concurrently accessed variables with state message semantics. Note that this approach directly corresponds to the semantics of ”implicit” sender-receiver communication (see 3.6.2.2).

This means in particular that for a concurrently used variable a copy is created on which a `RunnableEntity` entity can work without any danger of data inconsistency.

This concept requires additional code to write the value of the concurrently accessed variable to the copy before the `RunnableEntity` that accesses the variable is executed. The value of the copy must be written back to the concurrently accessed variable after the `RunnableEntity` has been terminated.

This concept is sketched in Figure 5.8. Since it would be too expensive and error-prone to manually care about the copy routines it would be a good idea to leave the creation of the additional code to a suitable code generator.

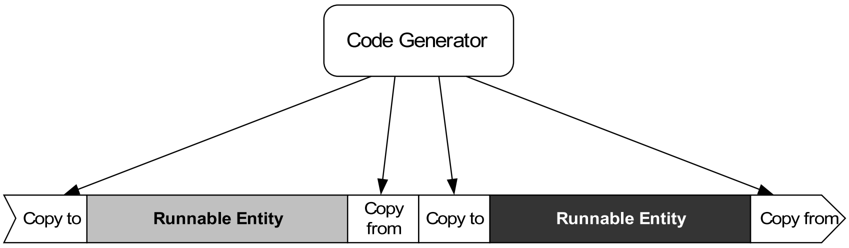


Figure 5.8: Generation of copy routines around `RunnableEntities`

The additional copy routines as sketched in Figure 5.8 already protect the particular `RunnableEntities` from unintended changes of concurrently accessed variables. It would, however, be possible to further optimize the process by reducing the additional code at the beginning and end of each task (see Figure 5.9).

In addition, copy routines will only be inserted where appropriate, e.g. a copy routine for writing the value of a copy back to the concurrently accessed variable will only be inserted if the `RunnableEntity` has write access to the concurrently used variable.

Please note that the copy routines have to temporarily make sure that the copy process is not interrupted in order to be capable of consistently copying the values from and to the concurrently accessed variable. These periods, however, are supposed to be very short compared with the overall run-time consumption of the `RunnableEntity` and thus would not have a significant impact on the runtime behavior.

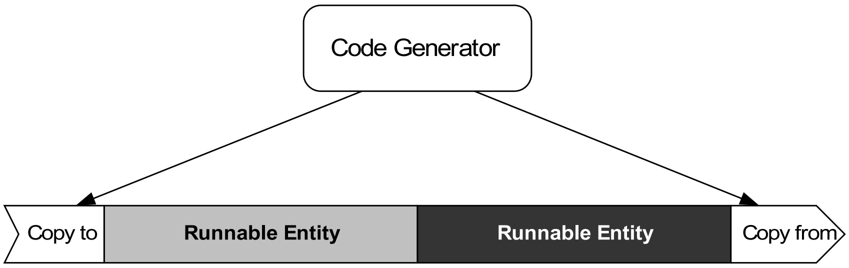


Figure 5.9: Optimized insertion of copy routines

Further optimization criteria can be applied, for example: it would be perfectly safe to avoid the creation of copies for runnables that are scheduled in the task with the highest priority of all tasks that (via contained runnables) access a certain concurrently accessed variable.

In order to keep the application code free of any dependencies from the code generation, access to concurrently accessed variables will be guarded by macros that are later resolved by the code generator.

The presence of the guard macros directly supports the reuse on the level of source code. The reuse on the level of object code is only possible if the scheduling scenario (in terms of the assignment of `RunnableEntities` to priority levels) does not change.

This concept can only be implemented properly with the aid of a code generator if the variables in question can be identified. In other words: the description of an `AtomicSoftwareComponentType` has to expose all concurrently accessed variables to the outside world.

5.4.2 Description possibility 1: Exclusive Area

This section describes how the concept of `ExclusiveAreas` can be used in the description of the `InternalBehavior` of an `AtomicSoftwareComponentType`. These `ExclusiveAreas` do not imply a specific implementation (e.g. with mutual-exclusion semaphores).

Class	**⟨⟨atpObject⟩⟩ ExclusiveArea**			
Package	M2::AUTOSARTemplates::CommonStructure::InternalBehavior			
Class Desc.	Prevents an executable entity running in the area from being preempted.			
Base Class(es)	Identifiable			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**

Table 5.14: ExclusiveArea

An `ExclusiveArea` (please find details about the formal definition of this meta-class in figure 5.10) merely specifies a constraint on the scheduling policy and configuration of the RTE: If two or more `RunnableEntities` refer to the same `ExclusiveArea` only one of these `RunnableEntities` is allowed to be executed while being inside that `ExclusiveArea`.

In other words: these `RunnableEntities` must not run concurrently (preempt each other) while executing inside the `ExclusiveArea`.

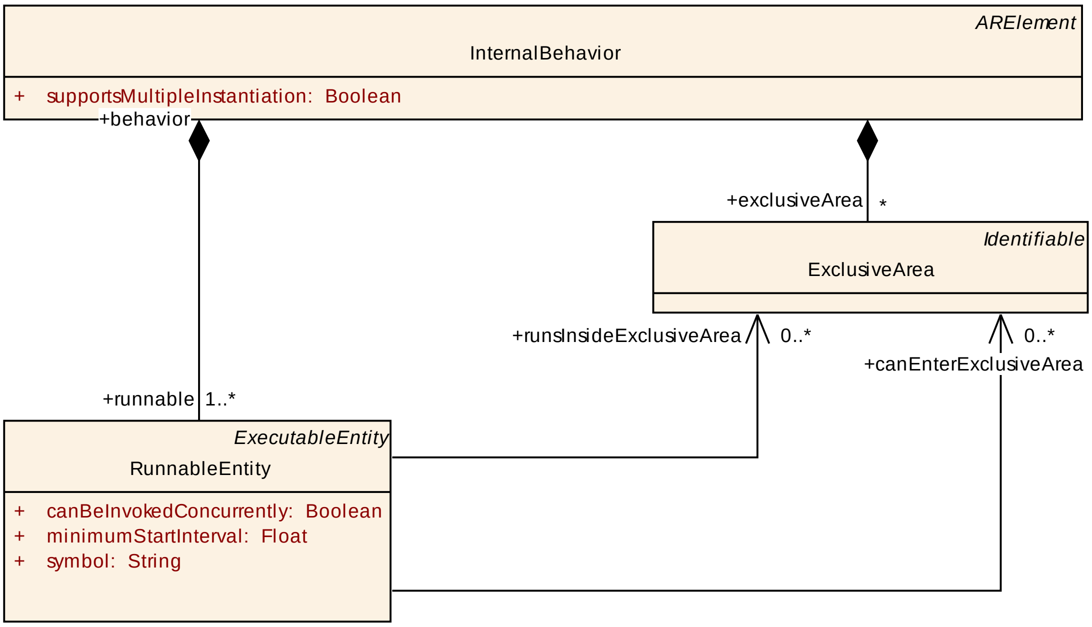


Figure 5.10: Description of logical exclusive areas

There are in general two ways to use the `ExclusiveAreas`. Note that it is even possible to use a specific `ExclusiveArea` in one `RunnableEntity` according to chapter 5.4.2.1 while another `RunnableEntity` might go for accessing the `ExclusiveArea` according to chapter 5.4.2.2.

5.4.2.1 Entire Runnable Runs in the Exclusive Area

In the first approach, the formal description specifies that certain `RunnableEntities` always run inside an exclusive area. For example, if the formal description specifies that both `RunnableEntity` 'r1' and `RunnableEntity` 'r2' run within `ExclusiveArea` 's1', the RTE must make sure that `RunnableEntities` 'r1' and 'r2' never run concurrently; the scheduler should never preempt 'r1' to run 'r2'.

Note that this pattern does not force the RTE to implement this by using semaphores or mutexes that are taken before the `RunnableEntity` starts and given when the `RunnableEntity` returns. It only obliges the RTE to make sure that both `RunnableEntities` are never running concurrently.

This requirement could be implemented by several of the implementation strategies described above. For example:

1. Scheduling strategy: if, for example, `RunnableEntities` 'r1' and 'r2' are mapped to the same task, the criterion is automatically satisfied. For this pur-

pose it is necessary to make sure that the OS can only execute a single instance of the task into which the `RunnableEntities` are put.

2. Mutual exclusion semaphores: in case ’r1’ and ’r2’ are mapped to different tasks (’T1’, respectively ’T2’), the OS must make sure that while ’T1’ is executing ’r1’, ’T2’ running ’r2’ can never preempt it and vice-versa. This could be implemented by taking a mutual-exclusion semaphore before executing ’r1’ (resp. ’r2’) in the context of ’t1’ (resp. ’t2’) and returning the semaphore on exiting the `RunnableEntity`.

5.4.2.2 Runnable would Dynamically Enter and Leave the Exclusive Area

In the second approach, the `RunnableEntity` would explicitly make API-calls to the RTE within the implementation of the `RunnableEntity` to enter and leave a specific `ExclusiveArea`. This could, for example, be implemented by means of the priority ceiling concept described in chapter 5.4.1.3.

Additionally it is possible to define the execution time the `RunnableEntity` will spend in this `ExclusiveArea` segment. Please note that although this aspect is described in [8] the concept can be applied to software-components as well.

5.4.3 Description possibility 2: Inter-Runnable Variable

For certain important strategies (like the ”variable copies” described above) the `ExclusiveArea` concept does not provide enough information to configure the RTE correctly.

The concept of copying concurrently accessed variables is very efficient and can even be used in ambitious automotive applications like, for example, engine management.

Please note however, that a certain amount of RAM has to be reserved for the copies. This is obviously a slight drawback of the concept.

Concerning the introduction in the AUTOSAR meta-model, data required for communication among `RunnableEntities` needs to be explicitly identified (`InterRunnableVariable`). Furthermore, the relationship of these data with `RunnableEntities` must be specified. For this purpose references with role `send` and `receive` from `RunnableEntity` to `InterRunnableVariable` are introduced.

`InterRunnableVariables` must have a data type; therefore the meta-class `InterRunnableVariable` is derived from `DataPrototype`.

Class	⟨⟨**atpPrototype**⟩⟩ **InterRunnableVariable**
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::InterRunnableCommunication
Class Desc.	Implement state message semantics for establishing communication among runnables of the same component.

Base Class(es)	DataPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
communicationApproach	CommunicationApproachType	1	aggregation	Communication among RunnableEntities resembles the approaches taken for the communication among software components. The explicit communication corresponds to DataReceivePoint/DataSendPoint. The implicit communication resembles DataReadAccess/DataWriteAccess
initValue	ValueSpecification	0..1	reference	

Table 5.15: InterRunnableVariable

Please note that it is possible to define an initial value for a specific `InterRunnableVariable`. For this purpose the AUTOSAR meta-model features an association between an `InterRunnableVariable` and a `ValueSpecification` in the role of an `initValue` (see Figure 5.11).

The behavior is undefined if no initial value is specified and a `RunnableEntity` reads an `InterRunnableVariable` before it is actually written to by another `RunnableEntity`.

As already mentioned before, the concept of `InterRunnableVariables` can be used in *two different flavors* (indicated by the attribute `communicationApproach`) that resemble the communication principles applied for the communication on the level of `ComponentTypes`.

Please note that the attribute directly controls the usage of RTE API calls and is therefore obligatory for any subsequent process step, especially the ECU configuration. A subsequent tool (e.g. ECU configuration editor) must under no circumstances ignore or change the settings made for `communicationApproach`.

The semantics of the attribute is that *explicit* implies the direct access to the value of an `InterRunnableVariable`. By this means it is possible to get different values for a specific `InterRunnableVariable` each time the corresponding API call is executed.

The setting *implicit* corresponds to an execution model where the value of an `InterRunnableVariable` does not change (for the reading `RunnableEntity`, obviously) during the runtime of a `RunnableEntity`. This approach is in detail described in chapter 5.4.1.4.

5.5 Port API Options

The RTE Generator needs additional options per `PortPrototype` to choose the proper generation schema. These are subsumed in the `PortAPIOption` element which is shown in Figure 5.12.

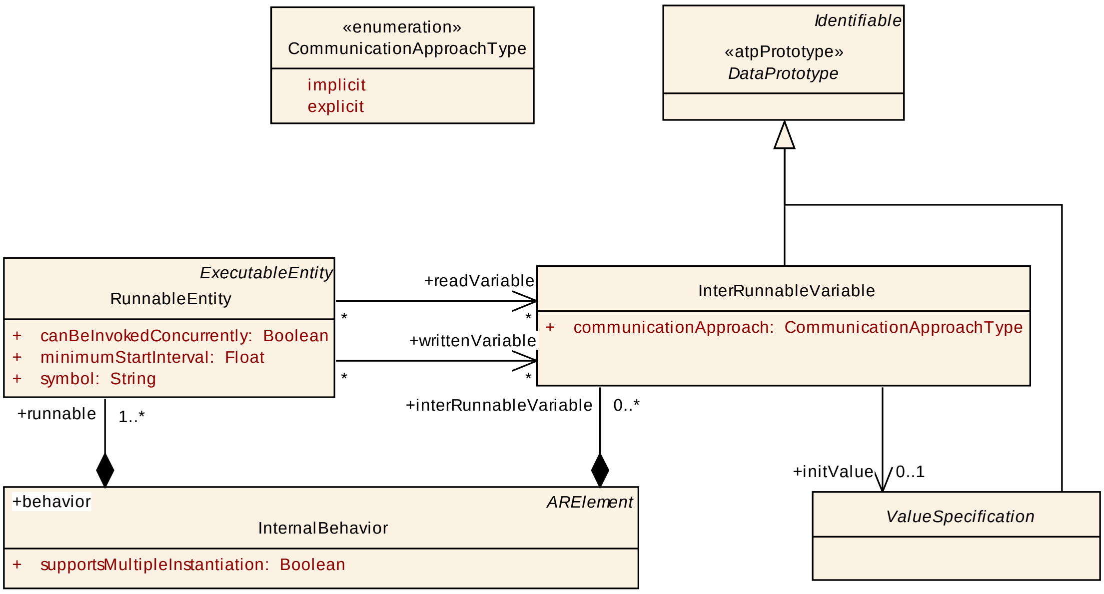


Figure 5.11: `InterRunnableVariable`

Class	⟨⟨**atpObject**⟩⟩ **PortAPIOption**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::PortAPI Options			
Class Desc.	Options how to generate the signatures of calls for an AtomicSoftwareComponentType in order to communicate over a PortPrototype (for calls into a RunnableEntity as well as for calls from a RunnableEntity to the PortPrototype).			
Base Class(es)	ARObject			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
enable TakeAddress	Boolean	1	aggregation	If set to true, the software-component is able to use the API reference for deriving a pointer to an object.
indirectAPI	Boolean	1	aggregation	true: Specifies an ”indirect API” to be generated for the associated port, which means that the SWC is able to access the actions on a port via a pointer to an object representing a port. This allows e.g. iterating over ports in a loop. This option has no effect for PPorts of client/server interfaces.
port	PortPrototype	1	reference	the option is valid for generated functions related to communication over this port
portArg Value (ordered)	Primitive Specification	*	aggregation	A ”port defined argument values” is passed to a runnable dealing with the operations provided by a given port. Restricted to PPorts of a client/server interface.

Table 5.16: PortAPIOption

5.5.1 Enable to TakeAddress

If `enableTakeAddress = TRUE` the generated API related to this `PortPrototype` is provided in a way that the software component is able to used the API reference for deriving an pointer to an object.

5.5.2 Indirect API Generation

The `indirectAPI` option switches the generation of the RTE's indirect API functionality for a certain `PortPrototype`. The generated indirect API does allow to iterate over ports within the SW-Component.

5.5.3 Port Defined Argument Value

In addition to the formal parameters of a client/server invocation that are defined as part of the server's `PortInterface`, it is possible to specify a number of implicit values that are passed by the RTE to the server's entry point.

The initial need for this feature arises in the context of basic software services, although it is not limited to those. For a service like the NVRAM manager every accessing port is in addition to its logical identity as a sequence of `ShortNames` - uniquely identified through a NVRAM specific memory block id.

Instead of exposing this mechanism on the logical `ClientServerInterface` level in form of a formal `Argument`, one or more port-defined arguments can be specified. This way, the implementation detail is hidden from the logical component designer.

Figure 5.12 shows the meta-model of Port API Options and the `portArgValue`. The values are primitive types, typically integer values to specify an id. In case of the NVRAM example this list would have just one value of type int8 holding the memory block id.

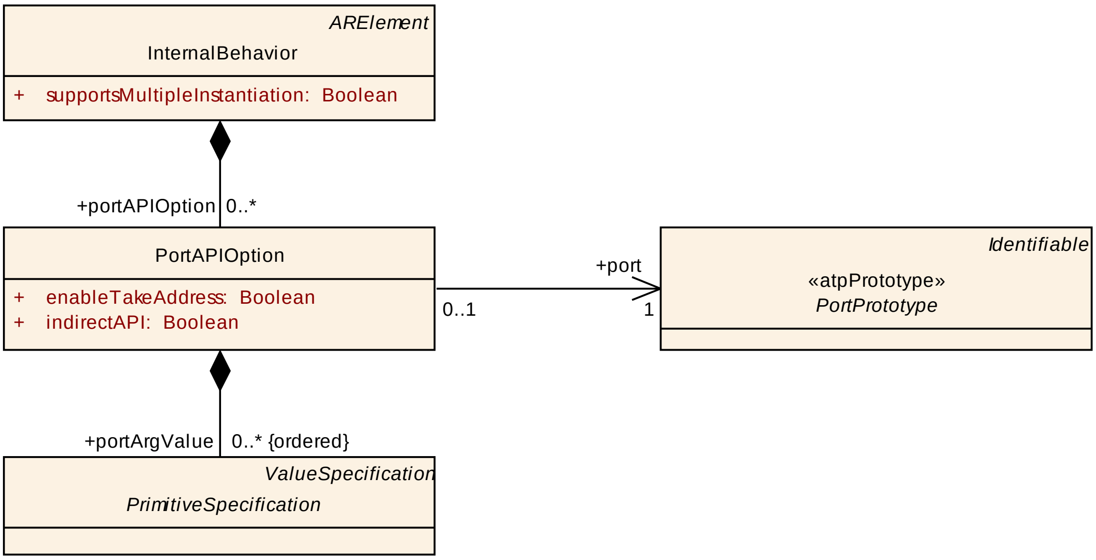


Figure 5.12: Port API Options.

5.6 PerInstanceMemory

`AtomicSoftwareComponentTypes` that support multiple instantiation (attribute `supportsMultipleInstantiation == TRUE`) will typically need a given amount of private memory per instance. It is the responsibility of the RTE to provide a mechanisms with which each instance of an `AtomicSoftwareComponentType` can access its own instance-specific memory.

An `AtomicSoftwareComponentType` can define an arbitrary number of per-instance memory blocks (formally defined by aggregating the meta-class `PerInstanceMemory`).

For each such memory block, the software-component description must provide the name of the data type (the ”C”-type) it needs to store in the memory block. This attribute allows for the RTE to generate an API function that provides a convenient and type-safe access to the data item.

In addition, the software-component description must define the data type in the attribute `typeDefinition`. This attribute is supposed to contain a *C* typedef of the data type in valid C-syntax. In other words, this `typeDefinition` must be formulated such that it can be included verbatim in a C header file.

Note that the `PerInstanceMemory` is not explicitly initialized by the RTE. Instead, it is the responsibility of the `AtomicSoftwareComponentType` to initialize the `PerInstanceMemory`.

More details on the use of these attributes in the generation of software-component header-files can be found in the RTE specification [1].

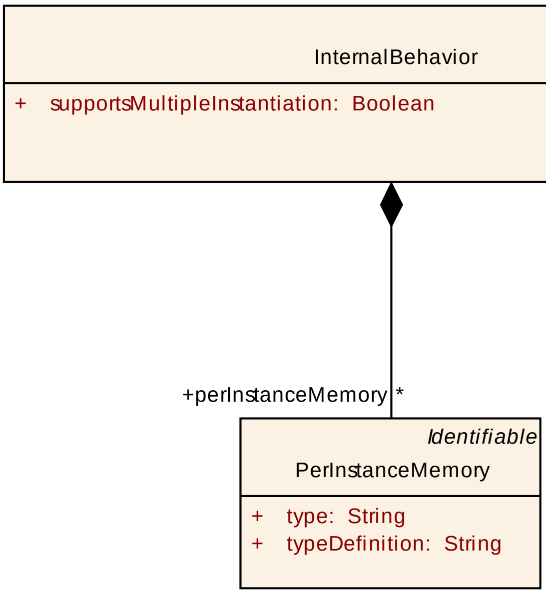


Figure 5.13: `PerInstanceMemory`

`AtomicSoftwareComponentTypes` that do *not* support multiple instantiation (attribute `supportsMultipleInstantiation == FALSE`) do not necessarily need to use the `PerInstanceMemory`: because there will only be a single instance of the `AtomicSoftwareComponentType` on an ECU, the `AtomicSoftwareComponentType` can use static variables to store the `AtomicSoftwareComponentType`'s internal state. However, the usage of `PerInstanceMemory` is also allowed in this case.

Class	⟨⟨**atpObject**⟩⟩ **PerInstanceMemory**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::PerInstanceMemory			
Class Desc.	Defines a memory-block that needs to be available for each instance of the SW-component. This is typically only useful if supportsMultipleInstantiation is TRUE of if the component defines NVRAM access via permanent blocks.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
type	String	1	aggregation	The "C"-type
typeDefinition	String	1	aggregation	A definition of the type

Table 5.17: PerInstanceMemory

5.7 Service Needs

5.7.1 Overview

`ApplicationSoftwareComponentTypes` are designed to be independent of their mapping to actual ECU Hardware. However, each software-component might need services which are provided by the ECU's Basic Software through AUTOSAR Services. The `ServiceNeeds` (see figure 5.14) are used to provide detailed information what the software-component expects from the AUTOSAR Services when integrated on an actual ECU. Note that only atomic software-components can be connected to AUTOSAR Services.

When integrating application software-components on an ECU, the actual values of ECU configuration parameters must be chosen so that they fulfill the requirements given by the `ServiceNeeds` of all the integrated atomic software-components.

Note that the actual values of configuration parameters will in addition depend on the properties of the basic software and the hardware of that specific ECU, see also chapter 10. For further information about the relation between the `ServiceNeeds` and the ECU configuration parameters see [16].

The meta-class `ServiceNeeds` and the sub-classes for several Services are located in the `CommonStructure` package of the meta-model, because they are also used in the `Basic Software Module Description Template` [8]. Note that `ServiceNeeds` is not abstract, which allows to use it via textual information also for those AUTOSAR Services for which no sub-classes are defined.

Class	⟨⟨**atpObject**⟩⟩ **ServiceNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	This expresses the abstract needs that a Software Component or Basic Software Module has on the configuration of an AUTOSAR Service to which it will be connected. ”Abstract needs” means, that the model abstracts from the Configuration Paramaters of the underlying Basic Software.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 5.18: ServiceNeeds

`ServiceNeeds` specified by `AtomicSoftwareComponentTypes` are part of the `InternalBehavior` because in special cases they can have associations to other parts of the `InternalBehavior` like `RunnableEntity` or `PerInstanceMemory`. In most cases they are also related to certain ports belonging to the `AtomicSoftwareComponentTypes` (or more precisely, one of its non-abstract derived meta-classes) of this `InternalBehavior`, because `AtomicSoftwareComponentTypes` communicate with AUTOSAR Services via those ports.

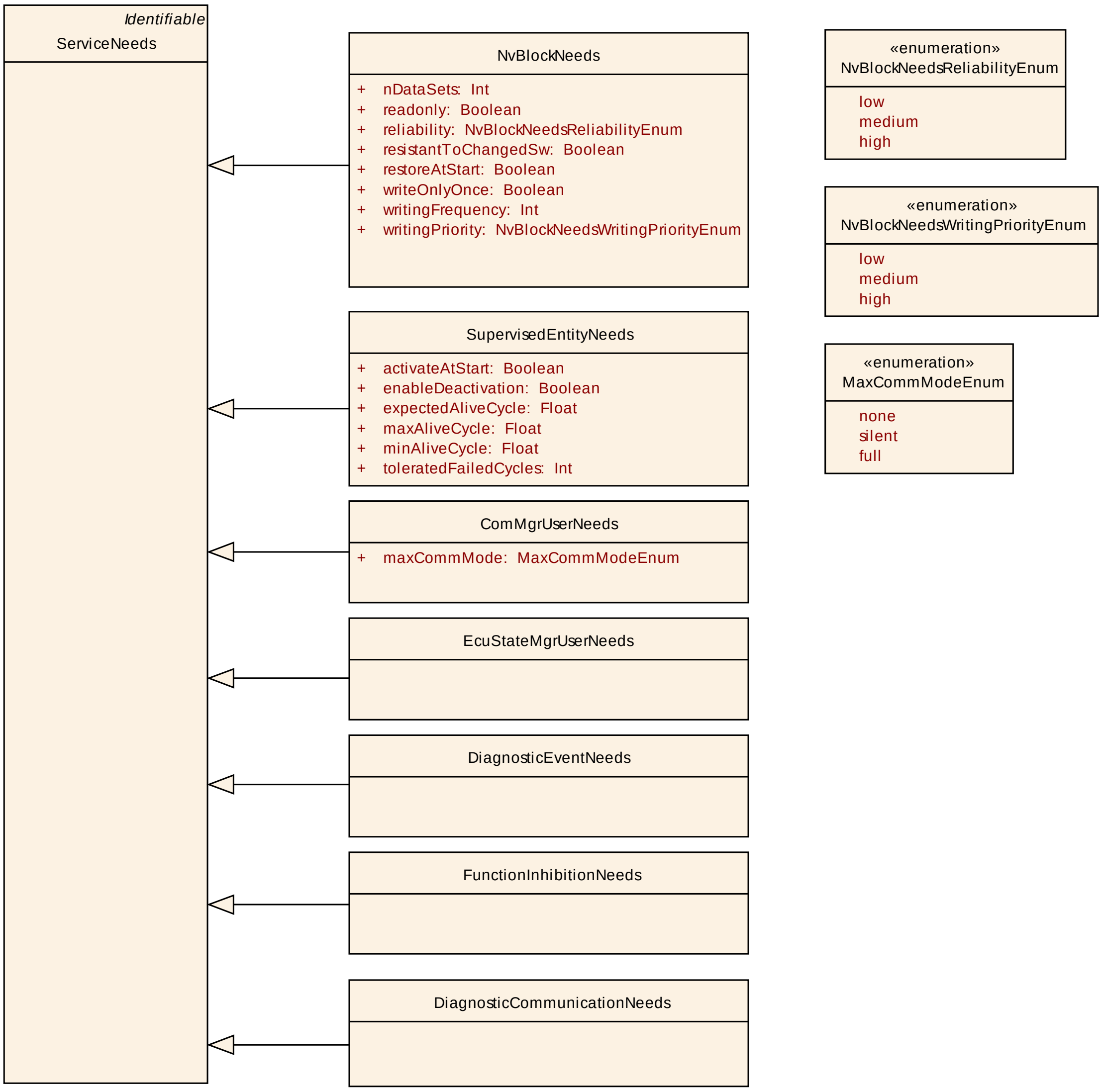


Figure 5.14: `ServiceNeeds`: Common structure

This relationship to ports is defined via `RoleBasedRPortAssignment` for `RPortPrototype` and `RoleBasedPPortAssignment` for `PPortPrototype`. `RoleBasedRPortAssignment` and `RoleBasedPPortAssignment` are aggregating the attribute `role`.

Class	**⟨⟨atpObject⟩⟩ RoleBasedRPortAssignment**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Service Mapping			
Class Desc.	This class specifies an assignment of a role to a particular R-Port. This port must contain a service which is outside of the component and called by the component in order to handle a particular issue (e.g. a communication event).			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

rPortPrototype	RPort Prototype	1	reference	Port which requires the software component to be connected to an AUTOSAR Service.
role	Identifier	1	aggregation	This is the role the assigned Port in given context. The value must be a name of a PortInterface as standardized in Software Specification of the related AUTOSAR Service.

Table 5.19: RoleBasedRPortAssignment

Class	⟨⟨**atpObject**⟩⟩ **RoleBasedPPortAssignment**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Service Mapping			
Class Desc.	This class specifies an assignment of a role to a particular P-Port. This port must contain a service which is inside of the component and called by outside entity in order to handle a particular issue (e.g. a communication event). This is often named as callback.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
pPortPrototype	PPort Prototype	1	reference	Port which provides the software component to be connected to an AUTOSAR Service.
role	Identifier	1	aggregation	This is the role of the assigned Port in the given context. The value must be a name of a PortInterface as standardized in the Software Specification of the related AUTOSAR Service.

Table 5.20: RoleBasedPPortAssignment

The attribute `role` specifies the role of the `PortPrototype` in the interaction of the software-component with the `AUTOSAR Service` and is required for the generation of Service-related Model Elements, see chapter 10.

In order to define these special associations, further sub-classes exist which are used to describe the detailed `ServiceNeeds` in the scope of the `InternalBehavior` of an `AtomicSoftwareComponentType`. They are explained in the next sub-sections together with the generic classes for the individual Services.

5.7.2 Service Needs for the NVRAM Service

Figure 5.15 and the following class tables show the meta-classes `NvBlockNeeds` and `SwcNvBlockNeeds` which are used to define requirements and special associations needed to configure the NVRAM Service. An `AtomicSoftwareComponentType` may provide several `SwcNvBlockNeeds` elements, each defines all the mappings for one NV Block (for the terms related to the AUTOSAR NVRAM Manager see [17]).

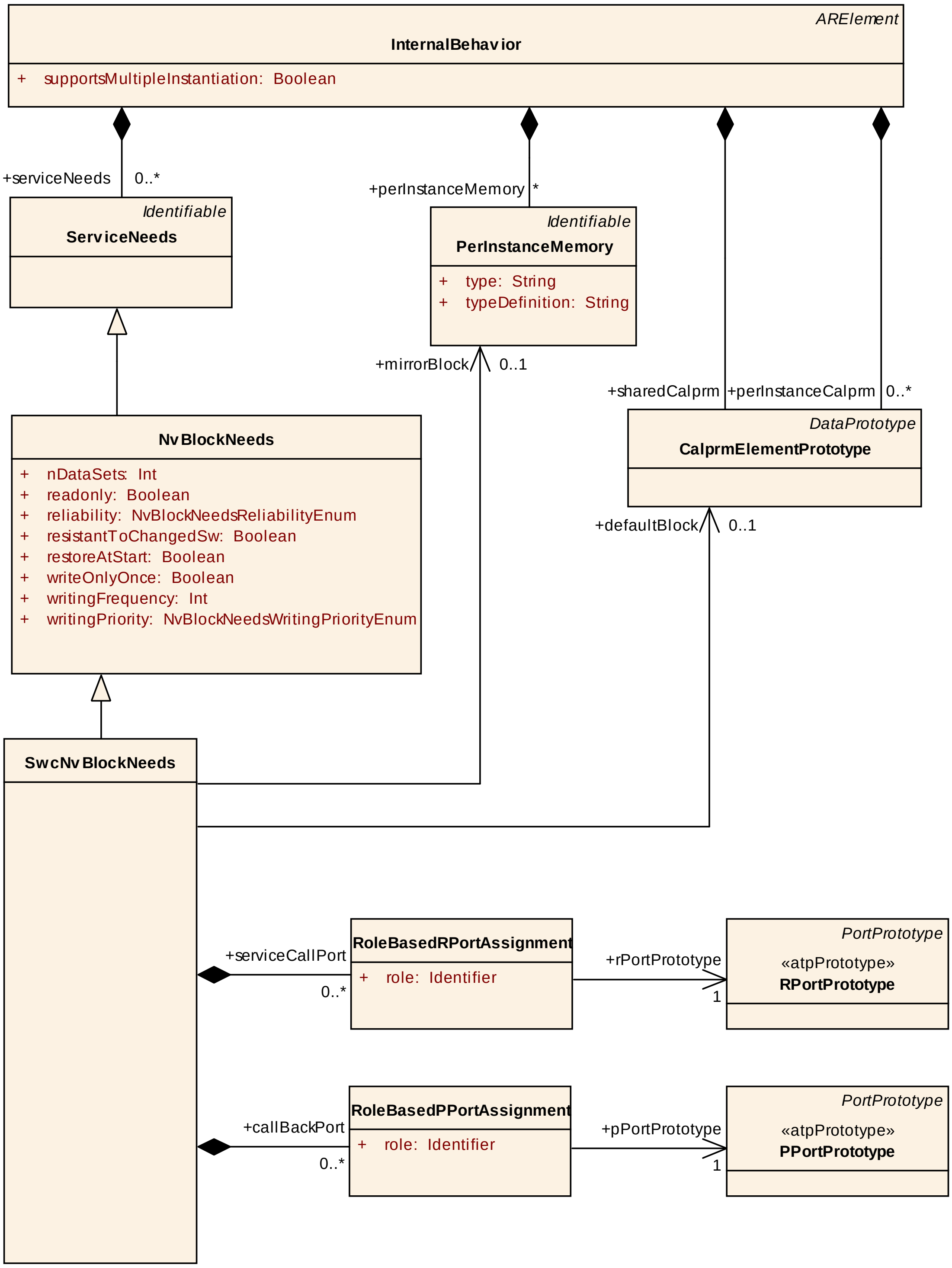


Figure 5.15: **`SwcNvBlockNeeds`**

Class	⟨⟨**atpObject**⟩⟩ **NvBlockNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of a single Nv block.			
Base Class(es)	ServiceNeeds			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
nDataSets	Integer	1	aggregation	number of data sets to be provided by the NVRAM manager for this block
readonly	Boolean	1	aggregation	true: data of this block are write protected for normal operation (but protection can be disabled) false: no restriction
reliability	NvBlockNeedsReliabilityEnum	1	aggregation	Reliability against data loss on the non-volatile medium.
resistantToChangedSw	Boolean	1	aggregation	Defines whether an Nv block shall be treated resistant to configuration changes (true) or not (false). For details how to handle initialization in the latter case, refer to the NVRAM specification.
restoreAtStart	Boolean	1	aggregation	Defines whether the associated RAM mirror block shall be implictly restored during startup by the basic SW or not. Only relevant if a RAM mirror block (PerInstanceMemory) is associated with this port.
writeOnlyOnce	Boolean	1	aggregation	Defines write protection after first write: true: This block is prevented from being changed/erased or being replaced with the default ROM data after first initialization by the SWC. false: No such restriction.
writingFrequency	Integer	1	aggregation	Provides the amount of updates to this block from the application point of view. It has to be provided in "number of write access per year".
writingPriority	NvBlockNeedsWritingPriorityEnum	1	aggregation	Requires the priority of writing this block in case of concurrent requests to write other blocks.

Table 5.21: NvBlockNeeds

Class	⟨⟨**atpObject**⟩⟩ **SwcNvBlockNeeds**
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::ServiceMapping

Class Desc.	Specialization of NvBlockNeeds for the case it is owned by a SoftwareComponentType. It specifies all mappings to elements of the SoftwareComponentType concerning a single Nv block. Note that the mapping is the same for all instances of a SoftwareComponentType (because the code depends on it). Note that the block size is not specified here because - it can be derived from the associated PerInstanceMemory size (implementatiion specific) in case of implicit storage/restauration of the block - if can be derived from the array size passed via the correponding operations of the Service Interface in case of explicit storage/restauration of the block			
Base Class(es)	NvBlockNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
callBack Port	RoleBased PPortAssignment	*	aggregation	This is the provided service to be called by the NvRam Manager to handle a particular NvBlock. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification of NVRAM Manager” (e.g. something like ”NvMNotify”)
default Block	Calprm Element Prototype	0..1	reference	Defines the ROM default for an Nv block. This data can be also calibratable.
mirror Block	PerInstance Memory	0..1	reference	Defines the RAM mirror in case of a permanant Nv block.
serviceCall Port	RoleBased RPortAssignment	*	aggregation	This is the expected service to be called by the software component to handle a particular NvBlock. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification of NVRAM Manager” (e.g. something like ”NvMAdministration” , ”NvMService”)

Table 5.22: SwcNvBlockNeeds

For each NV Block the NVRAM Manager can be configured to use a RAM area as mirror for the access of the NV Block content at runtime. It is the responsibility of the NVRAM Manager to provide the content of the NV Block in this RAM mirror during startup and write back the content to the storage medium during shut-down.

If an `AtomicSoftwareComponentType` is using the RAM mirror feature, a `PerInstanceMemory` section is used as mirror for each NV Block. The `PerInstanceMemory` section is allocated by the RTE during ECU Configuration. If the `AtomicSoftwareComponentType` is using some NV Blocks without a RAM mirror it is the responsibility of the `AtomicSoftwareComponentType` to provide a memory area available to the API call to the NVRAM Manager for storage of the NV data.

5.7.3 Service Needs for the Watchdog Service

Figure 5.16 and the following class table show the meta-classes `SupervisedEntityNeeds` and `SwcSupervisedEntityNeeds` which are used to define requirements and special associations needed to configure the Watchdog Service. An `AtomicSoftwareComponentType` may provide several `SwcSupervisedEntityNeeds` elements, each defines all the mappings for one supervised entity (for the terms related to the AUTOSAR Watchdog Manager see [18]).

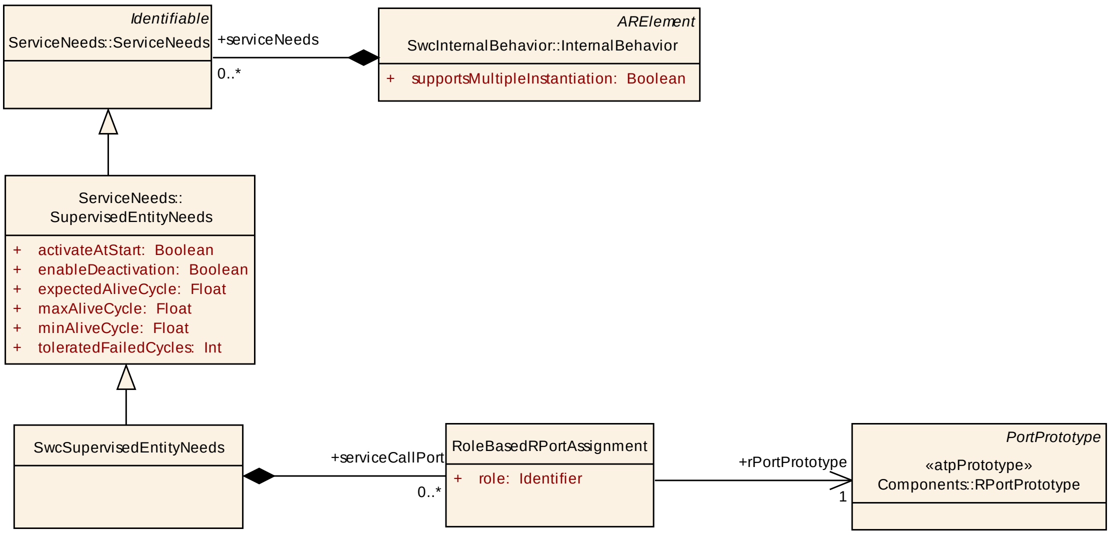


Figure 5.16: **`SwcSupervisedEntityNeeds`**

Class	⟨⟨**atpObject**⟩⟩ **SupervisedEntityNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of the Watchdog Manager for one specific Supervised Entity (SE).			
Base Class(es)	ServiceNeeds			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
activateAtStart	Boolean	1	aggregation	true/false: supervision activation status of SE shall be enabled/disabled at start
enableDeactivation	Boolean	1	aggregation	true: SWC shall be allowed to deactivate supervision of this SE false: not
expectedAliveCycle	Float	1	aggregation	Expected cycle time of alive trigger of this SE (in seconds)
maxAliveCycle	Float	1	aggregation	Maximum cycle time of alive trigger of this SE (in seconds)
minAliveCycle	Float	1	aggregation	Minimum cycle time of alive trigger of this SE (in seconds)

toleratedFailedCycles	Integer	1	aggregation	Number of consecutive failed alive cycles for this SE which shall be tolerated until the supervision status of the SE is set to EXPIRED (see WdgM documentation for details). Note that this has to be recalculated w.r.t. the WdgMs own cycle time for ECU configuration.

Table 5.23: SupervisedEntityNeeds

Class	⟨⟨**atpObject**⟩⟩ **SwcSupervisedEntityNeeds**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::ServiceMapping			
Class Desc.	Specialization of SupervisedEntityNeeds for the case it is owned by a SoftwareComponentType.			
Base Class(es)	SupervisedEntityNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
serviceCallPort	RoleBasedRPortAssignment	*	aggregation	This is the expected service to be called by the software component to handle a supervised entity by the watchdoc. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification of Watchdog Manager” (e.g. something like ”WdgMService”)

Table 5.24: SwcSupervisedEntityNeeds

5.7.4 Service Needs for the ComM Service

Figure 5.17 and the following class tables show the meta-classes `ComMgrUserNeeds` and `SwcComMgrUserNeeds` which are used to define requirements and special associations needed to configure the ComM Service. An `AtomicSoftwareComponentType` may provide several `SwcComMgrUserNeeds` elements, each defines all the mappings for one ”user” of the ComM Service (for the terms related to the AUTOSAR Communication Manager see [19]).

Class	⟨⟨**atpObject**⟩⟩ **ComMgrUserNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of the Communication Manager for one ”user”.			
Base Class(es)	ServiceNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
maxCommMode	MaxCommModeEnum	1	aggregation	Maximum communication mode requested by this ComM user

Table 5.25: ComMgrUserNeeds

Class	⟨⟨**atpObject**⟩⟩ **SwcComMgrUserNeeds**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Service Mapping			
Class Desc.	Specialization of the ComMgrUserNeeds for the case it is owned by a SoftwareComponentType.			
Base Class(es)	ComMgrUserNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
callback Port	RoleBased PPortAssignment	*	aggregation	This is the provided service to be called by the Com Manager to handle a particular communication channel of the Com Manager. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification ot Com Manager” (e.g. something like ”modeRequester”)
serviceCall Port	RoleBased RPortAssignment	*	aggregation	This is the expected service to be called by the software component to handle a particular Com Manger event. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification ot Com Manager” (e.g. something like ”modeRequester”)

Table 5.26: SwcComMgrUserNeeds

5.7.5 Service Needs for the EcuM Service

Figure 5.18 and the following class tables show the meta-classes `EcuStateMgrUserNeeds` and `SwcEcuStateMgrUserNeeds` which are used to define special associations needed to configure the ECU State Manager Service. An `AtomicSoftwareComponentType` may provide several `SwcEcuStateMgrNeeds` elements, each defines all the mappings for one ”user” of the EcuM Service (for the terms related to the AUTOSAR ECU State Manager see [20]).

Class	⟨⟨**atpObject**⟩⟩ **EcuStateMgrUserNeeds**
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds
Class Desc.	Specifies the abstract needs on the configuration of the ECU State Manager for one ”user”. This class currently contains no attributes. Its name can be regarded as a symbol identifying the user from the viewpoint of the component or module which owns this class.
Base Class(es)	ServiceNeeds

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 5.27: EcuStateMgrUserNeeds

Class	**⟨⟨atpObject⟩⟩ SwcEcuStateMgrUserNeeds**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Service Mapping			
Class Desc.	Specialization of the EcuStateMgrUserNeeds for the case it is owned by a SoftwareComponentType. It allows to navigate to all the ports which are used by this component to put requests for this ”user”. Note that there are further ports which a component can use to obtain various information from the ECU State Manager. These ports are not included in the mapping because they will be implemented as pure function calls which can be called independently of being a certain”user”. Note that the AUTOSAR ECU State Manager does not support callbacks to services provided by users of ECU State Manger, therefore there is not property ”callbackPort”.			
Base Class(es)	EcuStateMgrUserNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
serviceCall Port	RoleBased RPortAs-signment	*	aggregation	This is the expected service to be called by the software component to handle a particular User of the Ecu State Manager.. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification ot ECU State Manager”. Examples are ”CurrentMode”, ”ShutdownTarget”, ”BootTarget” , ”ApplicationMode”, ”StateRequest”.

Table 5.28: SwcEcuStateMgrUserNeeds

5.7.6 Service Needs for the DEM Service

Figure 5.19 and the following class tables show the meta-classes `DiagnosticEventNeeds` and `SwcDiagnosticEventNeeds` which are used to define special associations needed to configure the Diagnostic Event Manager Service. An `AtomicSoftwareComponentType` may provide several `SwcDiagnosticEventNeeds` elements, each defines all the mappings for one diagnostic event (for the terms related to the AUTOSAR Diagnostic Event Manager see [21]).

Class	**⟨⟨atpObject⟩⟩ DiagnosticEventNeeds**
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds

Class Desc.	Specifies the abstract needs on the configuration of the Diagnostic Event Manager for one diagnostic event. Its name can be regarded as a symbol identifying the diagnostic event from the viewpoint of the component or module which owns this class.			
Base Class(es)	ServiceNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 5.29: DiagnosticEventNeeds

Class	**⟨⟨atpObject⟩⟩ SwcDiagnosticEventNeeds**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Service Mapping			
Class Desc.	Specialization of the DiagnosticEventNeeds for the case it is owned by a SoftwareComponentType. It allows to navigate to all ports associated with this diagnostic event. Note that there may be further ports to communicate with the DEM Service (e.g. setting the operation cycle type) which are not included in this mapping because they are independent of the diagnisic event.			
Base Class(es)	DiagnosticEventNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
callback Port	RoleBased PPortAssignment	*	aggregation	This aggregation specifies the expected service to be called by the Diagnostic Event Manager. The value of the role attribute in the aggregated class must be be a name of a PortInterface as standardized in ”Specification of Diagnostics Event Manager”, for example CallbackInitMonitorForEvent.
serviceCall Port	RoleBased RPortAssignment	*	aggregation	This is the expected service to be called by the software component to handle a particular diagnostic event. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification of Diagnostics Event Manager”, for example ”DiagnosticMonitor”.

Table 5.30: SwcDiagnosticEventNeeds

5.7.7 Service Needs for the FIM Service

Figure 5.20 and the following class table show the meta-classes `FunctionInhibitionNeeds` and `SwcFunctionInhibitionNeeds` which are used to define special associations needed to configure the Diagnostic Event Manager Service. An `AtomicSoftwareComponentType` may provide several

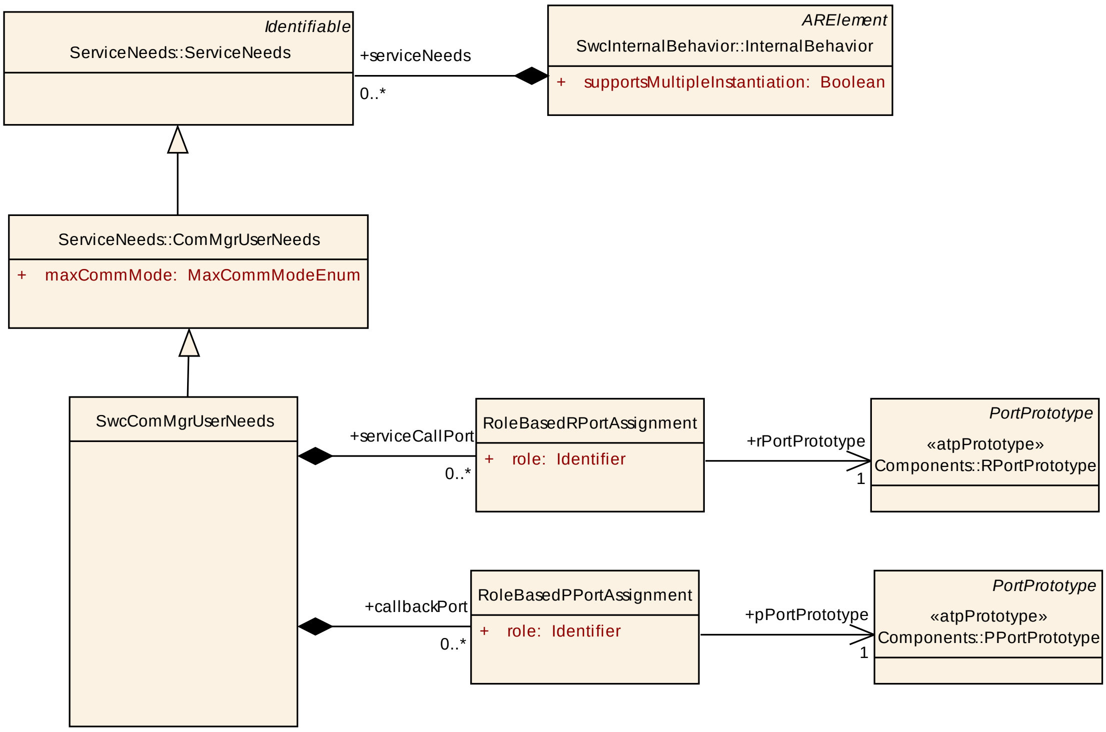


Figure 5.17: `SwcComMgrUserNeeds`

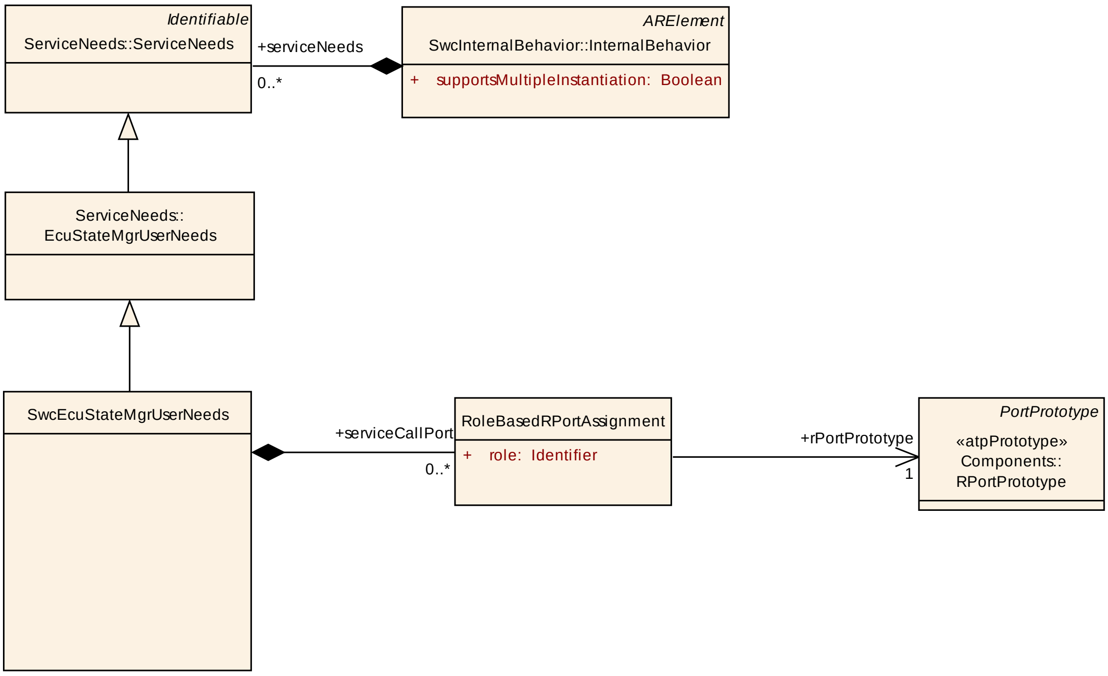


Figure 5.18: `SwcEcuStateMgrUserNeeds`

`FunctionInhibitionNeeds` elements, each defines all the mappings for one

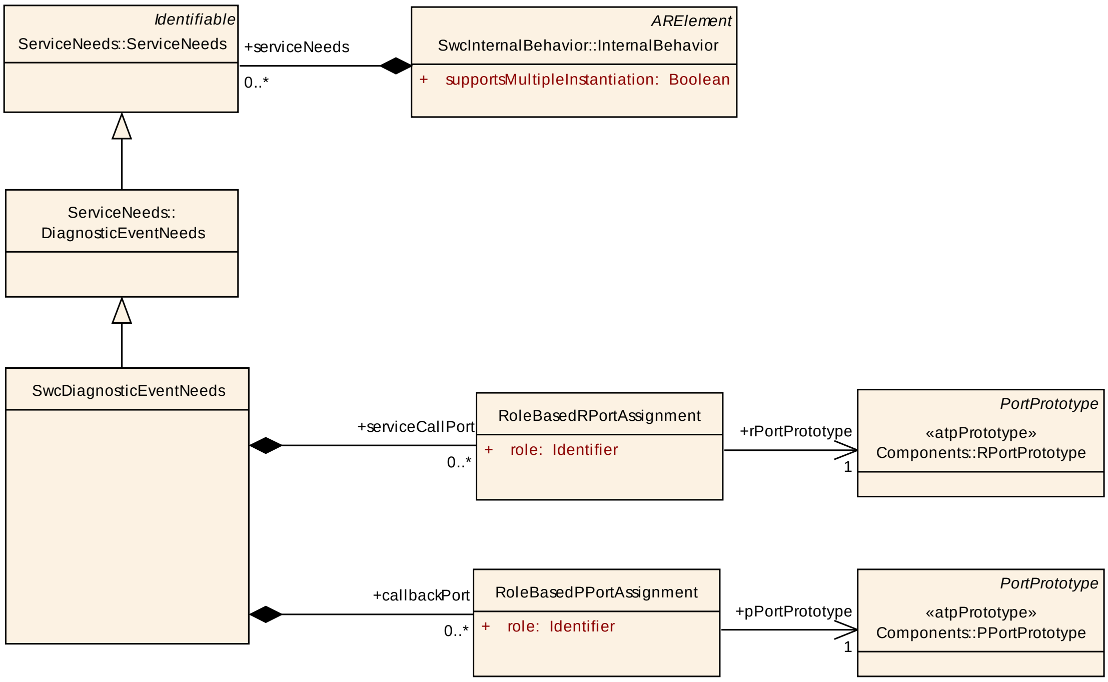


Figure 5.19: `SwcDiagnosticEventNeeds`

diagnostic event (for the terms related to the AUTOSAR Function Inhibition Manager see [22]).

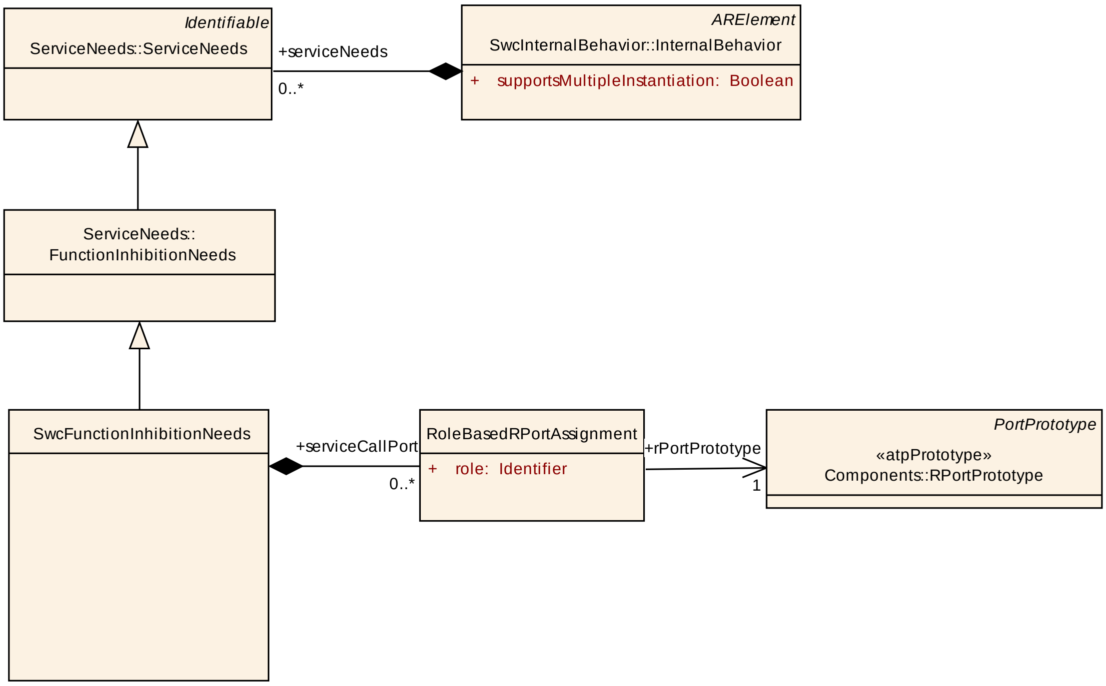


Figure 5.20: `SwcFunctionInhibitionNeeds`

Class	⟨⟨**atpObject**⟩⟩ **FunctionInhibitionNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of the Function Inhibition Manager for one Function Identifier (FID). This class currently contains no attributes. Its name can be regarded as a symbol identifying the FID from the viewpoint of the component or module which owns this class.			
Base Class(es)	ServiceNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 5.31: FunctionInhibitionNeeds

Class	⟨⟨**atpObject**⟩⟩ **SwcFunctionInhibitionNeeds**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::ServiceMapping			
Class Desc.	Specialization of the FunctionInhibitionNeeds for the case it is owned by a SoftwareComponentType. Note that the Function Inhibit Manger does not provide callbacks to services provided by software components. Therefoer there is no property ”callbackPort”.			
Base Class(es)	FunctionInhibitionNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
serviceCallPort	RoleBasedRPortAssignment	*	aggregation	This is the expected service to be called by the software component to handle a particular inhibition of a particular function. This inhibition is controlled by the FunctionInhibitManager. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification ot Function Inhibition Manager”. e-g- ”FunctionInhibition”.

Table 5.32: SwcFunctionInhibitionNeeds

5.7.8 Service Needs for the DCM Service

Figure 5.21 and the following class table show the meta-classes `DiagnosticCommunicationNeed` and `SwsDiagnosticCommunicationNeed` which are used to define special associations needed to configure the Diagnostic Communication Manager Service. An `AtomicSoftwareComponentType` may provide several `DiagnosticCommunicationNeed` elements, each defines all the mappings for one diagnostic communication (for the terms related to the AUTOSAR Diagnostic Communication Manager see [23]).

Class	⟨⟨**atpObject**⟩⟩ **DiagnosticCommunicationNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of the Diagnostic Communication Manager for one ”user”. Details are an expert task for AUTOSAR Release 4.0.			
Base Class(es)	ServiceNeeds			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**

Table 5.33: DiagnosticCommunicationNeeds

Class	⟨⟨**atpObject**⟩⟩ **SwcDiagnosticCommunicationNeeds**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Service Mapping			
Class Desc.	Specialization of the DiagnosticCommunicationNeeds for the case it is owned by a SoftwareComponentType.			
Base Class(es)	DiagnosticCommunicationNeeds			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
callback Port	RoleBased PPortAssignment	*	aggregation	This is the provided service to be called by the Diagnostic Communication Manager to handle a particular Diagnostic Communication.. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification ot Diagnostic Communication Manager” (e.g. something like ”CallBakReqTreatment”).
serviceCall Port	RoleBased RPortAssignment	1	aggregation	This is the expected service to be called by the software component to handle a particular Diagnostic Communkication. The value of the role attribute in the aggregated class must be a name of a PortInterface as standardized in ”Specification ot Diagnostic Communication Manager” (e.g. something like ”DcmService”)

Table 5.34: SwcDiagnosticCommunicationNeeds

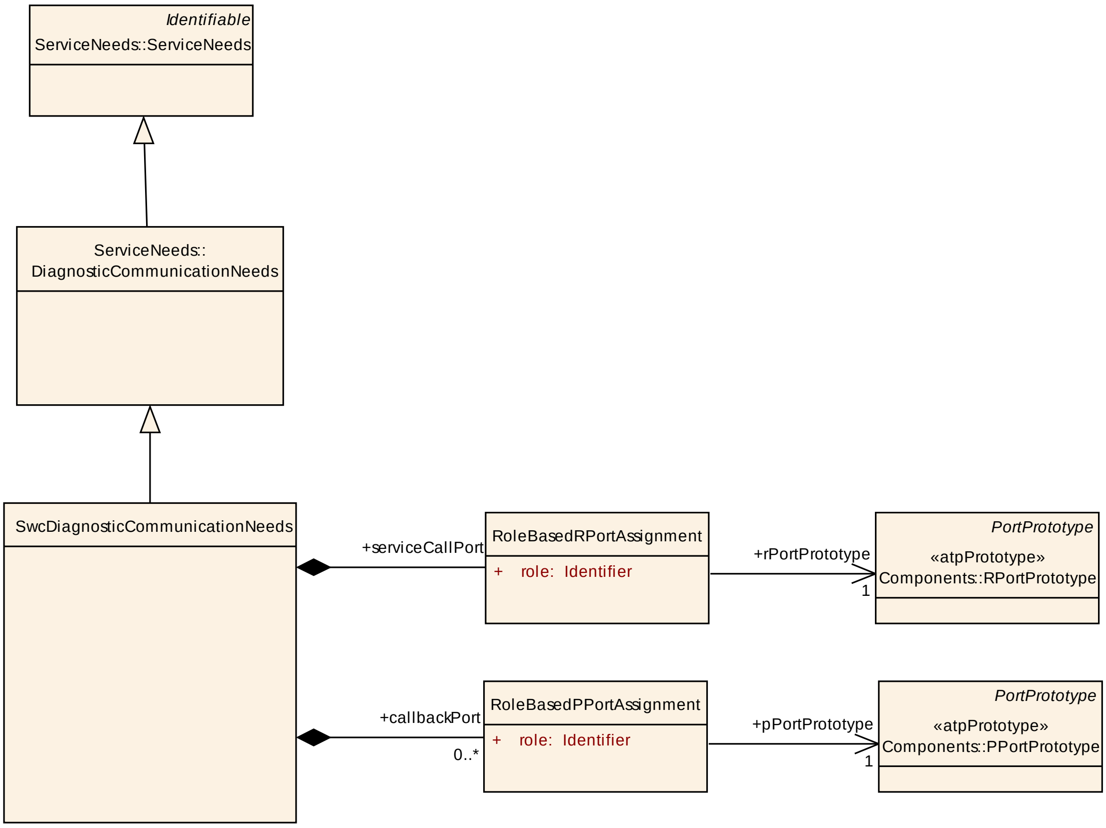


Figure 5.21: `SwcDiagnosticCommunicationNeed`

6 Implementation

Previous versions of this document contained a comprehensive description of the meta-class `Implementation`. This meta-class still exists but the description of most of its content has been moved to another document, in particular the specification of the `Basic Software Module Description Template` [8].

Please note that the `Software Component Template` and the `Basic Software Module Description Template` share the content of `Implementation`. However, the semantics of `Implementation` is closer to the `Basic Software Module Description Template`.

Nevertheless, there is still content strictly related to the `Software Component Template`. This part of `Implementation` consisting of `SwcImplementation` (see Figure 6.1) remains in this document.

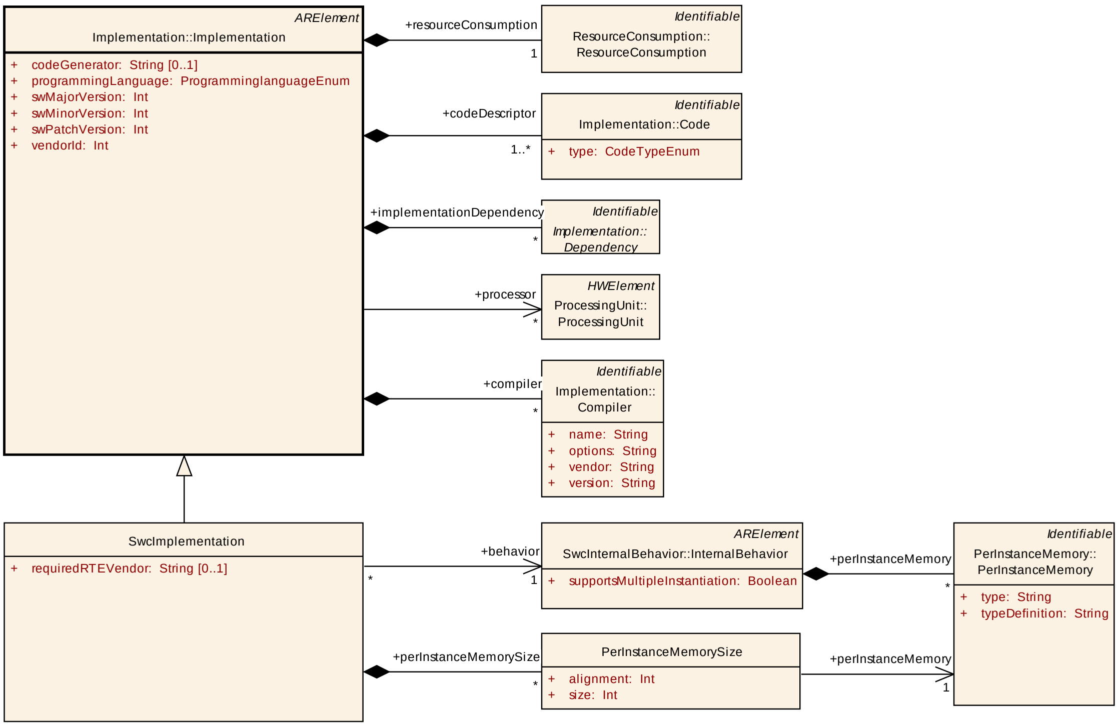


Figure 6.1: Implementation part specific to the Software Component Template

Class	**⟨⟨atpObject⟩⟩ SwcImplementation**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcImplementation			
Class Desc.				
Base Class(es)	Implementation			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
behavior	Internal Behavior	1	reference	The internal behavior implemented by this Implementation.

perInstanceMemorySize	PerInstanceMemorySize	*	aggregation	Allows a definition of the size of the per-instance memory for this implementation.
requiredRTEVendor	String	0..1	aggregation	Identify a specific RTE vendor. This information is potentially important at the time of integrating (in particular: linking) the application code with the RTE. The semantics is that (if the association exists) the corresponding code has been created to fit to the vendor-mode RTE provided by this specific vendor. Attempting to integrate the code with another RTE generated in vendor mode is in general not possible.

Table 6.1: SwcImplementation

Class	⟨⟨**atpObject**⟩⟩ **PerInstanceMemorySize**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcImplementation			
Class Desc.	Resources needed by the allocation of PerInstanceMemory for each SWC instance. Note that these resources are not covered by an ObjectFileSection, because they are supposed to be allocated by the RTE.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
alignment	Integer	1	aggregation	Required alignment (1,2,4,...) of the referenced PerInstanceMemory
perInstanceMemory	PerInstanceMemory	1	reference	
size	Integer	1	aggregation	Size (in bytes) of the reference perInstanceMemory

Table 6.2: PerInstanceMemorySize

7 Mode Management

In general the Software Component Template doesn't define the kind of modes, which must be supported by State Managers or software-components explicitly. However the Software Component template provides generic mechanisms for describing modes. In this section the general relationship between modes, interfaces and software-components is discussed.

The assumption from the software-component point of view is that State Managers are using a Standardized AUTOSAR Interface [1] to influence the software-component and also provide an interface to get requests and confirmations from the software-component. They will be implemented as AUTOSAR services and be part of the Basic Software on each ECU. The actual modes a State Manager provides will have to be standardized as well to allow compatibility between software-components.

7.1 Declaration of Modes

The SW-Component Template provides some simple means to define collections of modes. The name of the mode is the most important attribute that has to be provided for each `ModeDeclaration`. The `ModeDeclarations` are grouped together within the `ModeDeclarationGroup`. The `initialMode` is active before any mode switches occurred. This is shown in Figure 7.1

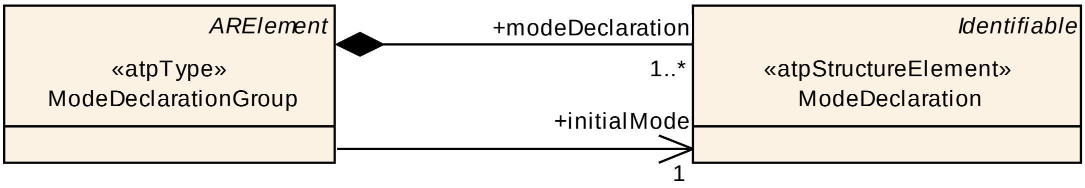


Figure 7.1: `ModeDeclaration`

The class `ModeDeclarationGroup` has been introduced to support the grouping of modes and (on M1 level) to provide predefined sets of modes that could be standardized and re-used. The set of modes eventually defines a flat (i.e. no hierarchical states) state-machine where only one mode can be active at a given point in time.

Please note that the actual definition of modes and their relationship is not in the responsibility of this document. In other words: the definition of modes represents M1 artifacts whereas this document is limited to describing M2 model elements.

Class	⟨⟨**atpStructureElement**⟩⟩ **ModeDeclaration**
Package	M2::AUTOSARTemplates::SWComponentTemplate::ModeDeclaration
Class Desc.	Declaration of one Mode. The name and semantics of a special mode is not defined in the metamodel.

---

[1] See also AUTOSAR Glossary for "Standardized AUTOSAR Interface".

Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 7.1: ModeDeclaration

Class	⟨⟨**atpType**⟩⟩ **ModeDeclarationGroup**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ModeDeclaration			
Class Desc.	A collection of Mode Declarations.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
initialMode	ModeDeclaration	1	reference	The initial mode of the ModeDeclarationGroup. This mode is active before any mode switches occured.
modeDeclaration	ModeDeclaration	1..*	aggregation	The ModeDeclarations collected in this ModeDeclarationGroup.

Table 7.2: ModeDeclarationGroup

7.2 Communication of Modes

The Software-Component Template describes the concept of the communication of `ModeDeclarationGroupPrototypes` similar to the communication of `DataElementPrototypes`: The collections of `ModeDeclarations` that are required or provided by a `ComponentType` are defined through its `SenderReceiverInterfaces` as shown in Figure 7.2.

This allows for explicitly defining `ConnectorPrototypes` which communicate between `ComponentPrototypes` and to define service interfaces for communication with `ServiceComponentPrototypes`. Due to the compatibility rules of `PortInterfaces` (see chapter 3.4) each `ComponentType` can rely on the availability of required mode activations.

Eventually, the abstract definition of the mode management concept refers to the ECU state management [2], i.e. an AUTOSAR service. Consequently, the communication of modes by means of `ModeDeclarationGroupPrototypes` is - like other services - not allowed to go beyond the scope of a particular ECU.

This is because the AUTOSAR concept does not foresee *any* means to map `ModeDeclarationGroupPrototypes` to bus elements (for more details please refer to the specification of the System Template [10]). It is therefore by concept *not possible* to communicate mode changes over a communication bus.

Furthermore, `ConnectorPrototypes` for communicating modes can only be created at the time of ECU configuration (see chapter 10 for more details).

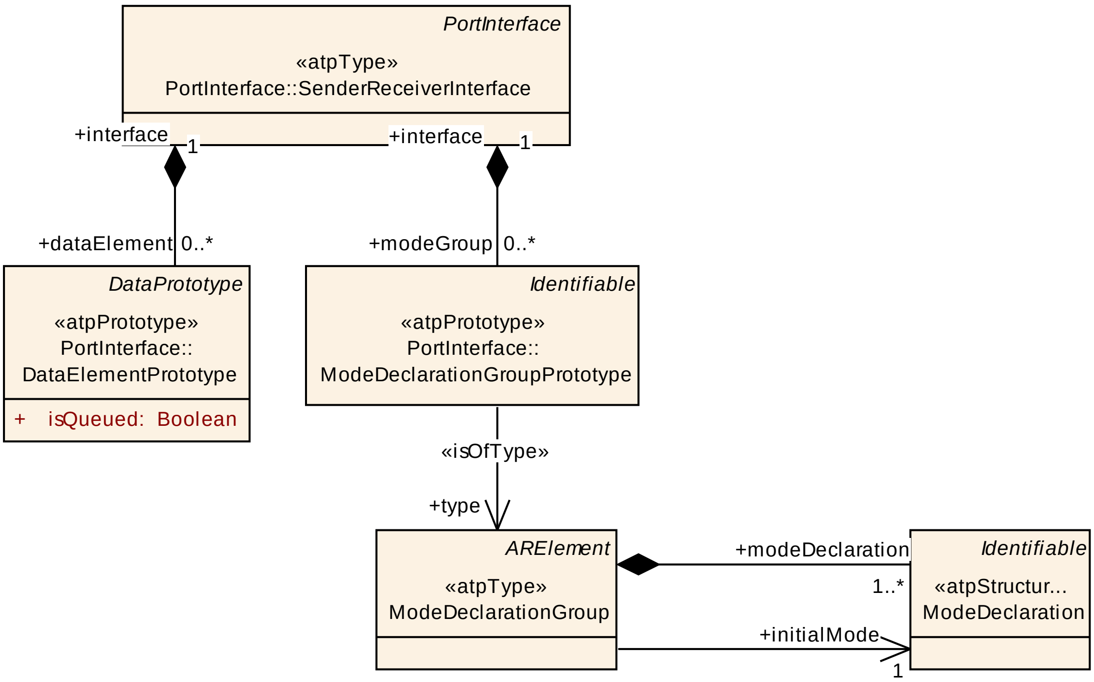


Figure 7.2: Communication of modes

Please note, that each `ComponentType` - `AtomicSoftwareComponentType` as well as `CompositionType` - can provide (via their `PortPrototypes` and `SenderReceiverInterfaces`) a list of required and provided `ModeDeclarationGroupPrototypes`.

Eventually, a `CompositionType` requires and provides the modes that are required or provided by its contained `ComponentPrototypes`. The delegation of these modes from `ComponentPrototypes` to the enclosing `CompositionType` is explicitly described by `DelegationConnectorPrototypes`.

The Software-Component description does not make any assumptions about the semantics of the required and provided `ModeDeclarationGroupPrototypes`. It just requires and provides the `ModeDeclarationGroupPrototypes` by name.

7.3 Modes and Events

Software-components need to be capable of reacting to state changes issued by some Mode Manager and adopt their behavior to the new situation. Such a mode dependent software-component is shown in Figure 7.3.

Since the behavior of `AtomicSoftwareComponentTypes` is mainly determined by the `RunnableEntities` contained in the `InternalBehavior` it is necessary to configure the response to mode changes on the level of `RunnableEntities`.

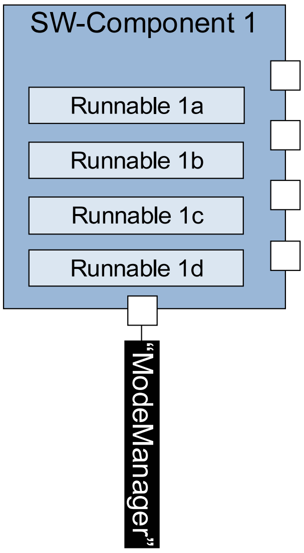


Figure 7.3: State Managers and software-components

Figure 7.4 shows an excerpt of the meta-model illustrating how the relationship between the current mode and the `InternalBehavior` of the `AtomicSoftwareComponentType` can be described.

The `AtomicSoftwareComponentType` can use two mechanisms to define how its `InternalBehavior` should interact with the mode management.

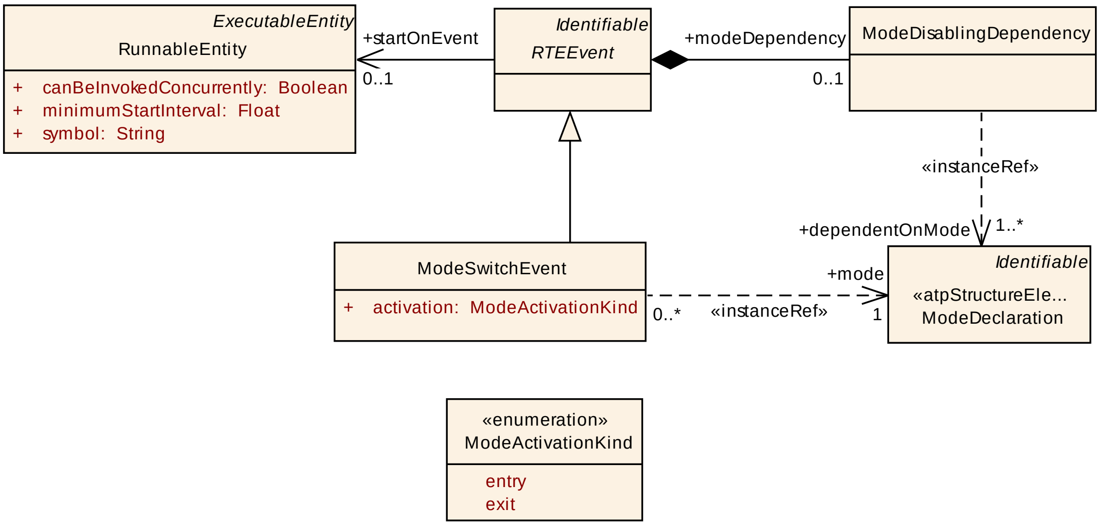


Figure 7.4: Modes and events

Using the first mechanism (`ModeSwitchEvent`, see figure 7.5), an `AtomicSoftwareComponentType` can define an `RTEEvent` to specify that a specific `RunnableEntity` must be started whenever a mode is entered and/or exited.

Using the second mechanism (`ModeDisablingDependency`), the `AtomicSoftwareComponentType` can indicate whether an `RTEEvent` that starts an associated `RunnableEntity` is mode-dependent. `RTEEvents` without a `modeDependency` occur regularly according to their definition. `RTEEvents` with the optional `modeDependency` have the additional limitation that the associated `RunnableEntity` is *not* started when the `ModeDeclaration` referenced by the `ModeDisablingDependency` is active.

Class	⟨⟨**atpObject**⟩⟩ **ModeDisablingDependency**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ModeDeclaration			
Class Desc.	Collection of references to the Modes that disable the RTEEvent			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
dependentOnMode	ModeDeclaration	1..*	instanceRef	Reference to the Modes that disable the Runnable Entity.

Table 7.3: ModeDisablingDependency

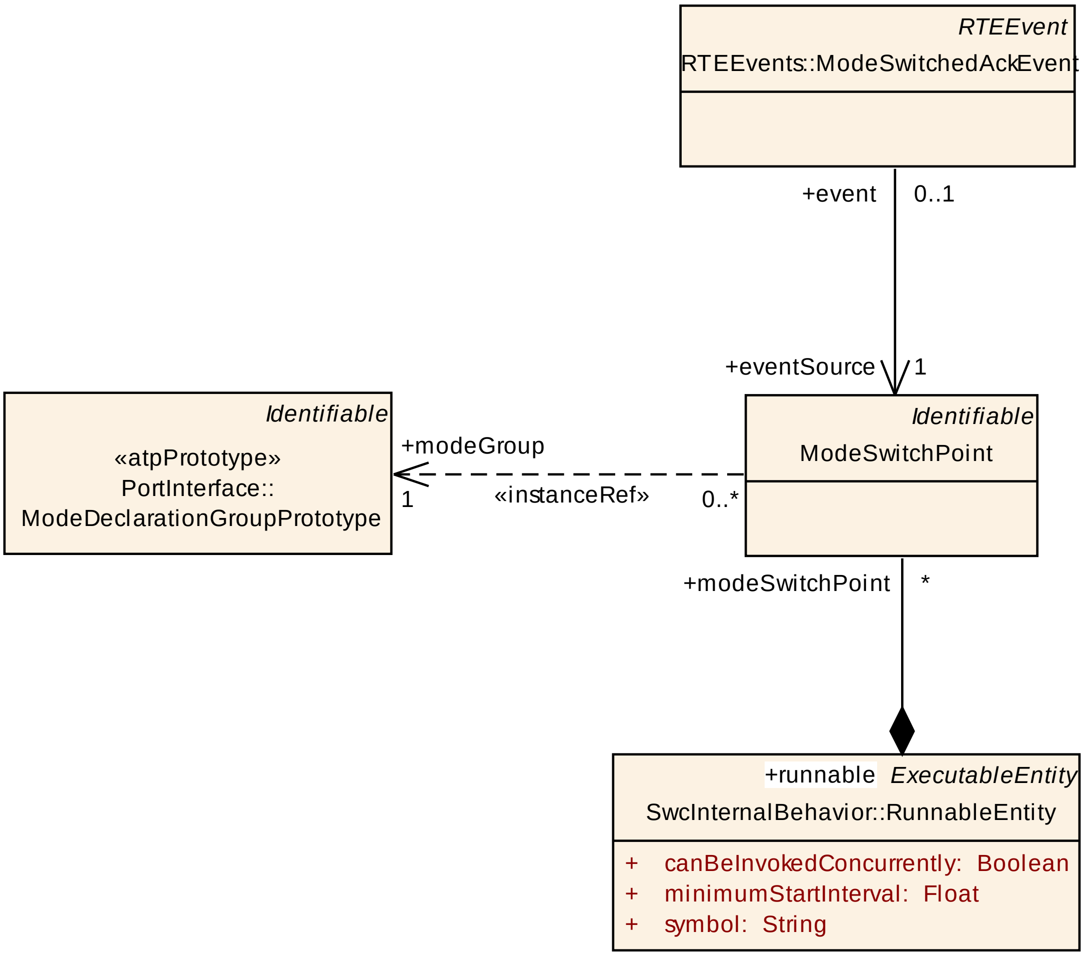


Figure 7.5: `ModeSwitchEvent`

A `RunnableEntity` can also have `ModeSwitchPoints` that eventually associates a `RunnableEntity` with a specific `ModeDeclarationGroup`.

Class	⟨⟨**atpObject**⟩⟩ **ModeSwitchPoint**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::Mode DeclarationGroup			
Class Desc.	A ModeSwitchPoint is required by a RunnableEntity owned a Mode Manager. Its semantics implies the ability to initiate a mode switch.			
Base Class(es)	Identifiable			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
mode Group	ModeDeclaration Group Prototype	1	instanceRef	

Table 7.4: ModeSwitchPoint

The `ModeSwitchPoint` also allows for the definition of a `ModeSwitchedAckEvent`. This `RTEEvent` is eventually owned by a mode manager to allow for getting confirmation of a mode change.

Class	⟨⟨**atpObject**⟩⟩ **ModeSwitchedAckRequest**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Desc.	Requests acknowledgements that a mode switch has been proceeded successfully			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
timeout	Float	1	aggregation	Number of seconds before an error is reported or in case of allowed redundancy, the value is sent again.

Table 7.5: ModeSwitchedAckRequest

Class	⟨⟨**atpObject**⟩⟩ **ModeSwitchedAckEvent**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::RTEEvents			
Class Desc.	The event is raised when the referenced mode have been received or an error occurs.			
Base Class(es)	RTEEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
eventSource	ModeSwitchPoint	1	reference	Mode switch point that triggers the event.

Table 7.6: ModeSwitchedAckEvent

7.4 Initialization / Finalization

The AUTOSAR standard must support the execution of initialization code for every `AtomicSoftwareComponentType`. Most `AtomicSoftwareComponentTypes` will need to initialize by executing specific code; this code must complete before any other code in the component is executed. Data will be initializing to specific values before the ”normal” application software is running.

The AUTOSAR standard must also support the execution of finalization code for every `AtomicSoftwareComponentType`. Most `AtomicSoftwareComponentTypes` will need to finalize by calling specific code; this code must complete before the functionality of the application software shut down (e.g. a motor drive in a start or end position).

With the mechanisms provided by the mode manager and the activation of `RunnableEntities` driven by `ModeSwitchEvents` it is easily possible to define a mode ”Initialization”. When ”Entering” this state initialization `RunnableEntities` can be activated. When all initialization `RunnableEntities` have finished the mode manager can change to further modes.

Also the equivalent can be realized for the finalization of `AtomicSoftwareComponentTypes`.

Please note: The initial modes of `AtomicSoftwareComponentTypes` are defined by the initial mode references of the required mode groups. These modes are activated before any other mode activation has occurred. It is the responsibility of the RTE to activate all initial modes on a certain ECU.

7.5 Summary Meta-Model Excerpt Related to Modes

Figure 7.6 provides an overview of all meta-model elements that have a semantical relationship to the mode-management aspect.

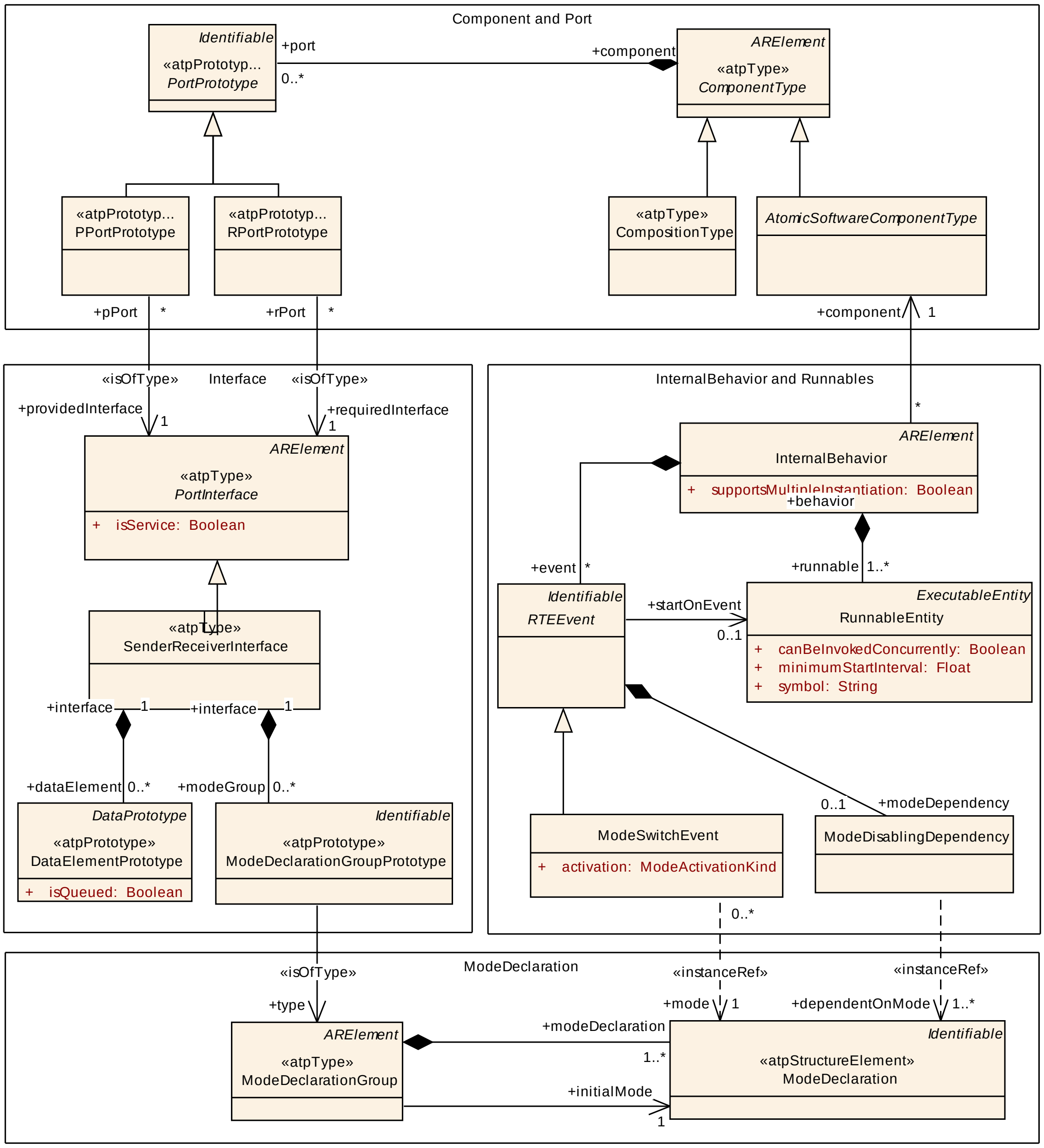


Figure 7.6: Summary meta-model excerpt related to modes

8 Measurement and Calibration

This section describes how software components have to be prepared for measurement & calibration. It is the goal to merge the AUTOSAR ideas with practice currently supported by ASAM definitions such as A2L, MDX, CDF.

Please note: Calibration and Measurement support is taken over from the approaches of ASAM, and in particular MDX which is based on MSRSW. This takeover was done by reverse engineering the MSRSW to UML and importing the relevant classes. Also note that some of the documentation provided here is taken from MSR and might even reflect some differences between the MSR approach and AUTOSAR which will be harmonized in future versions.

8.1 Basic Approach

While performing the calibration process using a MCD tool (Measurement, Calibration and Diagnostic), the calibration engineer needs to have a specific insight to the data within the CPU at runtime. This insight is provided by access to ECU internal variables (also called measurements) as well as calibration parameters (sometimes also called characteristic value).

A calibration parameter is a parameter which characterizes the dynamics of a control algorithm. From a software implementation point of view, it is a variable with only read-access during normal operation of an ECU. Similar to `DataPrototypes` Calibration Parameters can be defined for an `InternalBehavior` of a `ComponentType` (this relates to `InterRunnableVariables`), individually for a `ComponentPrototype` (similar to `PerInstanceMemory`) as well as for several `SoftwareComponentPrototypes` (using the port-/interface-concept).

Therefore, the description of variables and calibration parameters are basically the same. In AUTOSAR both appear finally as `(`DataPrototypes).

8.2 Properties of Data Definitions

Measurement and calibration entities are based on the concept of data definitions. The properties of these data definitions are reflected by a dedicated meta-model element, the so-called `SwDataDefProps`, which covers all properties of a particular data element under various aspects, e.g. how a `DataPrototype` can be measured or a parameter can be calibrated.

The aspects covered by the `SwDataDefProps` are

- Structure of the data element, is it a single value, a curve, or a map, but also the recordLayouts which specify, how such elements are mapped/converted to the DataTypes in AUTOSAR. This is mainly expressed by properties like `swRecordLayout` and `swCalprmAxisSet`

- Implementation policy, mainly expressed by `swImplPolicy`, `swVariableAccessImplPolicy`, `swAddrMethod`
- Access policy for the MDC system, mainly expressed by `swCalibrationAccess`
- Semantics of the data element, mainly expressed by `compuMethod` and/or `unit`, `dataConstr`
- Code generation policy provided by `swCodeSyntax`

In AUTOSAR, `SwDataDefProps` can be attached on primitive type level as well as on prototype level. In general, properties specified on prototype level override the ones specified on type level. Obviously such an override is not applicable in all cases. In particular, the properties covering the Structure must not be redefined on `DataPrototype`. Implementation policy, semantics and code generation policy may be changed under consideration of compatibility rules. Access policy for the MCD system is the most likely subject to be redefined on the DataPrototype.

In AUTOSAR `SwDataDefProps` are attached to derivations of `DataPrototypes`, namely

- `DataElementPrototypes` and `ArgumentPrototypes` in their respective context of `PortPrototypes` and `ComponentPrototypes`.
- `InterRunnableVariable` and
- `CalprmElementPrototype`

to set the `swCalibrationAccess` to READ respectively READ-WRITE in the first two cases or to define the properties of Calibration Parameters in case three.

Section 8.3 describes how `SwDataDefProps` are attached to `DataPrototypes` for measuring purposes while Section 8.4 and 8.5 describe the construction of charactistics based on the combination of `SwDataDefProps` with `DataPrototypes`. Section 8.4 describes in which context characteristics can be defined. Finally, sections 8.6, 8.7, and 8.8 show how characteristics are used in `RunnableEntities` and show the link to an actual ECU implementation.

The way the `SwDataDefProps` are attached to a `DataPrototype` depends on the purpose of the `DataPrototype` and is described in detail in the following sections.

Enumeration	**SwCalibrationAccessEnum**
Package	M2::AUTOSARTemplates::CommonStructure::DataDefProperties
Enum Desc.	Determines the access rights to a data object w.r.t. measurement and calibration.
Literal	***Description***
readOnly	The element will only appear as read-only in an ASAP file.
notAccessible	The element will not be accessible via MCD tools, i.e. will not appear in the ASAP file.
readWrite	The element will appear in the ASAP file with both read and write access.

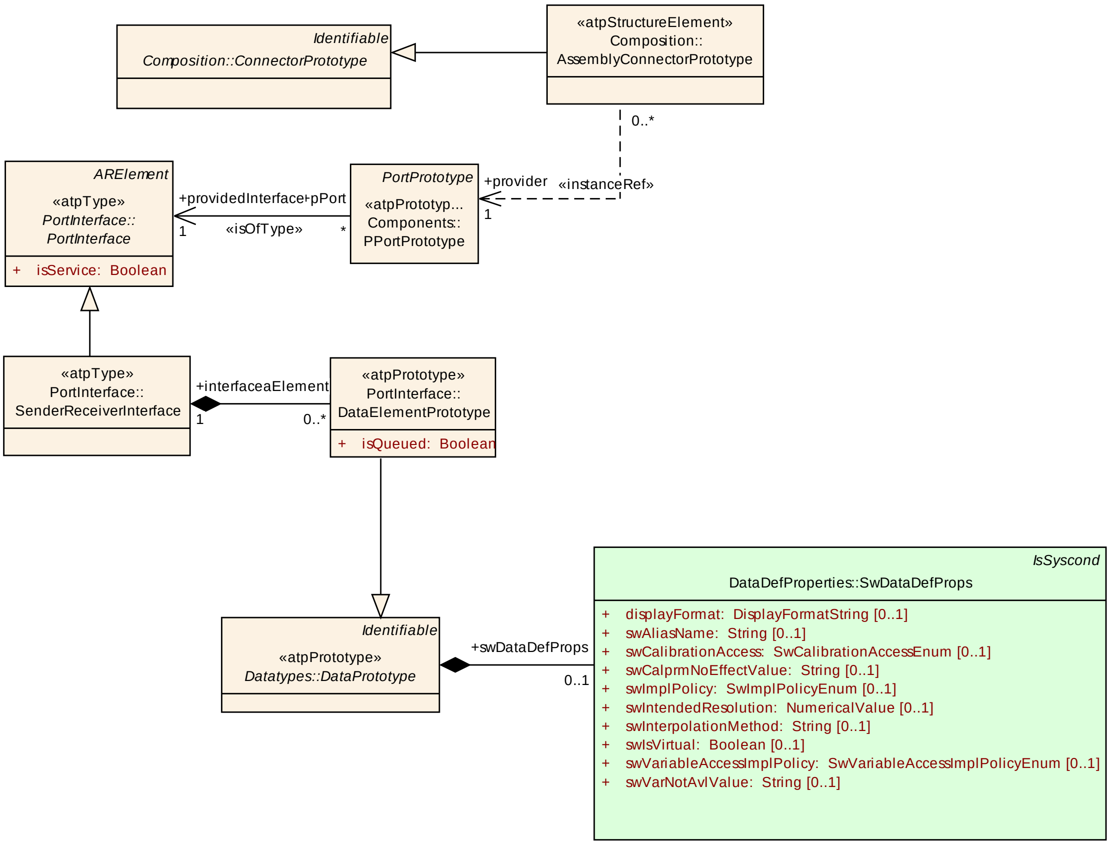


Figure 8.1: Data-Def-Properties in Connector Context

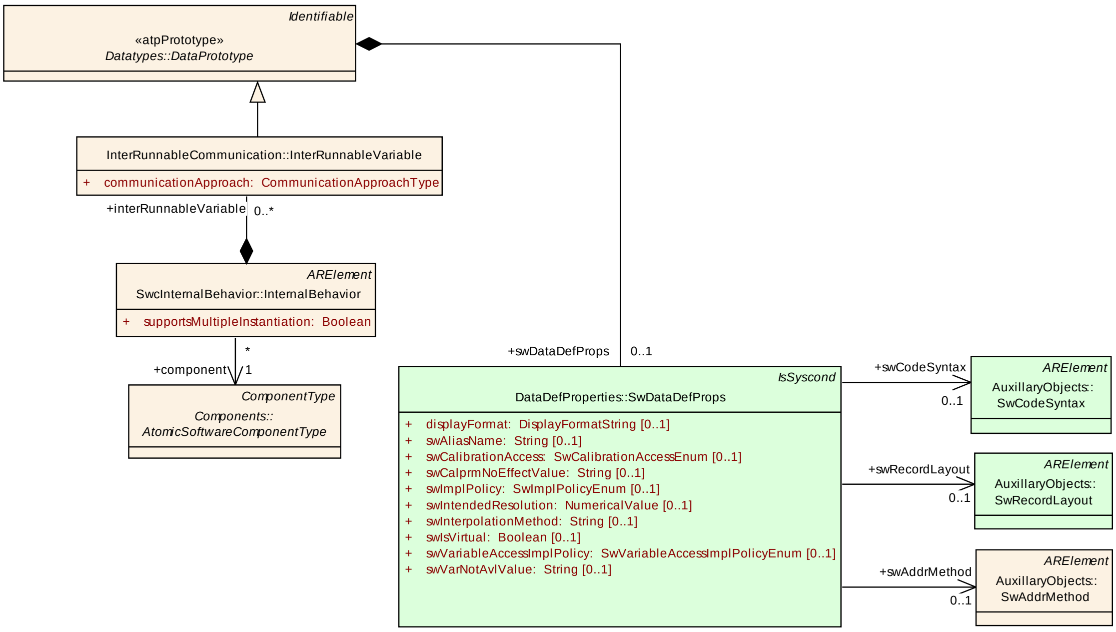


Figure 8.2: Data-Def-Props in Inter-Runnable-Variable Context

Class	⟨⟨**atpObject**⟩⟩ **SwDataDefProps**
Package	M2::AUTOSARTemplates::CommonStructure::DataDefProperties
Class Desc.	This class is a collection of properties relevant for data objects under various aspects. One could consider this class as a ”pattern of inheritance by aggregation”. The properties can be applied to all objects of all classes in which SwDataDefProps is agrregated. Note that not all of the attributes or associated elements are useful all of the time. Hence, the process definition (e.g. expressed with an OCL or a Document Control Instance) MSR-DCI has the task of implementing limitations. SwDataDefProps covers various aspects: * Structure of the data element, is it a single value, a curve, or a map, but also the recordLayouts which specify, how such elements are mapped/converted to the DataTypes in the programming language (or in Autosar). This is mainly expressed by properties like swRecordLayout and swCalprmAxisSet * Implementation policy, mainly expressed by swImplPolicy, swVariableAccessImplPolicy, swAddrMethod * Access policy for the MDC system, mainly expressed by swCalibrationAccess * Semantics of the data element, mainly expressed by compuMethod and/or unit, dataConstr * Code generation policy provided by swCodeSyntax
Base Class(es)	ARObject

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
annotation	Annotation	*	aggregation	This aggregation allows to add annotations (yellow pads ...) related to the current data object.
baseType	SwBaseType	0..1	reference	Base type associated with the value axis of this data object.
compuMethod	CompuMethod	0..1	reference	Computation method associated with the semantics of this data object.
dataConstr	DataConstr	0..1	reference	Data constraint for this data object.
displayFormat	DisplayFormatString	0..1	aggregation	This property describes how a number is to be rendered e.g. in documents or in a measurement and calibration system.
invalidValue	PrimitiveSpecification	0..1	aggregation	Optional value to express invalidity of the actual data element. If given, the owning component has the API to set this data element invalid, otherwise it does not.
swAddrMethod	SwAddrMethod	0..1	reference	Addressing method related to this data object.
swBitRepresentation	SwBitRepresentation	0..1	aggregation	Description of the binary representaion in case of a bit variable.

swCalibrationAccess	SwCalibrationAccess Enum	0..1	aggregation	Specifies the read or write access by MCD tools for this data object.
swCalprm AxisSet	SwCalprm AxisSet	0..1	aggregation	This specifies the properties of the axes in case of a curve or map etc. This is mainly applicable to calibration parameters.
swCode Syntax	SwCode Syntax	0..1	reference	Coding policy for this data object expressed as a reference to a Code syntax to be applied.
swDataDependency	SwData Dependency	0..1	aggregation	If the data object is virtual - that means it is not directly in the ecu, then this property describes how the ”virtual variable” can be computed from the real ones.
swHost Variable	SwVariable Ref	0..1	aggregation	Contains a reference to a variable, which serves as a host-variable for a bit variable. Only applicable to bit objects.
swImpl Policy	SwImpl Policy Enum	0..1	aggregation	Implementation policy for this data object.
swPointer	SwPointer	0..1	aggregation	Specifies that the containing data object is a pointer to another data object.
swRecord Layout	SwRecord Layout	0..1	reference	Record layout for this data object.
swText Props	SwText Props	0..1	aggregation	the specific properties if the data object is a text object.
swValue BlockSize	SwArraysize	0..1	aggregation	Specifies the size in case the data object is an VAL_BLK. It is there for compatibility reasons, where value blocks were introduced as a kind of an array.
swVariable Access ImplPolicy	SwVariable Access ImplPolicy Enum	0..1	aggregation	In case of a swImplPolicy set to ”message” the access policy can be refined here.
unit	Unit	0..1	reference	Physical unit associated with the semantics of this data object. This attribute applies, if no compuMethod is specified. If buth units (this as well as via compuMethod is specified,the units ust be the same.

Table 8.1: SwDataDefProps

8.3 Measurement

In embedded automotive software design, measurement means access to memory locations in an ECU and transferring its contents to the measurement & calibration system. While in classical software design, variables abstract the memory locations in the code, AUTOSAR provides for this purpose the DataPrototype, which is used in the context of several other prototypes. The following `DataPrototypes` corresponds to SW-VARIABLE in ASAM-MDX.

- `DataElementPrototype` of a `SenderReceiverInterface` used in a `PortPrototype` (of a `ComponentPrototype`), to capture sender-receiver communication between `ComponentPrototypes`, and `ArgumentPrototype` of an `OperationPrototype` in a `ClientServerInterface` to capture client-server communication between `ComponentPrototypes`, and
- `InterRunnableVariable` to capture communication between `RunnableEntities` within a `ComponentPrototype`.

Various categories ”variables”of the can be distinguished by the `category` in `Identifiable`

ASAM Category	purpose	Specific dataDefProps
VALUE	One single value	
VALUE_ARRAY	An array of values	Must refer to an `ArrayType`. Category in `ArrayElement` must be ”VALUE”. `DataDefProps` within `ArrayElement` must be specified.
ASCII	A String	swTextProps / swMaxTextSize
BOOLEAN	A Boolean value	
STRUCTURE	A Structure of Values	Must refer to a `RecordType`. Category within `RecordElement` must be ”VALUE”. `DataDefProps` within `RecordElement` must be specified.
STRUCTURE_ARRAY	An array of Structure of Values	Must refer to an `ArrayType` of which `ArrayElement` must refer to a `RecordType`. Category in `ArrayElement` must be STRUCTURE. `DataDefProps` within `RecordElement` must be specified. Category within `RecordElement` must be VALUE.

Note that the type of the `DataPrototype` must match the purpose denoted by the category value. For example if the measurement/category denotes a STRUCTURE, the data type must be a composite data type. The following structural features from `SwDataDefProps` apply:

Property	Explanation
`compuMethodRef`	Indicates the computation method of the particular measurement. Note that in case the `DataElementPrototype` is of type `PrimitiveType` referring to a `compuMethod`, both must refer to the same `compuMethod`. If it is missing the `CompuMethod` is either specified by the `PrimitiveType`, or it is the IDENTITY compu method.
`baseTypeRef`	Indicates the basic type how the object (measurement or calibration parameter) is handled within the ECU.
`swAddrMethodRef`	Indicates the method, how the object (measurement or calibration parameter) is addressed within the CPU such that a calibration system can handle it properly.
`swCalibrationAccess`	Indicates the modes how a calibration system can access the measurement
`dataConstrRef`	Refers to the data constraints allowing the calibration system to validate measurements and user input.
`swImplPolicy`	Indicates, how the access to the measurement is implemented.
`unitRef`	The physical unit if not specified by the `compuMethod`

Enumeration	**SwImplPolicyEnum**
Package	M2::AUTOSARTemplates::CommonStructure::DataDefProperties
Enum Desc.	Specifies the implementation strategy with respect to consistency mechanisms of variables.
Literal	***Description***
measurement Point	The data element is never read directly within the ECU software. It is written for measurement purposes only.
standard	No specific protection measures are taken. Usually applies to variables inside of an excutable entity.
message	The access to the measurement must be implemented using protection mechanisms. This mainly applies to variables shared by executable entities, i.e. InterRunnableVariables.

The ability of such a `Measurement` to be accessed by, e.g. a calibration tool, is given by setting the `swCalibrationAccess` attribute. The following table shows all valid settings of `swCalibrationAccess`:

Enumeration	**SwCalibrationAccessEnum**
Package	M2::AUTOSARTemplates::CommonStructure::DataDefProperties
Enum Desc.	Determines the access rights to a data object w.r.t. measurement and calibration.
Literal	***Description***
readOnly	The element will only appear as read-only in an ASAP file.
notAccessible	The element will not be accessible via MCD tools, i.e. will not appear in the ASAP file.
readWrite	The element will appear in the ASAP file with both read and write access.

Value of `swCalibrationAccess`	Explanation
NOT-ACCESSIBLE	The element will not appear in an ASAP file A2L.
READ-ONLY	The element will only appear as read-only in an ASAP file
READ-WRITE	Both read and write access.attribute

All properties defined in `SwDataDefProps` at any location must be processed and must be consistent. It is an error if conflicting properties are specified. As an example, a `dataConstraint` may be specified at type as well as at prototype level. In this case the prototype may specify stronger constraints than the type but not vice versa.

To keep it simple for AUTOSAR it is recommended to avoid the multiple definition of the same data definition property. For example `compuMethod` might be defined on type level only, while `baseType` might be defined on prototype level. In other words: the various options to aggregate `SwDataDefProps` provide flexibility where to define particular properties, but not to have properties overriding each other.

The same applies to units which may be defined at `SwDataDefProps` as well as within a `CompuMethod`. Usually units are defined within the `CompuMethod`. But if it is defined within `SwDataDefProps` (for exceptional use cases) it must be compatible to the ones defined in the referred `CompuMethod`.

8.4 Characteristic Values

A Calibration Parameter is a parameter which characterizes the dynamics of a control algorithm. From a software implementation point of view, it is a variable with only read-access during the normal operation of an ECU. Characteristics are specialized `DataPrototype` entities in terms of its associated type but are used in a similar way. This means that Calibration Parameters can be defined for

- `InternalBehavior` of a `ComponentType` (this relates to `InterRunnableVariables`),
- individually for a `ComponentPrototype` (similar to `PerInstanceMemory`) as well as
- for several `SwComponentPrototypes` (using the port-/interface-concept).

A characteristic is represented by the `CalprmElementPrototype` entity. It is derived from `Identifiable`, thus having a `longName` and a `shortName`, a `description` and a `category`. The category determines the type of the characteristic table. The categories (according ASAM - MDX) are shown in table 8.2. The main ones are illustrated in Figure 8.3

ASAM Category	purpose	Specific dataDefProps
VALUE	One single calprm value	

VALUE_ARRAY	Array of calprm values	Must refer to an `ArrayType`. Category in `ArrayElement` must be ”VALUE”. `DataDefProps` within `ArrayElement` must be specified.
VAL_BLK	Value block - a homogeneous fixed sized block of parameters.	`SwValueBlocksize`
CURVE	Curve (Characteristic) `SwCalprmAxisSet` with one `calprmAxis`	
CURVE_ARRAY	array of curves	Must refer to an `ArrayType`. Category in `ArrayElement` must be ”CURVE”. `DataDefProps` within `ArrayElement` must be specified as: `SwCalprmAxisSet` with one `calprmAxis`
MAP	Map	`SwCalprmAxisSet` with two calprmAxis
MAP_ARRAY	array of maps	Must refer to an `ArrayType`. Category in `ArrayElement` must be ”CURVE”. `DataDefProps` within `ArrayElement` must be specified as: `SwCalprmAxisSet` with two `calprmAxis`
COM_AXIS	Common Axis A COM_AXIS (common axis) is an axis definition as separate calibration parameter and can be referenced by any curve or map. The benefits by using a common axis is that it saves memory space, cause it is stored only one time and can be used in multiple curves or maps.	`SwCalprmAxisSet` with one `calprmAxis`

RES_AXIS	Rescale axis A RES_AXIS (rescale axis) is also a shared axis like COM_AXIS, the difference is that this kind of axis can be used for rescaling. Note that the RES_AXIS is by nature a CURVE which is used to implement a non linear scaling (rescale) of the axis. The benefits by using a rescale axis is that it saves memory space, because it is stored only one time and can be used in multiple curves or maps. In addition to this it can compress a huge range to a non linear distributed axis points thus retaining the required accuracy.	`SwCalprmAxisSet` with one `calprmAxis`
ASCII	calprm as text This indicates a parameter in text form (e.g. a message to be displayed to the driver).	`swText` / `swMaxTextSize`
STRUCTURE	A Structure of Values	Must refer to a `RecordType`. Category within `RecordElement` must be set accordingly. `DataDefProps` within `RecordElement` must be specified.

STRUCTURE_ARRAY	An array of Structure of Values	Must refer to an `ArrayType` of which `ArrayElement` must refer to a `RecordType`. Category in `ArrayElement` must be STRUCTURE. `DataDefProps` within `RecordElement` must be specified. Category within `RecordElement` must be set accordingly.

Table 8.2: CalPrm Categories

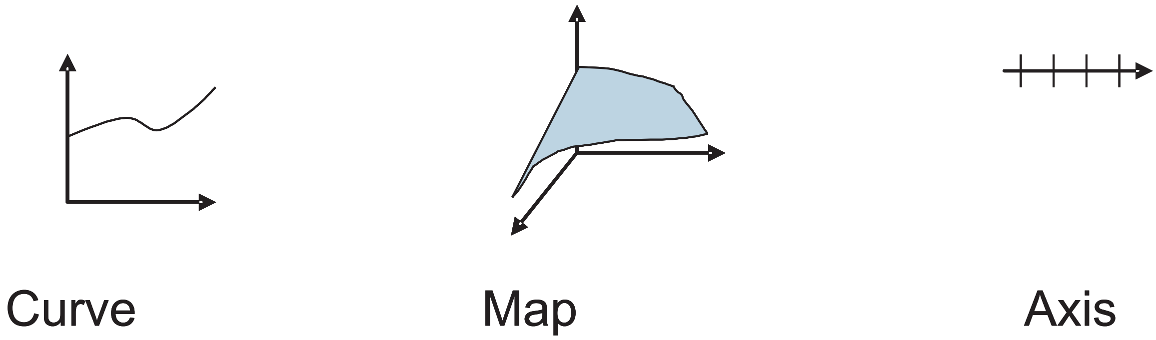


Figure 8.3: Some Categories of Calprms

Section 8.5 shows how to construct particular `CalprmElementPrototypes` based on categories and axis descriptions. Though all `DataPrototype` are derived from `Identifiable` and thus may have its category set to one of the entries above, this particular setting is only allowed in the meta-model-element `CalprmElementPrototype`. Authoring tools have to reflect this constraint.

8.5 Representing CalprmElementPrototypes based on Categories

A characteristic table is defined by setting the category of the `CalprmElementPrototype` to CURVE. Its `SwDataDefProps` determine an axis description. In MSRSW the type of the functional values is given by the attached `BaseType` and the `CompuMethod`.

The axis description is defined by the meta-model element `SwCalprmAxisSet` aggregating a `SwCalprmAxis`. In the latter's aggregated `SwCalprmAxisTypeProps` it is determined whether the axis is a so called ”individual axis” or a ”grouped axis”. The latter which is used to share axis points by several characteristic tables. The diagram below shows how an individual axis is represented by the meta-model element

`SwAxisIndividual`. The `SwAxisIndividual` references value-models to account the minimum and the maximum number of axis values as well as the number of axis points. Hence, the size of the structure to hold the functional values is determined by the number of axis values for all axis's. The type of the axis values is determined when the type of the referenced input value (`swVariableRef`) has been set. For further details see 8.6.4.

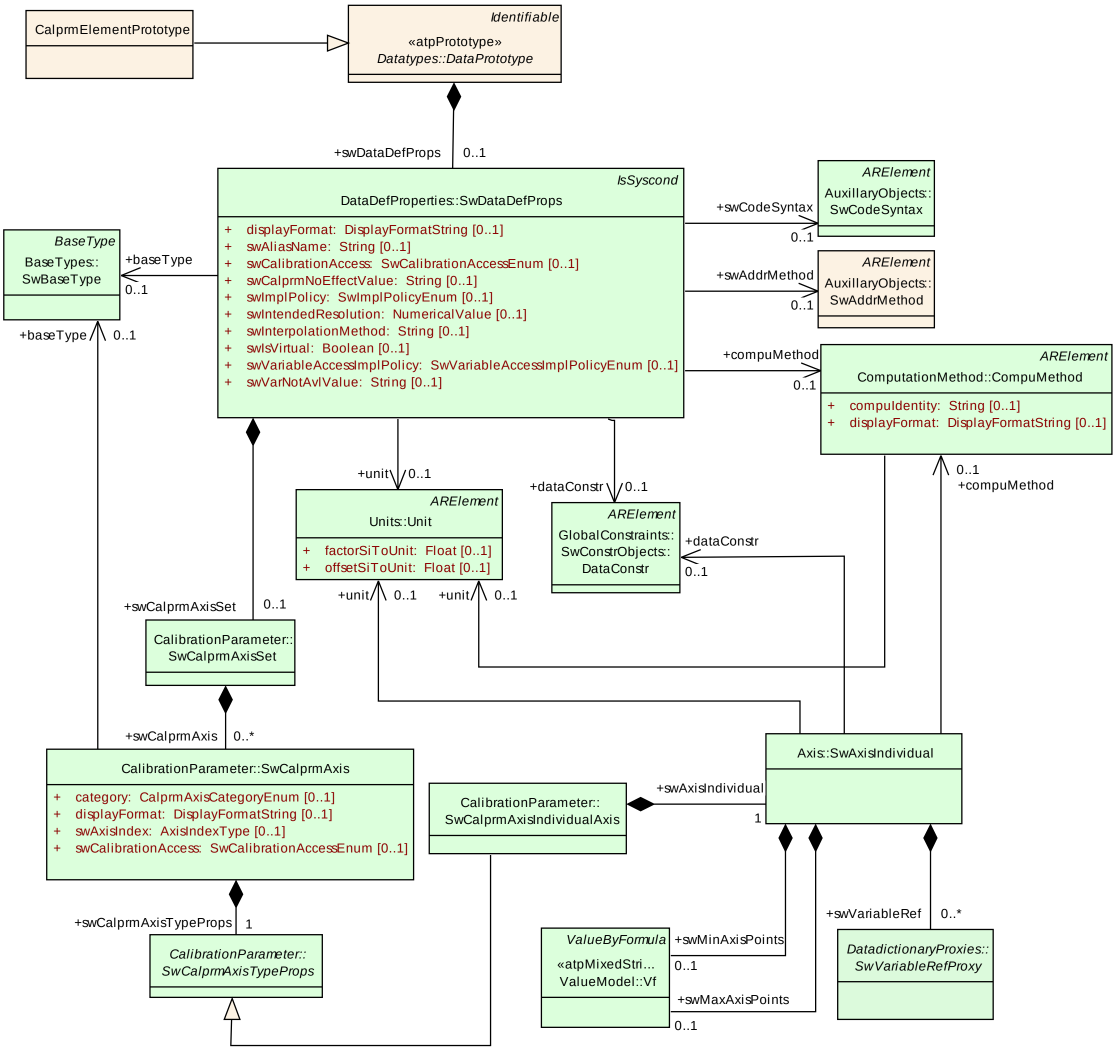


Figure 8.4: Model of a Curve

The actual memory layout of the characteristic in an ECU is determined by the `SwRecordLayout` which is referenced by the `SwDataDefProps` of `CalprmElementPrototype`. There are a tremendous number of record layouts used in automotive industry.

Constructing a record layout by using an AUTOSAR `CompositeDatatype` like record or array would just describe very simple layouts assuming the use of contiguous memory sections, which are rarely used. All employed meta-model entities to describe a curve are shown in 8.4.

In AUTOSAR, the type of DataType of a calibration parameter is given by the `Datatype` of the `CalprmElementPrototype`, which is derived from `DataElementPrototype` which is again derived from `DataPrototype`.

For primitive values, this type must be correlated with the `baseType` specified in the `DataDefProps`. For primitive values, this type correlates to the `DataStructure` in 4.1.

For multidimensional calibration parameters (curves, maps), the datatype from AUTOSAR perspective must be in sync with the more detailed specification provided by the referenced `SwRecordLayout`.

In migration scenarios from MSRSW to AUTOSAR, the baseType of the `Datatype` of the functional values must be consistent with a `baseType` referenced within the `DataPrototype`. This relationship is shown in 8.5 showing that the baseType can be specified on type and on prototype level.

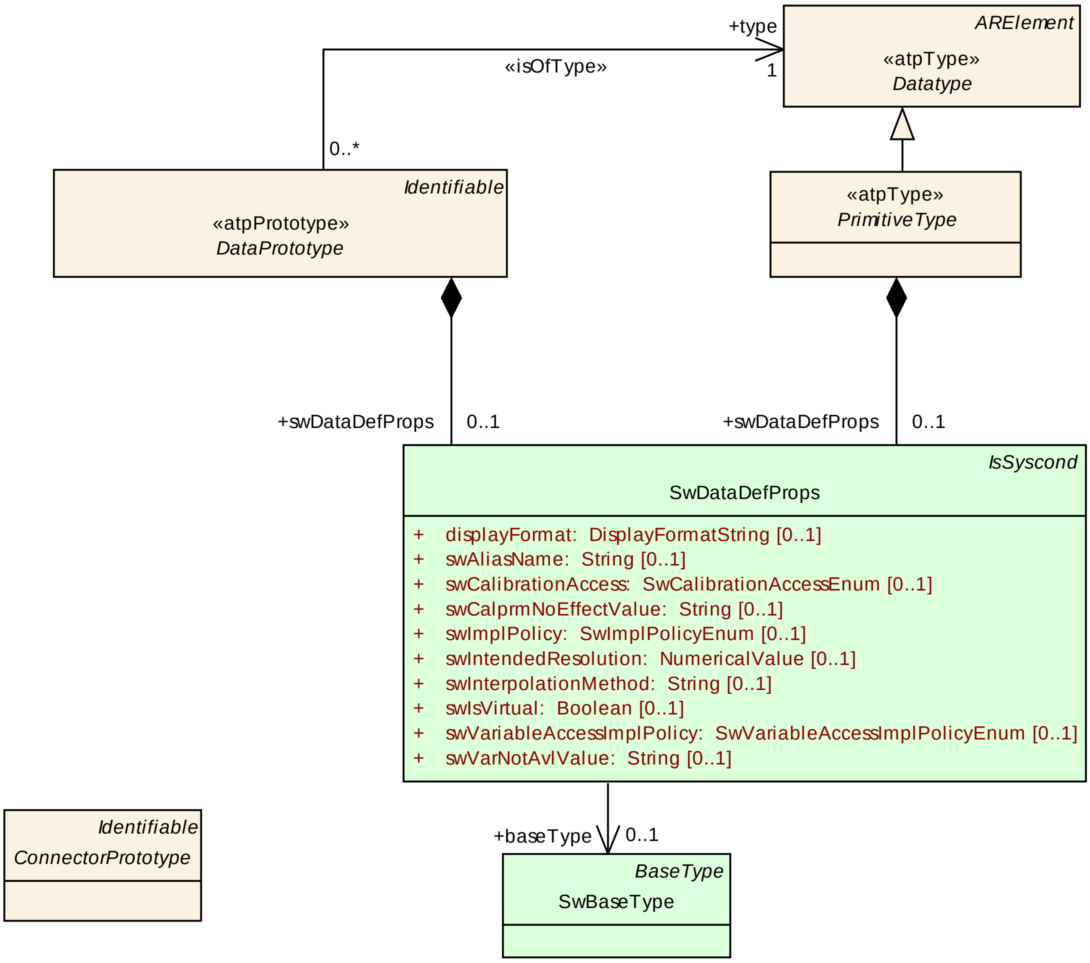


Figure 8.5: Type Determination of Calibration Data Value axis

For more details see 8.8.

8.6 Using Calibration Parameters

As mentioned above, a `CalprmElementPrototype` can be used in the context of `InternalBehavior` as well as in the context of `PortPrototypes`.

8.6.1 Sharing Calibration Parameters within Compositions

This case is based on `ComponentTypes`, `PortPrototypes`, and `PortInterfaces`. As provider, a dedicated software component called `CalprmComponentType` (see 8.6), which is derived from `ComponentType`, has to be used as prototype. This dedicated software component type has no `InternalBehavior` and employs exclusively `PPortPrototypes` of type `CalprmInterface`.

Class	⟨⟨**atpType**⟩⟩ **CalprmInterface**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::Characteristic			
Class Desc.				
Base Class(es)	PortInterface			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
calprm Element	Calprm Element Prototype	*	aggregation	

Table 8.3: CalprmInterface

Every software `ComponentType` requiring access to shared Calibration Parameters will have an `RPortPrototype` typed by a `CalprmInterface`. The definition of this shared calibration access in a composition context will be defined by creating a `ConnectorPrototype` between both `SoftwareComponentPrototype` entities.

A `ConnectorPrototype` will only be valid if the referenced `RPortPrototype` and `PPort-Prototype` are typed by the same interface. Calibration access can be provided and required even over compositions using delegation and assembly connectors.

This means that each access to calibration values between `ComponentPrototypes` is explicitly visible. If a connector spans after the mapping of software `ComponentPrototypes` over two different ECUs, the system generation process has to ensure the proper allocation of the `CalprmElementPrototype` (see 8.7) while the calibration system has to cope with setting the parameter synchronously.

Class	⟨⟨**atpType**⟩⟩ **CalprmComponentType**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components
Class Desc.	
Base Class(es)	ComponentType

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 8.4: CalprmComponentType

8.6.2 Sharing Calibration Parameters between ”SoftwareComponentPrototypes” of the Same ”ComponentType”

To use the same Calibration Parameters between several `SoftwareComponentPrototypes` of the same `SoftwareComponentType`, a `CalprmElementPrototype` is attached to an `InternalBehavior` in `sharedCalprm` role.

When the `InternalBehavior` is later on attached to an `AtomicSoftwareComponentType`, the actual calibration values of the `CalprmElementPrototype` is the same for all `ComponentPrototypes`.

A typical example for this kind of sharing code between instances is dealing with two lambda sensors in multiple cylinder-bank engines, where (at least) two `ComponentPrototypes` for each lambda sensor will use the very same Calibration Parameters.

8.6.3 Providing Instance Individual Characteristic Data

To provide instance individual Calibration Parameters, a `CalprmElementPrototype` is attached to an `InternalBehavior` in `perInstanceCalprm` role. When the latter is attached to a `SoftwareComponentType`, the actual calibration values are specific for each `ComponentPrototype`.

The provision of an initial value of calibration parameters owned by `PortPrototype`s is described in section 3.6.1. The same mechanism can be applied to `sharedCalprm` and `perInstanceCalprm`. That is, `InternalBehavior` might aggregate `LocalParameterInitValueAssignment` in the role `initValue` in order to allow for the provision of initial values of local calibration parameters.

Class	**⟨⟨atpObject⟩⟩ LocalParameterInitValueAssignment**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::SwcInternalBehavior::ComponentLocalCalprm			
Class Desc.	This is the specialization for local parameters.			
Base Class(es)	InitValueAssignment			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 8.5: LocalParameterInitValueAssignment

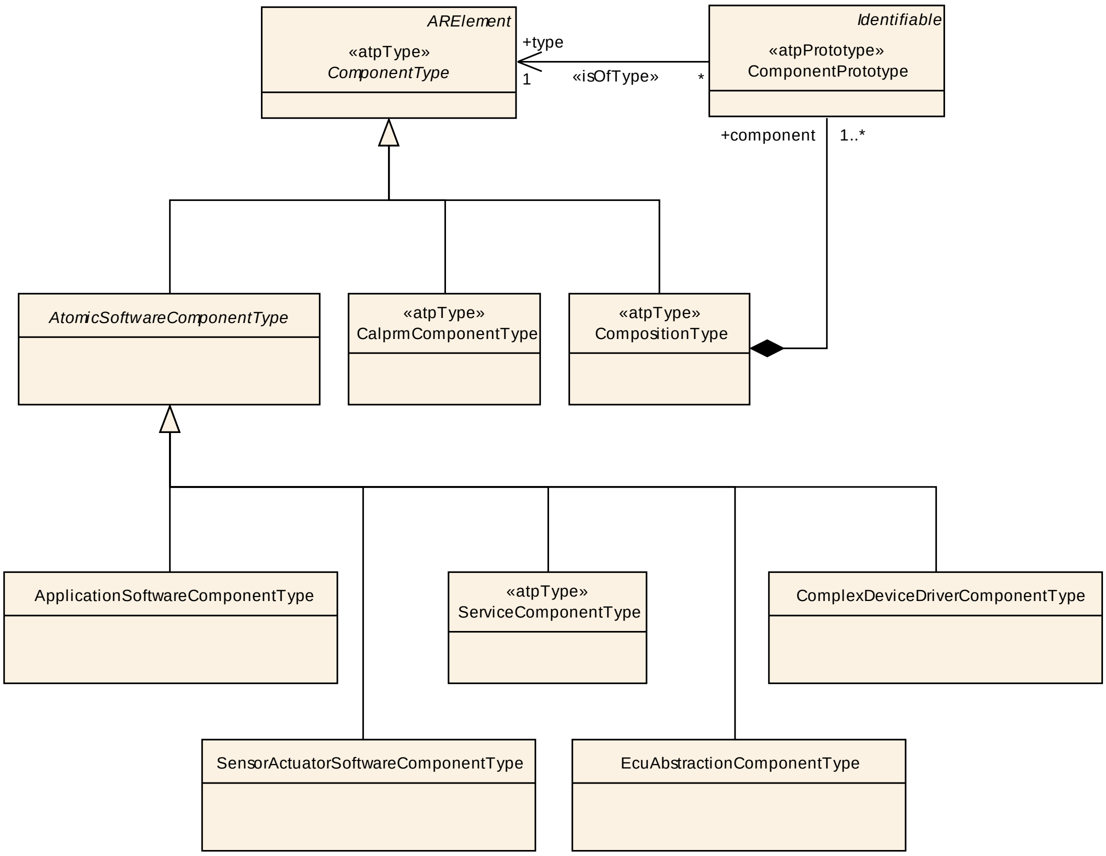


Figure 8.6: CalprmComponentType

8.6.4 Setting an "SwAxis" Input Value

When an interpolation routine is called, an input value has to be provided to find the appropriate axis entry in the implementation of a runnable. However, this input value cannot be arbitrarily chosen, but only be selected from available `DataPrototype` entities having a `Measurable` entity assigned to it.

Every `CalprmElementPrototype` allows to specify zero or more input values in its axis description. This means that at the specification time of an internal behavior a list of input values has to be specified where the implementor of a runnable can choose of. The input values are `DataPrototype` entities either being

- a `DataElementPrototype` in a `SenderReceiverInterface` of a `PortPrototype`, of the `AtomicSoftwareComponentType` where the `InternalBehavior` is associated to, or an `ArgumentPrototype` in an `OperationPrototype` of a `ClientServerInterface` in a `PortPrototype` of the `AtomicSoftwareComponentType` where the `InternalBehavior` is associated to, or
- an `InterRunnableVariable` within the `InternalBehavior`.

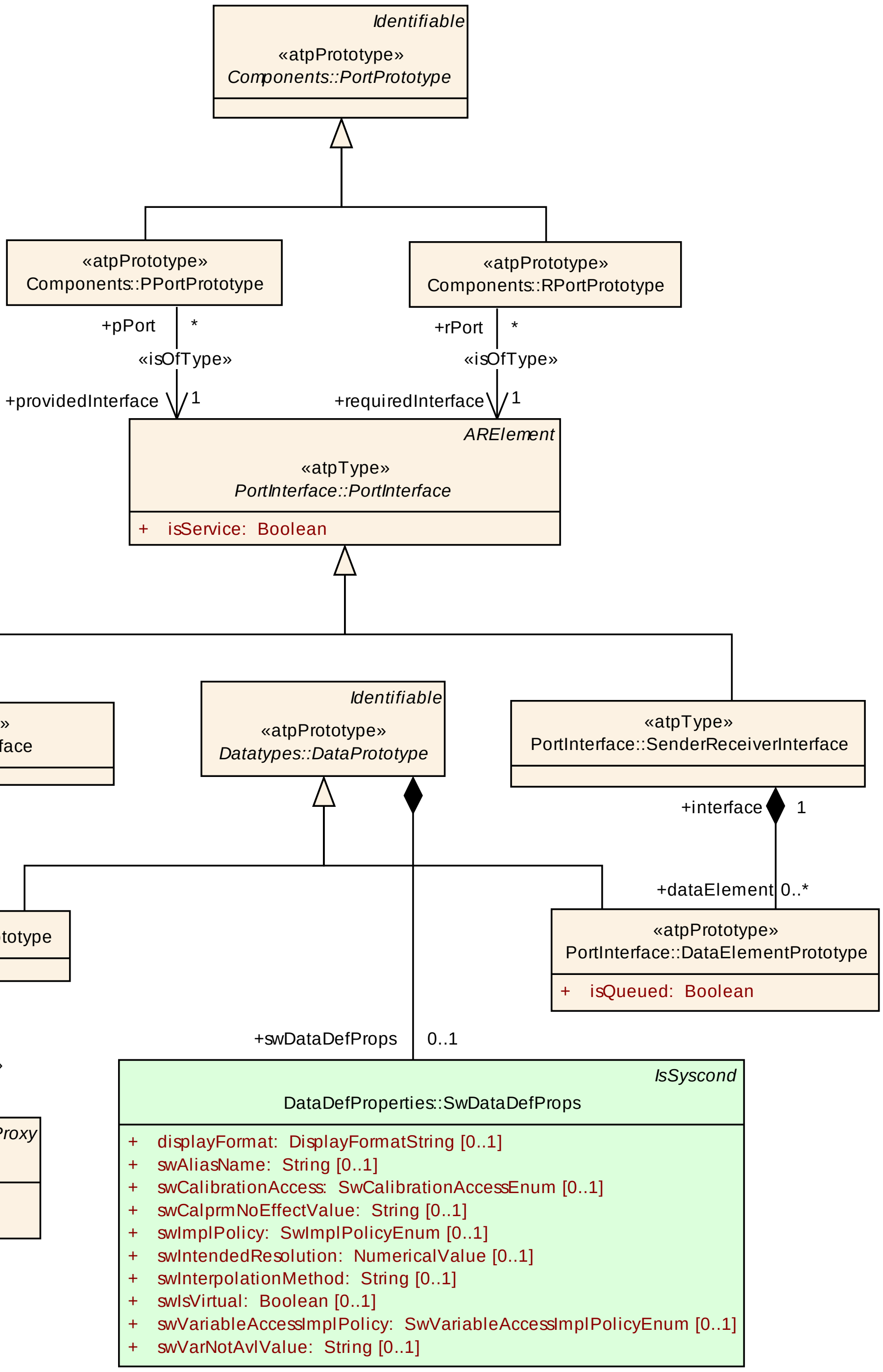


Figure 8.7: CalprmElementPrototype

To achieve this, `SwAxisIndividual` is referencing a `SwVariableRefProxy`. This proxy is an abstract class being refined in AUTOSAR style by a `DataPrototypeRefProxy` entity as shown in 8.9. This `DataPrototypeRefProxy` has an instanceRef to a `DataPrototype` in the appropriate context.

Class	⟨⟨**atpObject**⟩⟩ **SwVariable**			
Package	M2::AUTOSARTemplates::CommonStructure::Variable			
Class Desc.	This element specifies a variable in the ECU. Variables are not adapted to the vehicle in the calibration phase. They are manipulated during the normal operation of the software. Sub-structures are simulated through the aggregation of further swVariables .			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
swArray-size	SwArray-size	0..1	aggregation	Specifies the size in case the variable is an array.
swDataDef Props	SwData DefProps	0..1	aggregation	Associated SwDataDefProps describing the technical characteristics of the variable.
swVariable	SwVariable	*	aggregation	Reference used to specify a sub-structure.

Table 8.6: SwVariable

Class	⟨⟨**atpObject**⟩⟩ **SwCalprm**			
Package	M2::AUTOSARTemplates::CommonStructure::CalibrationParameter			
Class Desc.	This element specifies the properties of calibration parameters in the ECU. Calibration parameters are adapted to the vehicle in the calibration phase. Variables are quite the opposite, they are manipulated during the normal operation of the software. The category of the calprm is used to specify particular shapes of calibration parameters (e.g. the categories as defined by ASAM MDX)			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
swArray-size	SwArray-size	0..1	aggregation	Array size in case the parameter is an array.
swCalprm	SwCalprm	*	aggregation	Sub-structure is simulated through the recursive use of SwCalprm.
swDataDef Props	SwData DefProps	0..1	aggregation	Data properties for this calibration parameter.

Table 8.7: SwCalprm

Class	⟨⟨**atpObject**⟩⟩ **SwCalprmAxisSet**			
Package	M2::AUTOSARTemplates::CommonStructure::CalibrationParameter			
Class Desc.	This element specifies the input parameter axes (abscissas) of parameters (and variables, if these are used adaptively).			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
swCalprm Axis	SwCalprm Axis	*	aggregation	One axis belonging to this SwCalprmAxisSet

Table 8.8: SwCalprmAxisSet

Class	⟨⟨**atpObject**⟩⟩ **SwCalprmAxis**			
Package	M2::AUTOSARTemplates::CommonStructure::CalibrationParameter			
Class Desc.	This element specifies an individual input parameter axis (abscissa).			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
category	CalprmAxisCategoryEnum	0..1	aggregation	This property specifies the category of a particular axis.
baseType	SwBaseType	0..1	reference	The SwBaseType to be used for the axis.
displayFormat	DisplayFormatString	0..1	aggregation	This property specifies how the axis values shall be displayed e.g. in documents or in measurement and calibration tools.
swAxisIndex	String	0..1	aggregation	Describes the index referring to the axis currently described, for which the contents is specified.
swCalibrationAccess	SwCalibrationAccessEnum	0..1	aggregation	Describes the applicability of parameters and variables.
swCalprmAxisTypeProps	SwCalprmAxisTypeProps	1	aggregation	specific properties depending on the type of the axis.

Table 8.9: SwCalprmAxis

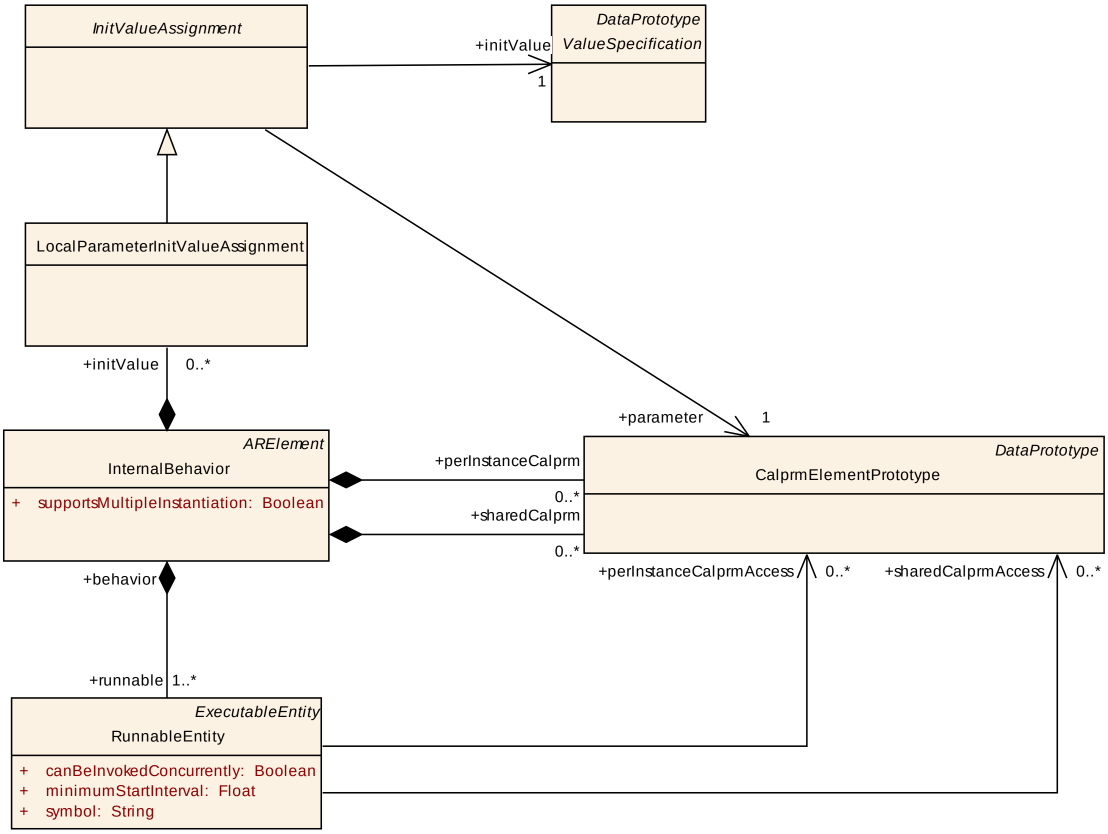


Figure 8.8: CalprmElementPrototypes in internal behavior

Class	**⟨⟨atpObject⟩⟩ SwCalprmAxisTypeProps (abstract)**			
Package	M2::AUTOSARTemplates::CommonStructure::CalibrationParameter			
Class Desc.	Base class for the type of the calibration axis. This provides the particular model of the specialization. If the specialization would be the directly from SwCalPrmAxis, the sequence of common properties and the specializes ones would be different.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 8.10: SwCalprmAxisTypeProps

Class	**⟨⟨atpObject⟩⟩ SwCalprmAxisIndividualAxis**			
Package	M2::AUTOSARTemplates::CommonStructure::CalibrationParameter			
Class Desc.	Container for the properties of an individual axis.			
Base Class(es)	SwCalprmAxisTypeProps			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
swAxis Individual	SwAxis Individual	1	aggregation	The grouped axis contained.

Table 8.11: SwCalprmAxisIndividualAxis

Class	⟨⟨**atpObject**⟩⟩ **SwAxisIndividual**			
Package	M2::AUTOSARTemplates::CommonStructure::Axis			
Class Desc.	This element describes an axis integrated into a parameter (field etc.). The integration makes this individual to each parameter. The so-called grouped axis represents the counterpart to this. It is conceived as an independent parameter (see class SwAxisGrouped). The attributes swVariableRefs, compuMethod and unit can exist in parallel, although physically speaking, only one is practical. This parallelism introduces flexibility into the development process, as axes can be described purely physically, without a conversion formula being available. The following priority exists: * swVariableRefs * compuMethod * unit			
Base Class(es)	ARObject			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
compu Method	Compu Method	0..1	reference	
dataConstr	DataConstr	0..1	reference	Refers to constraints, e.g. for plausibility checks.

swAxis Generic	SwAxis Generic	0..1	aggregation	This element defines an axis for the base points calculated in the ECU. The ECU is equipped with a fixed calculation algorithm. Parameters for the algorithm can be stored in the data component of the ECU. The following is valid: * The algorithm to be used is specified as <swAxisType> in the data dictionary ** (reservation of keyword and specification of parameters). Thus when forming an axis, the algorithm is given through the appropriate reference (<swAxisTypeRef>). * The number of base points to be calculated is defined in <SW-NUMER-OF-AXIS-POINTS>. This element exists to enable the number of axis points to be stored explicitly, although it could also be described as <swGenericAxisParam> . * The calculated base points can be stored on a physical level in the element <swValuesPhys> , which means that it is not necessary for the required calculation algorithm to be implemented in every MCD system. * The calculated base points can be stored on a standardized level in the element <swValuesCoded> , which means that it is not necessary for the required calculation algorithm to be implemented in every MCD system.
swMaxAxis Points	Vf	0..1	aggregation	Maximum number of base points contained in the axis of a map or curve.
swMinAxis Points	Vf	0..1	aggregation	This element specifies the minimum number of base points on the current axis of a map or curve.

swVariable Ref	SwVariable RefProxy	*	aggregation	Refers to an input variable of the axis. It is possible to specify more than one variable. Here the following is valid: * The variable with the highest priority must be given first. It is used in the generation of the code and is also displayed first in the application system. * All variables referenced must be of the same physical nature. This is usually detected in that the conversion formulae affected refer back to the same SI-units. * This multiple referencing allows a base point distribution for more than one input variable to be used. One example of this are the temperature curves, which can depend both on the induction air temperature and the engine temperature. These variables can be displayed simultaneously by MCD systems (adjustment systems), enabling operating points to be shown in the curves.
unit	Unit	0..1	reference	Use <unit> to enter the unit of a parameter.

Table 8.12: SwAxisIndividual

Originally, MSRSW uses a `SwVariableRef` to set the input value of an axis appropriately. In AUTOSAR, this has been extended by first introducing a `SwVariableRefProxy`. This will then be derived in `DataPrototypeRef` (AUTOSAR style) or `SwVariableRef` (MSR style).

As shown in 8.9 this approach is also used to represent a `DataPrototypeRef` in the roles of `swTargetValue`, i.e. the result of an interpolation routine applied to an axis, and a tentative `swHostVariable`, which can be used for an optimized bit-variable representation, and, as described above, the input value determination, a `swSemaphore`, and a list of dependent parameters, `swDataDependency`.

Class	⟨⟨**atpObject**⟩⟩ **SwVariableRefProxy (abstract)**			
Package	M2::AUTOSARTemplates::CommonStructure::DatadictionaryProxies			
Class Desc.	Parent class for several kinds of references to a variable.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 8.13: SwVariableRefProxy

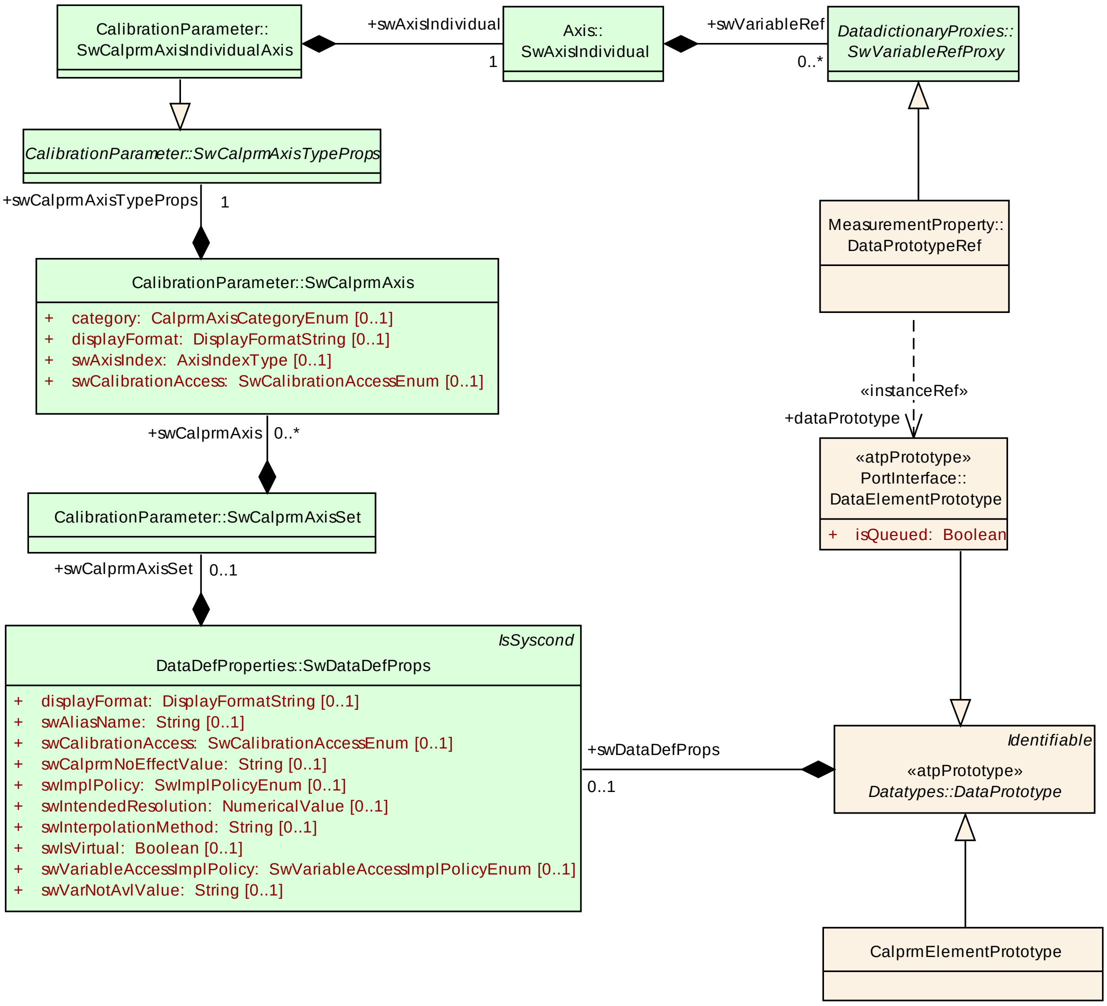


Figure 8.9: Extended Axis Elements and Input Variable Reference

Class	⟨⟨**atpObject**⟩⟩ **DataPrototypeRef**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::MeasurementProperty			
Class Desc.				
Base Class(es)	SwVariableRefProxy			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
dataPrototype	DataElementPrototype	1	instanceRef	

Table 8.14: DataPrototypeRef

Grouped curves share the same axis definition. In MSRSW, this is shown by referencing the `SwCalprm`, representing an individual curve, from a `SwAxisGrouped`. AUTOSAR applies a similar proxy approach for the `SwCalprm` as for the `SwVariable`. Therefore, a `SwCalprmProxy` is introduced in MSRSW, and is aggregated by the `SwAxisGrouped` element.

Class	⟨⟨**atpObject**⟩⟩ **SwAxisGrouped**			
Package	M2::AUTOSARTemplates::CommonStructure::Axis			
Class Desc.	An SwAxisGrouped is an axis which is shared between multiple calibration parameters.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
swCalprm	SwCalprm RefProxy	1	aggregation	This property specifes the calibration parameter which serves as the input axis.

Table 8.15: SwAxisGrouped

The `SwCalprmProxy` is refined into `ArCalprmRef` providing an association to a `CalprmElementPrototype`, representing a curve with an axis. The AUTOSAR-style is shown in the upper left part of 8.11, while in the upper middle the MSRSW style is shown, referencing the `SwCalprm`.

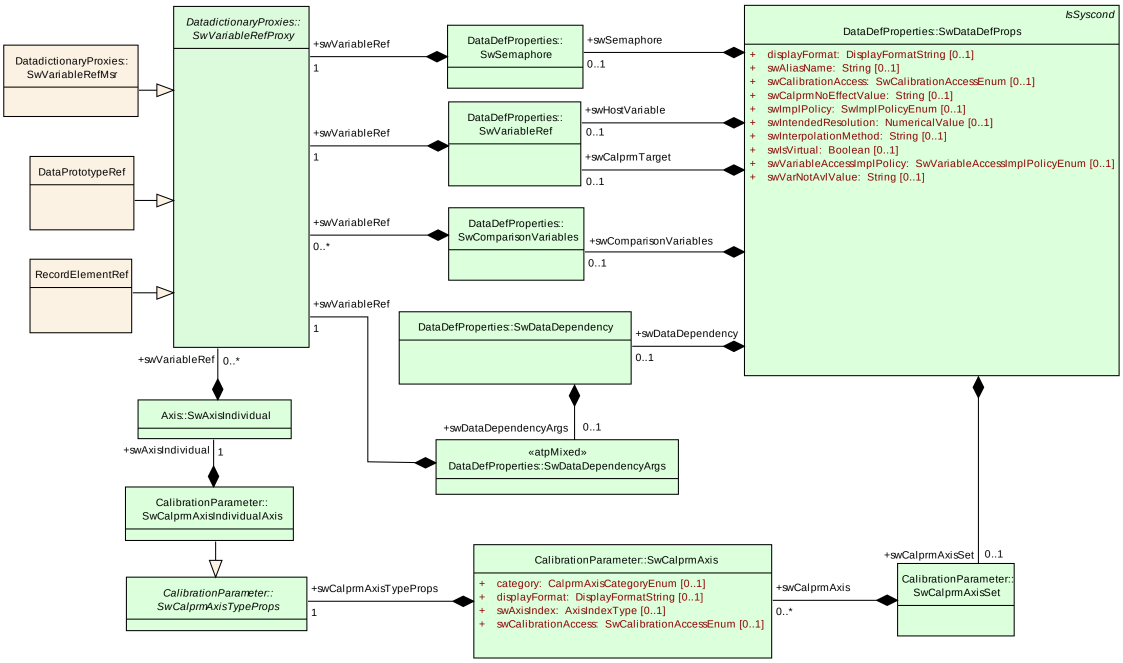


Figure 8.10: Extended Variable Reference Mechanism

Grouped curves share the same axis definition. In MSRSW, this is shown by referencing the `SwCalprm`, representing an individual curve, from a `SwAxisGrouped`.

AUTOSAR applies a similar proxy approach for the `SwCalprm` as for the `SwVariable`. Therefore, a `SwCalprmProxy` is introduced in MSRSW, and is aggregated by the `SwAxisGrouped` element. The `SwCalprmProxy` is refined into `ArCalprmRef` providing an association to a `CalprmElementPrototype`, representing a curve with an axis.

The AUTOSAR-style is shown in the upper left part of 8.11, while in the upper middle the MSRSW style is shown, referencing the `SwCalprm`.

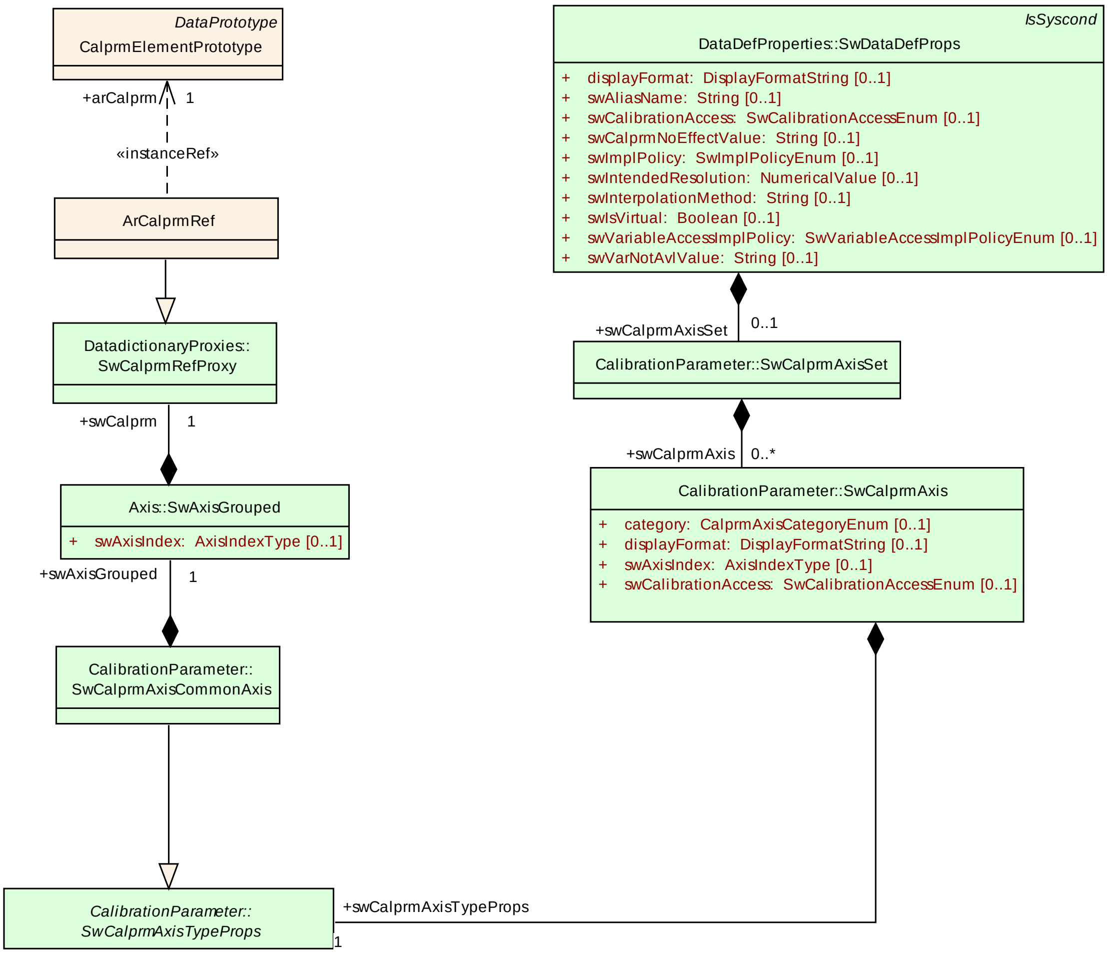


Figure 8.11: Grouped Curves sharing input values of another CalprmElementPrototype

Class	⟨⟨**atpObject**⟩⟩ **ArCalprmRef**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::Characteristic			
Class Desc.				
Base Class(es)	SwCalprmRefProxy			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
arCalprm	Calprm Element Prototype	1	instanceRef	

Table 8.16: ArCalprmRef

8.7 Behavioral Access

There are several ways a Calibration Parameter is provided within a software component. As mentioned above, if Calibration Parameters are shared among several `ComponentTypes` a dedicated `PortInterface` in a `PortPrototype` will be used. The designer of a software-component can use this access mechanism when designing a runnable using, as input value, a `DataPrototype`

- from an arbitrary `RPortPrototype` associated either with a `ClientServerInterface` or a `SenderReceiverInterface`,
- or from an `InterRunnableVariable`

This input value will be fed to an interpolation routine whose result can be used internally or transferred to a neighbored `ComponentPrototype` via dedicated `PortPrototypes`. Typically, there will be a dedicated runnable (with ”ReceiveMode” set to ”activation_of_runnable_entity”) that itself calls the interpolation routine with the appropriate input value and the appropriate ”CalprmElementPrototype”.

The result of this interpolation routine call is provided as an `ArgumentPrototype` with `Direction` being either set to `out` or `inout` in a `ClientServerInterface`.

Class	⟨⟨**atpObject**⟩⟩ **CalprmAccess**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::Characteristic			
Class Desc.				
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
calprm Access	Calprm Element Prototype	1	instanceRef	

Table 8.17: CalprmAccess

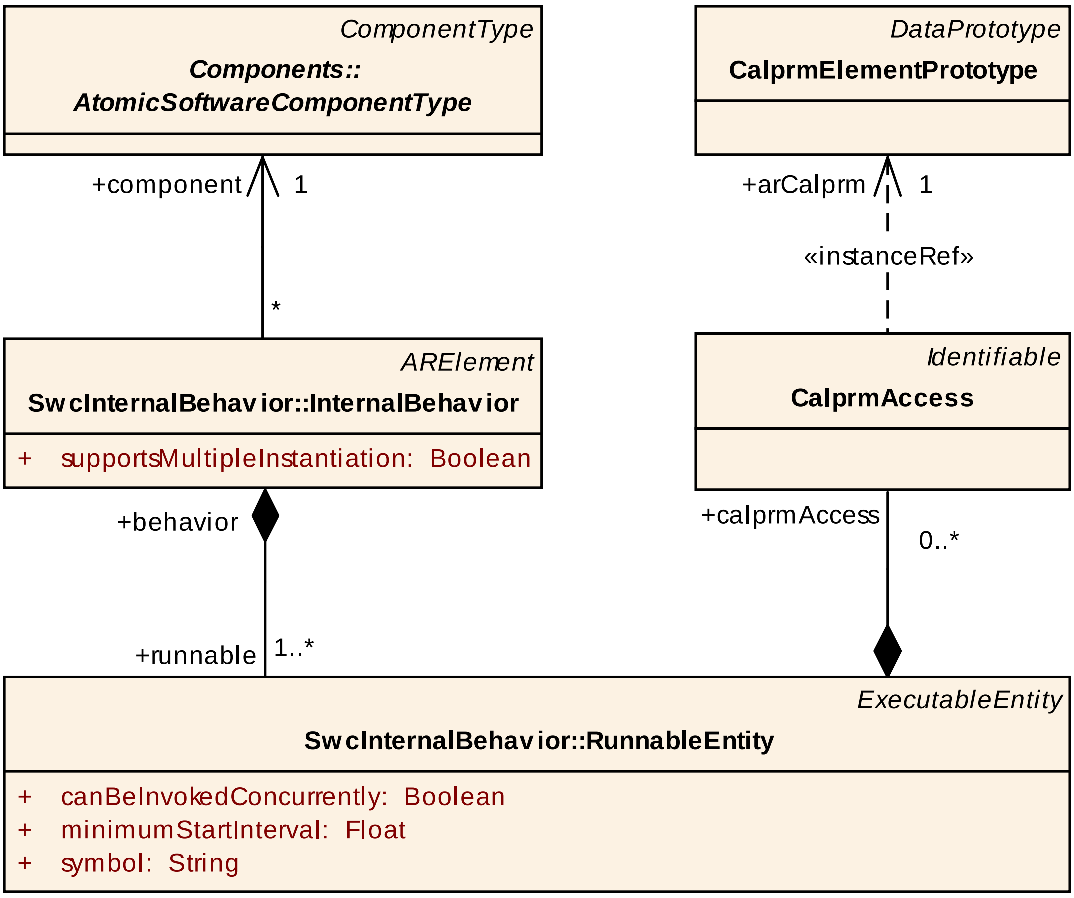


Figure 8.12: Runnable Access to a Calibration Port

The access to a `CalprmElementPrototype` will be indicated

- by the `CalprmAccess` entity if the `RunnableEntity` wants to access it from a `RPortPrototype`. This is shown in 8.12
- by defining the `sharedCalprmAccess` association from a `RunnableEntity` to the `CalprmElementPrototype`. This is shown in 8.8 in the lower association from `RunnableEntity` to `CalprmElementPrototype`
- by defining the `perInstanceCalprmAccess` association from a `RunnableEntity` to every instance of the `CalprmElementPrototype`. This is shown in 8.8 in the upper association from `RunnableEntity` to `CalprmElementPrototype`.

8.8 Addressing Methods

In an ECU there might be various methods to access a particular object (e.g measurement or calibration parameter) according to a given address. This variety might come from different kind of memory (near, far, . . .), but also from indirections which are introduced by the compiler. In order to allow a measurement and calibration system to access such objects `SwAddrMethods` are specified.

`SwAddrMethod` will be used to group calibration parameters with respect to cover the fact that sometimes it is required that one or more calibration parameters out of the mass of calibration parameters of an `CalprmComponentPrototype` respectively an AUTOSAR software component shall be placed in another memory location than the other parameters of the `CalprmComponentPrototype` respectively the AUTOSAR software component.

In `Implementation` the particular `MemorySection` is associated with the `SwAddrMethod`. This association indicates that all objects of the associated adressing method shall be placed in the given memory section. If this association is missing, the object can be placed anywhere without restriction e.g. using a default behavior of the RTE generator. Contradictive specifications (e.g. two diffent component types requiest different associations for one particular SwAddrMethod) must be flagged as an error.

Figure 8.13 illustrates the context for a `DataElementPrototype`.

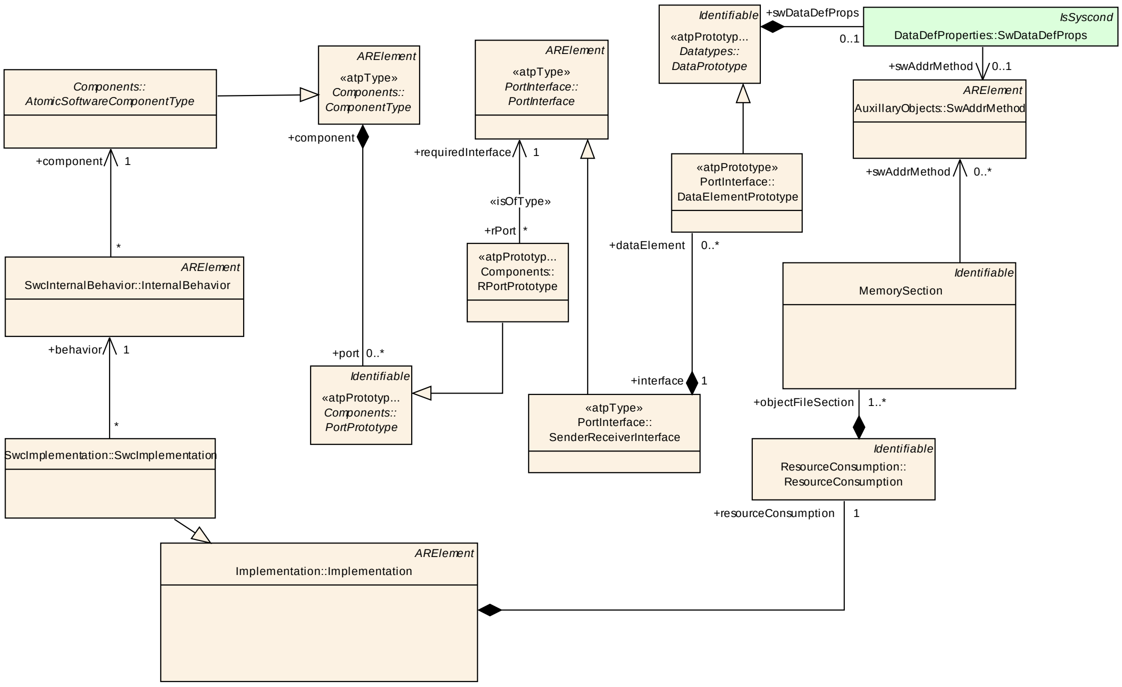


Figure 8.13: Assigning an adress method to a memory section

8.9 Record Layouts

ASAM defines common patterns for the record-layouts of calibration parameters. In AUTOSAR, the selection of the proper category of a ”CalprmElementPrototype” determines the shape of the characteristic.

Via the `SwDataDefProps` a record-layout can be associated to the `CalprmElementPrototype`. On the one hand, if the very same `CalprmInterface` is either used in several `PPortPrototypes` or even `ComponentPrototypes` all resulting instances of the `CalprmElementPrototype` will refer to the same `RecordLayout`.

On the other hand, the record layout has to be known at the time when the interpolation routines are configured. This is supposed to be done at ECU-configuration time prior to the RTE generation.

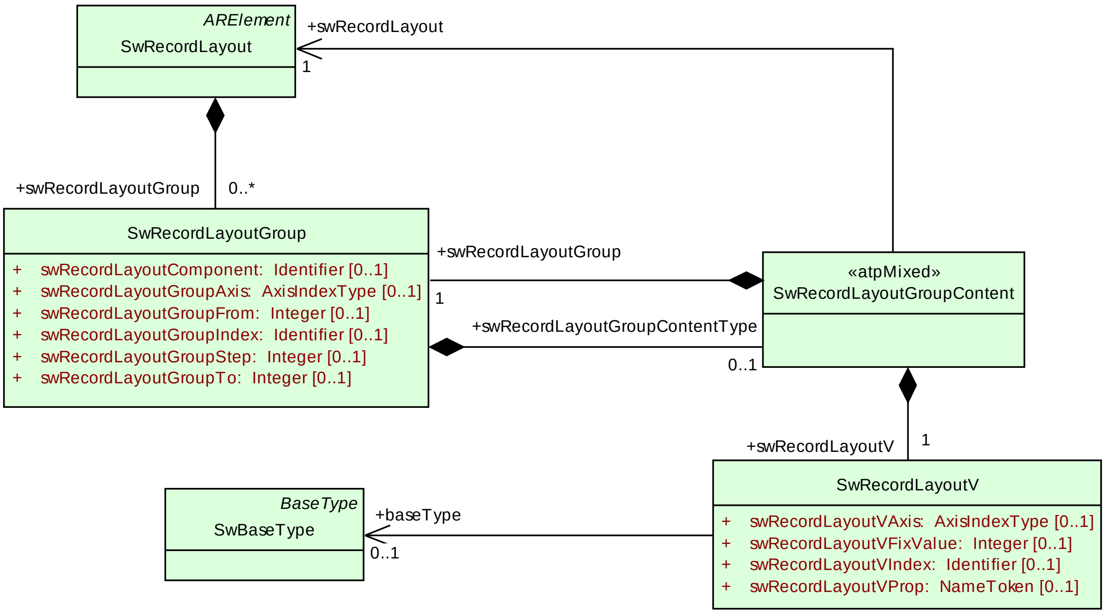


Figure 8.14: Specification of a record layout

The purpose of record layout is to specify how an object (e.g. a calibration parameter) is serialized in memory of an ECU. The basic approach for this is to define nested groups (`SwRecordLayoutGroup`). The Contents (`SwRecordLayoutGroupContent`) is a mixture of (thus nested) groups or particular values (`SwRecordLayoutV`) which refers to particular properties of the object (e.g. value, count, . . .). By this pattern, the serialization of any complex object can be specified.

Class	**⟨⟨atpObject⟩⟩ SwRecordLayoutV**
Package	M2::AUTOSARTemplates::CommonStructure::AuxillaryObjects

Class Desc.	This element specifies which values are stored for the current SwRecordLayoutGroup. If no baseType is present, the SwBaseType referenced initially in the father element SwRecordLayoutGroup is valid. The specification of swRecordLayoutVAxis gives the axis of the values to be stored in accordance with the current record layout SwRecordLayoutGroup. In swRecordLayoutVProp you are able to specify the type of values that are to be stored, e.g. number or value. Under swRecordLayoutVIndex, the symbolic values of the axes can be given, for which the value given under swRecordLayoutVProp is iterated. These symbolic values relate to the values given in swRecordLayoutGroupIndex.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
baseType	SwBase Type	0..1	reference	SwBaseType to be used for the values within this SwRecordLayoutV.
swRecord Layout	SwRecord Layout	0..1	reference	tbd: I (bernhard Weichel) ar not sure if this association is superfluous ...
swRecord LayoutV Axis	String	0..1	aggregation	This attribute specifies the axis from which the value properties are used.
swRecord LayoutVFix Value	Integer	0..1	aggregation	This attribute specifies the filler character for the current record layout, in the form of hex digits. The element present parallel to this in swRecordLayoutVProp must therefore have the contents FILL.
swRecord LayoutV Index	Identifier	0..1	aggregation	The symbolic value for iteration, or the symbolic values separated by white-spaces, refer to the symbolic values given in swRecordLayoutGroupIndex . The iterators are processed from left to right, in such a manner that they symbolize the loop index from the outside to the inside. An error has occurred if a parameter references a record layout which contains an swRecordLayoutVIndex with more components than the number of parameter axes.
swRecord LayoutV Prop	Name Token	0..1	aggregation	The contents of this attribute describes the type of values to be stored in the record.

Table 8.18: SwRecordLayoutV

Class	**⟨⟨atpObject⟩⟩ SwRecordLayoutGroup**
Package	M2::AUTOSARTemplates::CommonStructure::AuxillaryObjects
Class Desc.	Specifies how a record layout is set up. Using SwRecordLayoutGroup it recursively models iterations through axis values. The subelement swRecordLayoutGroupContentType may reference other SwRecordLayouts, SwRecordLayoutVs and SwRecordLayoutGroups for the modeled record layout.
Base Class(es)	ARObject

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
swRecord Layout Component	Identifier	0..1	aggregation	This element is used to denote the component to which the group in question applies. Thus, the record layout supports structured objects. This secures independence from the sequence of components, because they can be referred to via name.
swRecord Layout GroupAxis	String	0..1	aggregation	The contents of this element specifies the axis number within a record layout group.
swRecord Layout GroupContentType	SwRecord Layout Group Content	0..1	aggregation	this is the contents of the recordLayout which is produces for every step of iteration.
swRecord Layout Group From	Integer	0..1	aggregation	This element specifies the iterator index for the point in the axis from which a record layout group is commenced. Negative values are also possible, i.e. the value -4 counts from the fourth value from the end.
swRecord Layout Group Index	Identifier	0..1	aggregation	This element attributes a symbolic name to the iterator of the superimposed record layout group. This can be referenced as a loop index beneath superimposed or subsequent SwRecordLayoutV elements.
swRecord Layout GroupStep	Integer	0..1	aggregation	This element specifies the step width for the iterator index, which is used for a record layout group .
swRecord Layout GroupTo	Integer	0..1	aggregation	This element specifies the iterator index for a point in the axis up to which iteration for a record layout group takes place. Negative values are also possible, i.e. the value -4 counts up to the fourth value from the end.

Table 8.19: SwRecordLayoutGroup

The properties of `SwRecordLayoutGroup` are:

- `swRecordLayoutGroupAxis`: This attribute specifies the axis number within a `SwRecordLayoutGroup`. The current record layout group then refers exactly to the axis with this number.
- `swRecordLayoutGroupIndex`: This attribute assigns a symbolic name to the iterator assigned to the current record layout group. This name can be referenced as a loop index beneath superimposed or subsequent swRecordLay-outV elements. Note that this name can also be used to construct names for appropriate datatypes.

- `swRecordLayoutGroupFrom` specifies the starting point for the iteration. Negative values are also possible, i.e. the value -4 counts from the fourth value from the end.
- `swRecordLayoutGroupTo` specifies the end point for the iteration. Negative values are also possible, i.e. the value -4 counts up to the fourth value from the end.
- `swRecordLayoutGroupeStep` specifies the step width for the iterator index, which is used for the current record layout group. Note that negative values are also possible, in case of the starting point is higher than the endpoint.
- `swRecordLayoutComponent` is used to denote the component to which the group in question applies. Thus, the record layout supports structured objects.This secures independence from the sequence of components, because they can be referred to via name. `swRecordLayoutV` specifies which values are stored for the current record layout group. Possible values are shown below. `swRecordLayoutVprop` specifies, the property of the axis point to be stored, e.g. number or value. Under `swRecordVIndex`, the symbolic values of the axes can be given, for which the value given under `swRecordLayoutVProp` is iterated. These symbolic values relate to the values given in `swRecordLayoutGroupIndex`.

The Properties of `SwRecordLayoutV` are

- `BaseType` allows to refer to a base type in case a specific encoding is in-tended. If no base type is referred, the base type referenced initially in the corresponding `DataPrototype` is to be used.
- `swRecordLayoutVAxis` gives the index of the axis of which values that are stored in the ECU. `swRecordVIndex` refers to the symbolic names of the iterators for which the axis value shall be stored in the ECU. In case of nested iterators (mainly for multidimensional objects) the iteratornames are specified as whitespace separated names. These symbolic names relate to `swRecordLayout-GroupIndex`. The iterators are processed from left to right, in such a manner that they symbolize the loop index from the outside to the inside. It is an error if more components are specified than axis are there in the related calibration parameter.
- `swRecordLayoutVProp` describes the type of values to be stored. The following are permitted:
- `swRecordLayoutVFixValue` specifies the filler character for the current record layout, in the form of hex digits. It is also used to specify the fix value for FIXRIGHTDIFF.

Property	Description
VALUE	The value of the axis for the current axis point
COUNT	The amount of values of the axis
LEFTDIFF	The difference to the previous axis point
RIGHTDIFF	The difference to the next axis point
DIST	The distance value of this axis in case of a fixed axis with distance specification
SHIFT	The shift value of this axis in case of a fixed axis with shift/offset
OFFSET	The offset value of this axis in case of a fixed axis with shift/offset
SOURCE-ADR	The address of the source of this axis (Note that this does not apply to the value axis)
RESULT-ADR	The address of the result for this axis (note that this does not apply to input axis)
ADDRESS	The address of the axis point
FILL	Fill with the hex value specified as contents of `swRecordLayoutFixValue`
FIXLEFTDIFF	Difference between this and a fixed left-hand value specified in `swRecordLayoutFixValue`
FIXRIGHTDIFF	Difference between this and a fixed right-hand value specified in `swRecordLayoutFixValue`

Here you can see an example for a SwRecordLayout noted in XML **Example 8.1**

```
<SW-RECORD-LAYOUT>
  <SHORT-NAME>RecordLayoutCurve</SHORT-NAME>
  <SW-RECORD-LAYOUT-GROUP>
    <SW-RECORD-LAYOUT-V>
      <BASE-TYPE-REF>A_UINT8</BASE-TYPE-REF>
      <SW-RECORD-LAYOUT-V-PROP>SOURCE-ADR</SW-RECORD-LAYOUT-V-PROP>
    </SW-RECORD-LAYOUT-V>
    <SW-RECORD-LAYOUT-V>
      <SW-RECORD-LAYOUT-V-PROP>COUNT</SW-RECORD-LAYOUT-V-PROP>
    </SW-RECORD-LAYOUT-V>
    <SW-RECORD-LAYOUT-GROUP>
      <SW-RECORD-LAYOUT-GROUP-AXIS>1</SW-RECORD-LAYOUT-GROUP-AXIS>
      <SW-RECORD-LAYOUT-GROUP-INDEX>x</SW-RECORD-LAYOUT-GROUP-INDEX>
      <SW-RECORD-LAYOUT-GROUP-FROM>1</SW-RECORD-LAYOUT-GROUP-FROM>
      <SW-RECORD-LAYOUT-GROUP-TO>-1</SW-RECORD-LAYOUT-GROUP-TO>
      <SW-RECORD-LAYOUT-V>
        <SW-RECORD-LAYOUT-V-PROP>VALUE</SW-RECORD-LAYOUT-V-PROP>
        <SW-RECORD-LAYOUT-V-INDEX>x</SW-RECORD-LAYOUT-V-INDEX>
      </SW-RECORD-LAYOUT-V>
    </SW-RECORD-LAYOUT-GROUP>
    <SW-RECORD-LAYOUT-GROUP>
      <SW-RECORD-LAYOUT-GROUP-AXIS>0</SW-RECORD-LAYOUT-GROUP-AXIS>
      <SW-RECORD-LAYOUT-GROUP-INDEX>v</SW-RECORD-LAYOUT-GROUP-INDEX>
      <SW-RECORD-LAYOUT-GROUP-FROM>1</SW-RECORD-LAYOUT-GROUP-FROM>
```

```
      <SW-RECORD-LAYOUT-GROUP-TO>-1</SW-RECORD-LAYOUT-GROUP-TO>
      <SW-RECORD-LAYOUT-V>
        <SW-RECORD-LAYOUT-V-PROP>VALUE</SW-RECORD-LAYOUT-V-PROP>
        <SW-RECORD-LAYOUT-V-INDEX>v</SW-RECORD-LAYOUT-V-INDEX>
      </SW-RECORD-LAYOUT-V>
    </SW-RECORD-LAYOUT-GROUP>
  </SW-RECORD-LAYOUT-GROUP>
</SW-RECORD-LAYOUT>
```

Figure 8.15 and Figure 8.16 illustrate most of these properties.

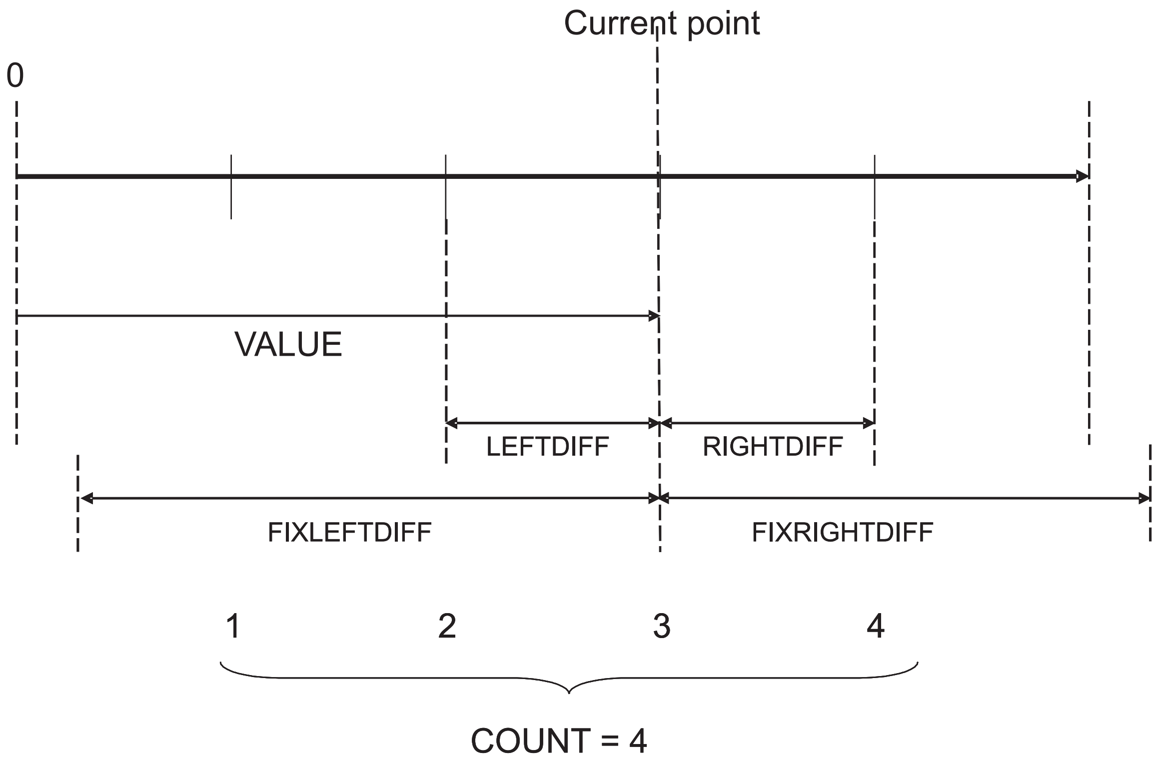


Figure 8.15: Values for swRecordLayoutVProp for individual axis

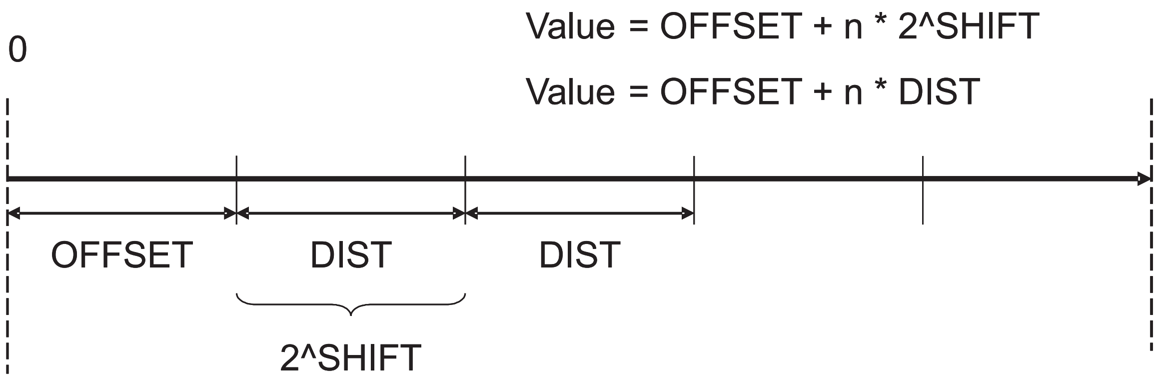


Figure 8.16: Values for swRecordLayoutVProp for fixed axis

8.10 Record Layouts and Data Types

As `DataPrototypes` have an `isOfType` Relation to `DataTypes`, the related data types must properly match to the details as specified in `swDataDefProps` as shown in the diagram

In order to maintain this compliance there are three approaches

- Manually create `DataTypes` for the calibration parameters and compatible `RecordLayouts`

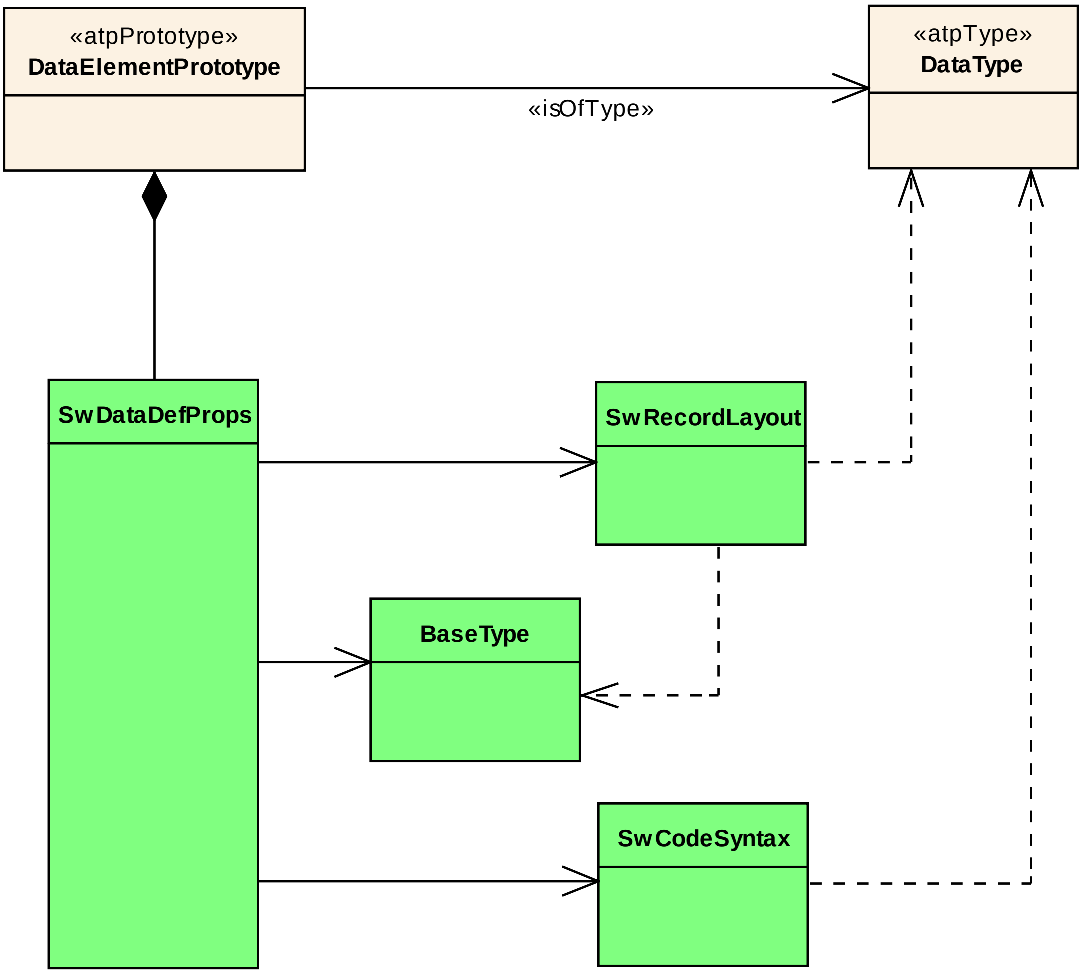


Figure 8.17: Dependency of `DataTypes` and `RecordLayouts`

- Automatically create `DataTypes` from `RecordLayouts`. This could be performed on a model transformation basis according to the algorithm shown below.
- Use `OpaqueDatatypes`. In this case the internals of a calibration parameter is not visible to a software-component. The interpolation has to be done using a service routine.

Note that computing record layouts from data types is not possible, since the particular meaning of the components is not available (`swRecordLayoutVProp`).

The following diagrams illustrate how data types can be derived from record layouts. The blue data types are derived from the record layout.

The algorithm to generate the desired data types are shown in the following two diagrams. We create a data type for each calibration parameter prototype.

For each data type, several subtypes must be created. The details of the algorithm are specified in the Figure 8.22.

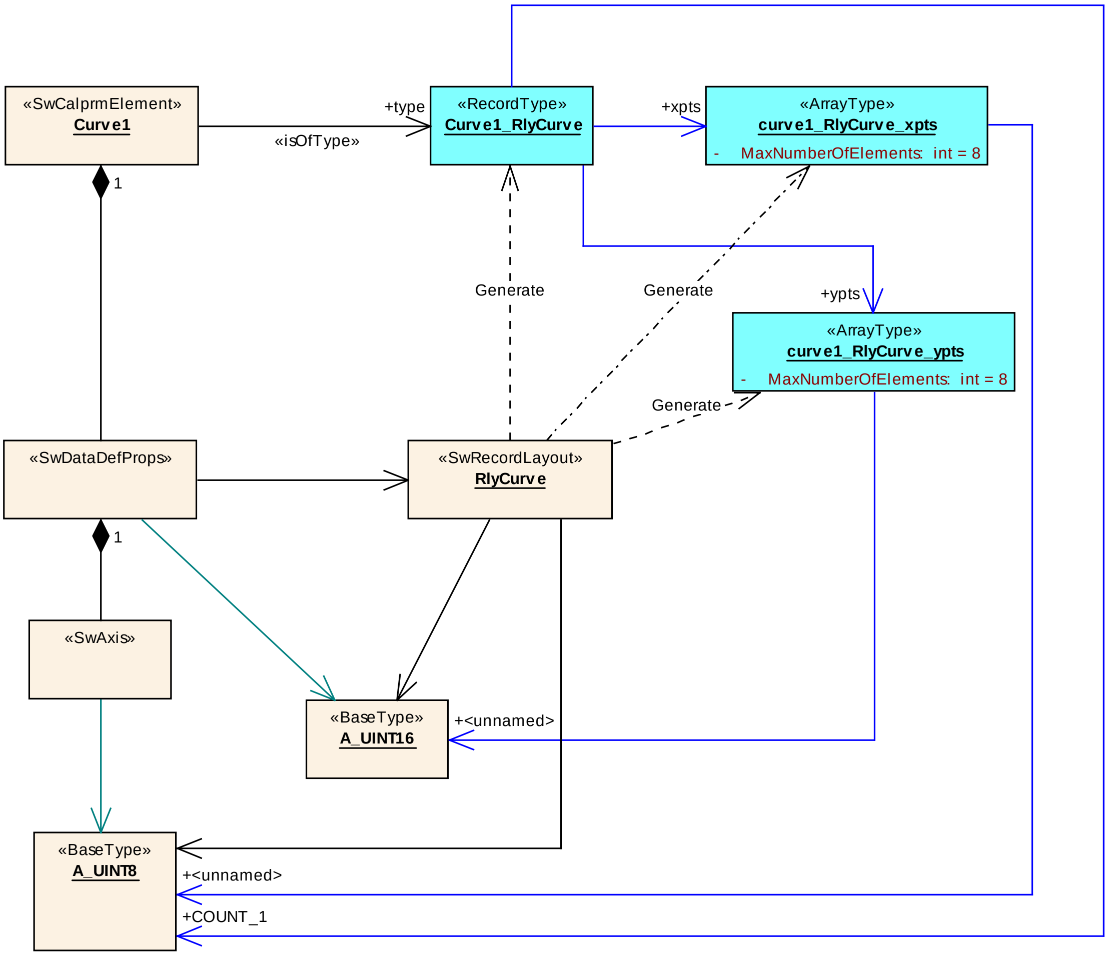


Figure 8.18: Curve implemented as two consecutive arrays

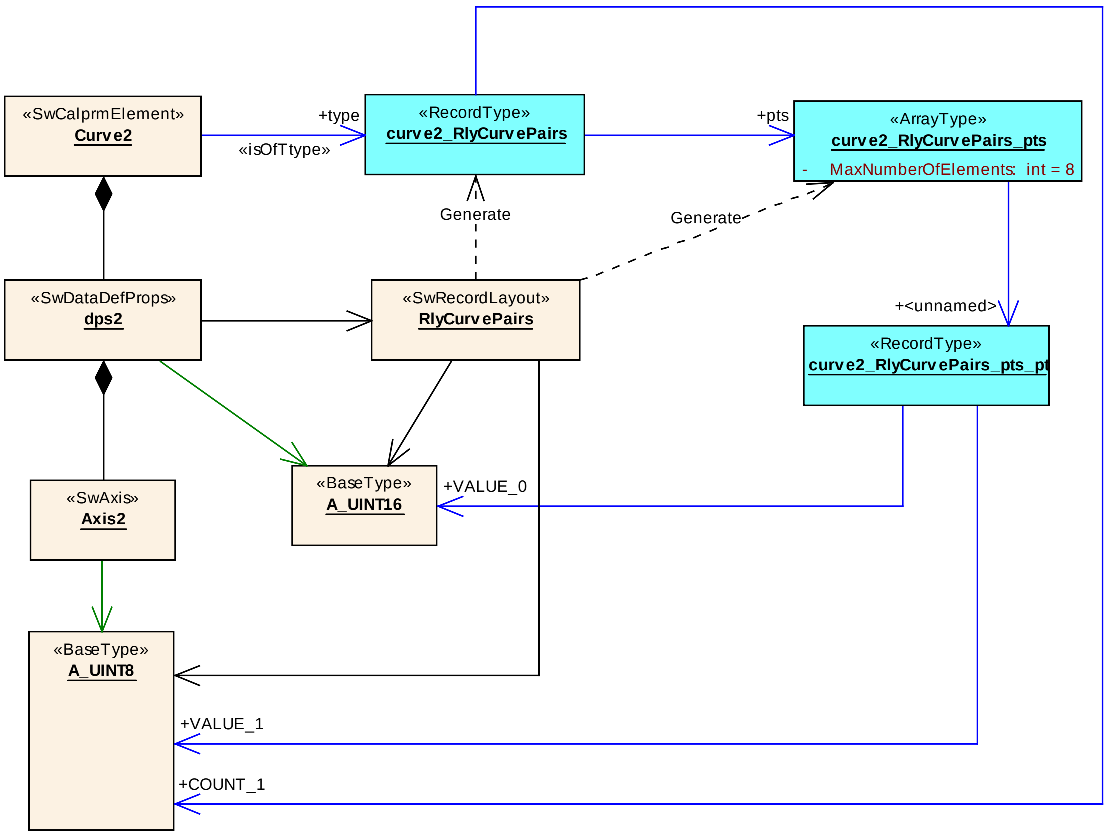


Figure 8.19: Curve implemented as array of record

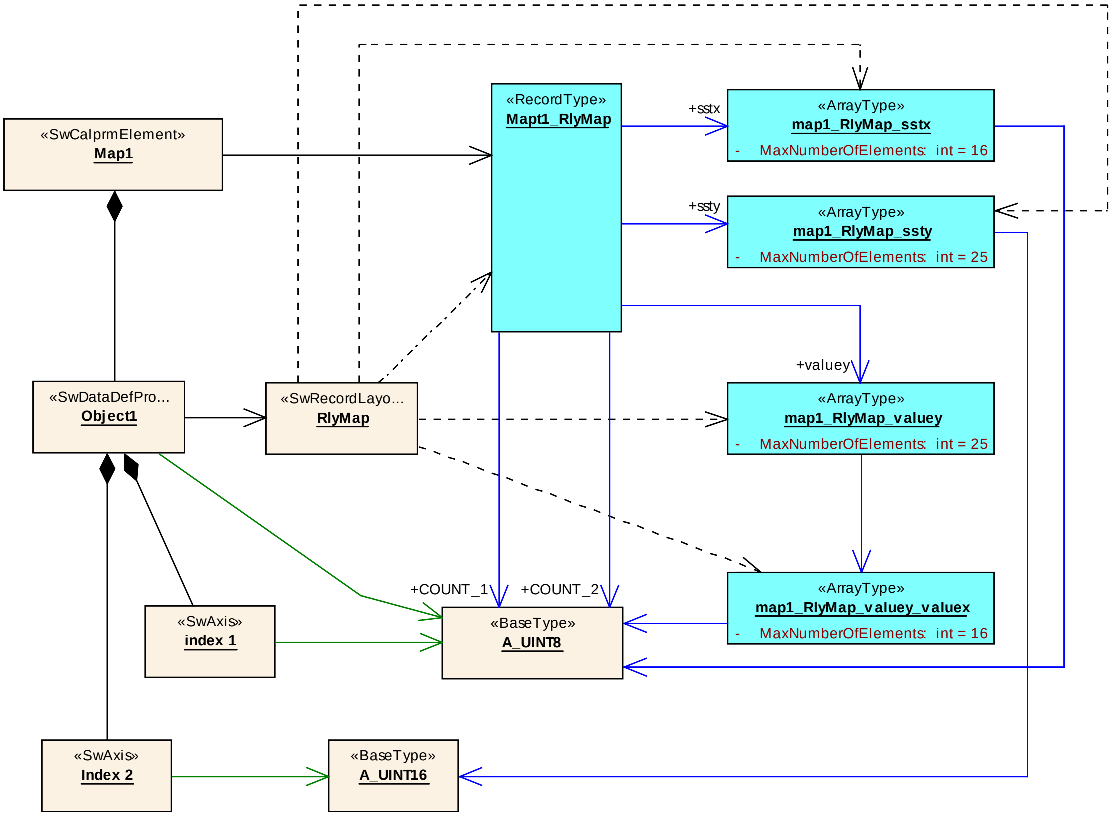


Figure 8.20: Record layout and data type for a map

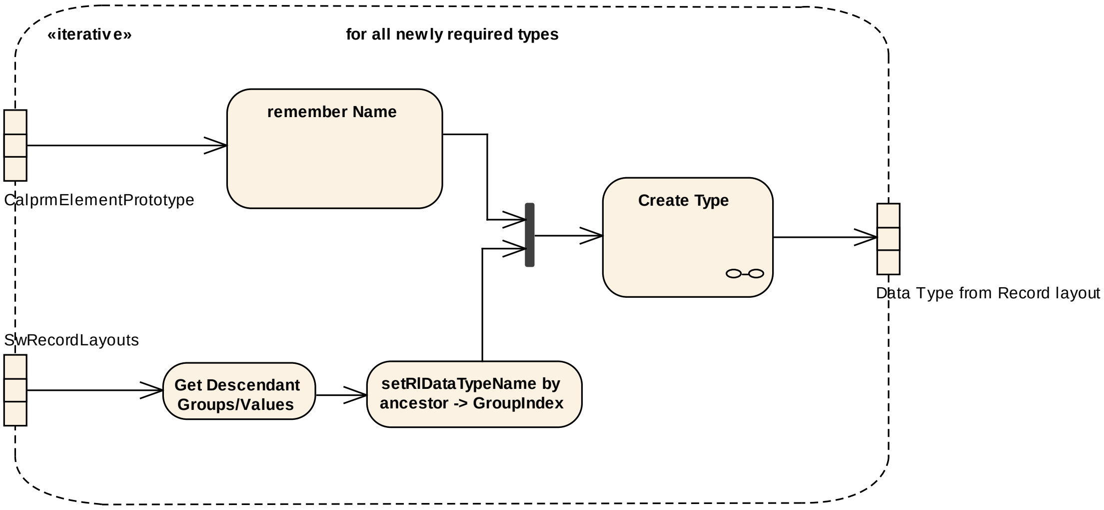


Figure 8.21: algorithm to map record layouts to types

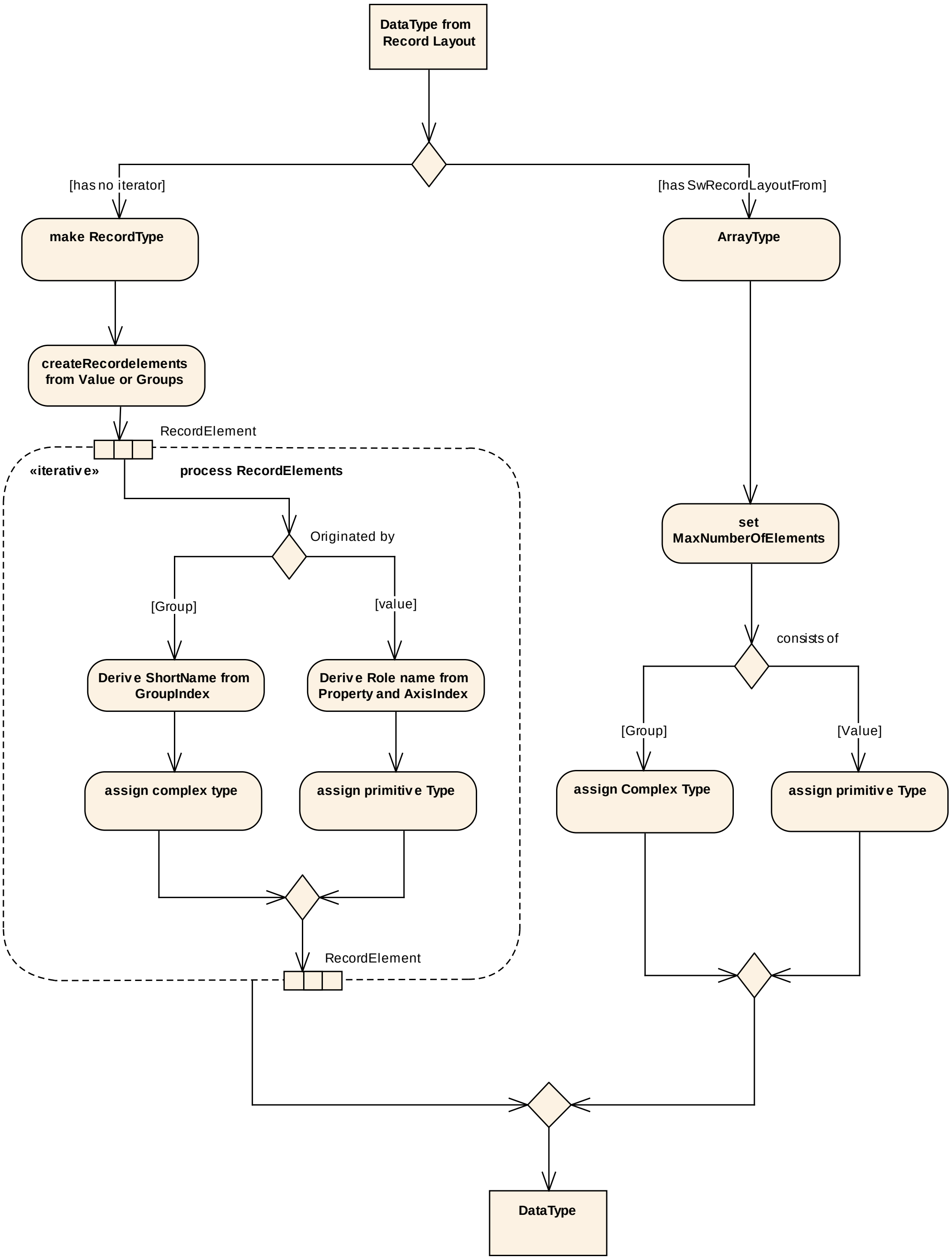


Figure 8.22: Creating types from record layouts

9 ECU Abstraction and Complex Drivers

9.1 Introduction

During the design of embedded systems there is one crucial point where the hardware and software have to be related. In AUTOSAR the `ECU Resource Template` describes the provided hardware resources.

On the other hand, the `Software Component Template` describes software generally without specific hardware in mind. But there are some places where both have to meet and fit.

One interface between hardware and software is discussed in the memory and execution time section of [8]. In this chapter the overall system view of the interface between sensors/actuators and software is described and the consequences for the `Software Component Template` are derived.

9.2 High Level Hardware and Software Architecture

The AUTOSAR concept defines a software architecture (see Figure 9.1) and within this layered architecture the interfaces between the hardware and the software are explicitly modeled.

The signal [1] flow from a hardware to software and vice versa will be described in the following sections.

A sensor [2] is converting a physical value (1) in Figure 9.2 (e.g. temperature, force, light intensity) into an electrical signal (2) which can be either a current or a voltage.

Inside the ECU generally there will be some electronics to enhance the electrical signal provided by the sensor. In AUTOSAR this is called ECU Electronics. This electronics is also responsible for the conversion of the electrical signal into a microcontroller compatible form (3), usually a voltage.

After the electrical signal has been enhanced and converted it will be captured by the microcontroller. This can either be done by a simple digital input, an analogue to digital converter or maybe a pulse-width demodulation module. Now the electrical signal is available as a software data value (4).

This signal flow is sketched in the top part of Figure 9.2.

This signal chain is represented one to one in the AUTOSAR software architecture and depicted in the lower part of Figure 9.2.

---

[1] The term "signal" is not going to be used here at its own but more specific terms will be used for the different abstractions of signals at the different stages of the signal flow.

[2] For the sake of simplicity this discussion is limited to the sensor aspects. Nevertheless, the same applies also for actuators.

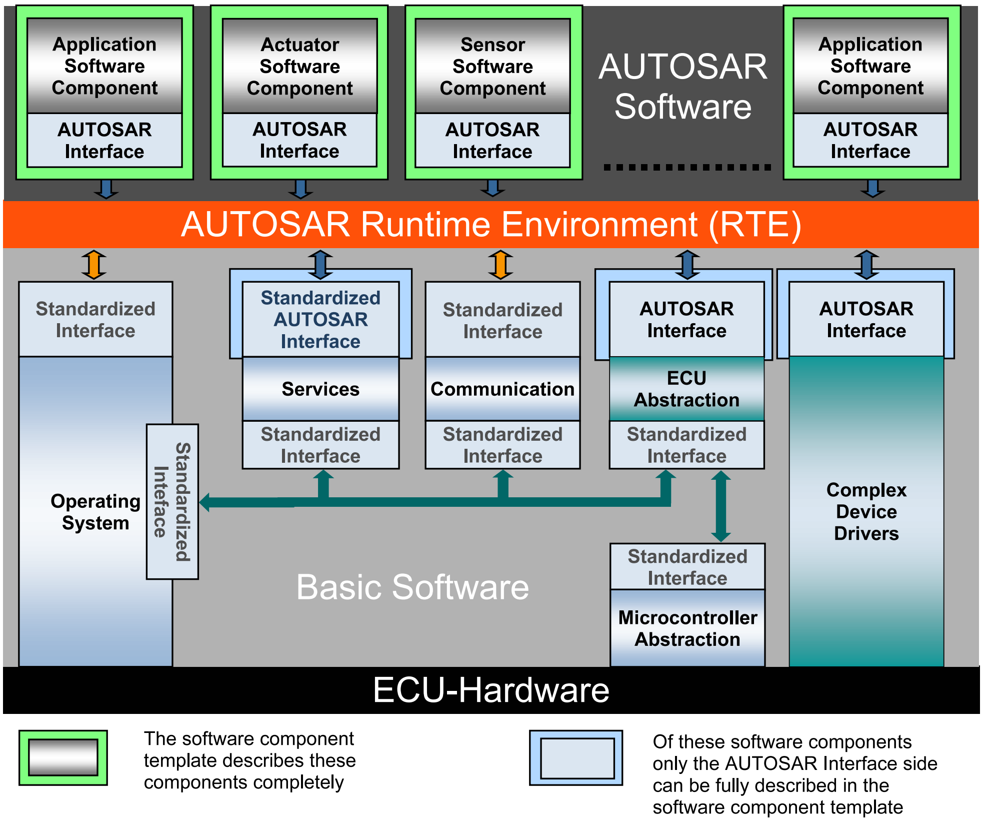


Figure 9.1: AUTOSAR ECU Software Architecture

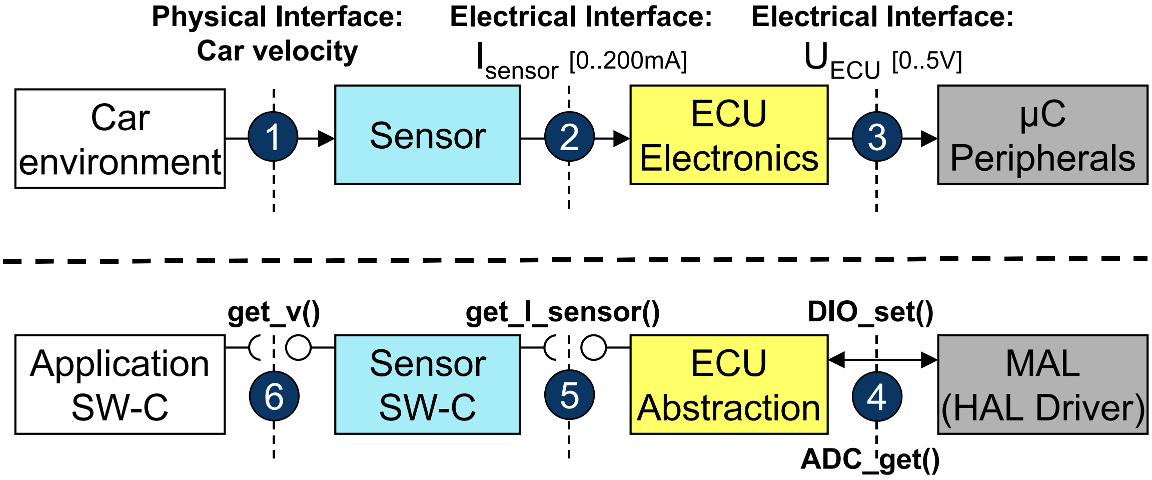


Figure 9.2: Interfaces between hardware and software

In an implementation of AUTOSAR only the Microcontroller Abstraction (MCAL) has direct access to the peripheral hardware. This layer is going to be standardized and all hardware access should go through this layer. The idea of the AUTOSAR signal flow is to map the hardware to the corresponding software modules.

So if an electrical current is the input to the microcontroller peripheral, the MCAL will deliver a data value that represents this current. As the ECU Electronics has enhanced and converted the electrical signal prior to the microcontroller, the corresponding software entity is reversing this conversion. This is performed in the ECU Abstraction layer.

So if the input to the ECU is an electrical current and the ECU Electronics has converted this current into a voltage (from 2 to 3), the ECU Abstraction will convert the data value voltage into an AUTOSAR signal representing a current (from 4 to 5). This AUTOSAR signal represents the actual current that was provided by the sensor (2).

Now the first step in the conversion has to be reversed: the sensor has converted a physical value into an electrical signal. And so the Sensor Software Component has to reverse this again. The Sensor Software Component will read the AUTOSAR signal representing the electrical value and transform it into an AUTOSAR signal representation of the physical value (from 5 to 6).

Now this physical value is available on the RTE and can be consumed or read by other SW-Components. Although the interface between the ECU Abstraction and the Sensor Software Component is also an AUTOSAR interface and could be routed through some communication bus, it will not be practical to separate the ECU Abstraction and the corresponding `SensorActuatorSoftwareComponentType` due to potentially high communication effort.

In Figure 9.3 a complete signal flow from a sensor input to an actuator output is shown.

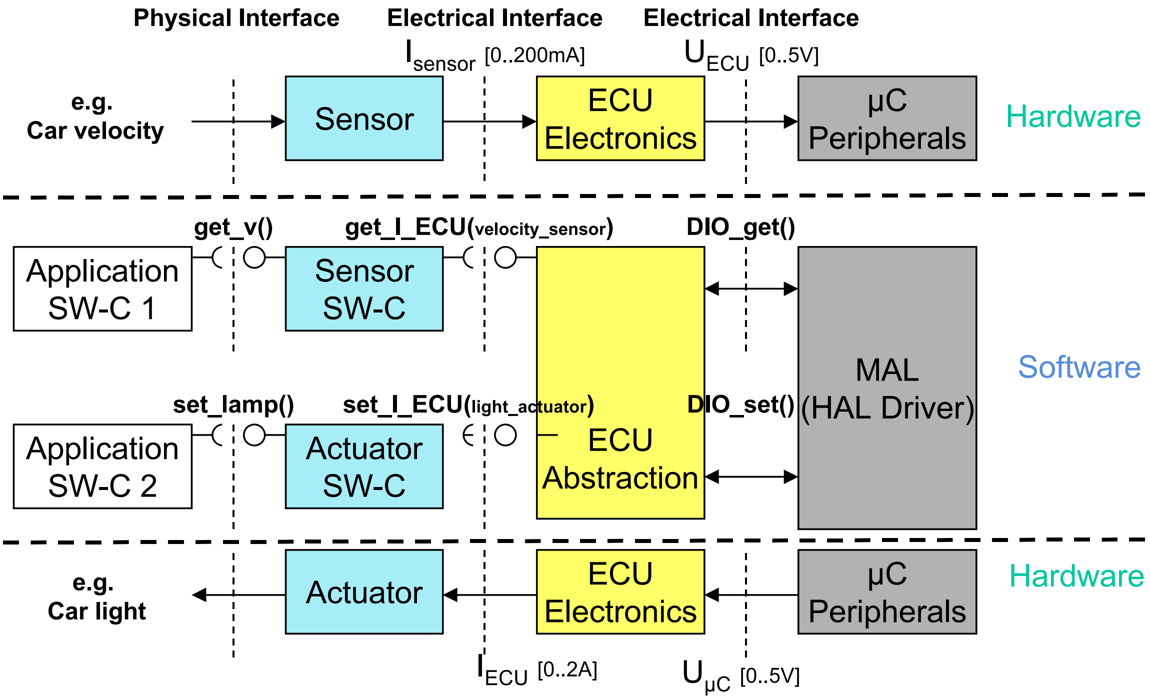


Figure 9.3: Sensor and Actuator Signal Flow

In the next section the interfaces between the involved software modules are discussed.

9.3 Interfaces and APIs

Two fundamentally different interfaces are involved when converting from sensors/actuators to software components, see markers ”4” and ”5” in figure 9.2.

The interface between the Microcontroller Abstraction and the ECU Abstraction is a Standardized Interface (see AUTOSAR Glossary [24]). This interface is not visible on the Virtual Function Bus and therefore the MCAL and ECU Abstraction have to be present on the same ECU.

For further description of this interface please refer to the `ECU Resource Template` documentation.

The interface to the `SensorActuatorSoftwareComponentTypes` is visible on the `Virtual Function Bus`. So the ECU Abstraction and the `SensorActuatorSoftwareComponentTypes` do not need to be present on the same ECU but can be separated. In general the `SensorActuatorSoftwareComponentType` should be on the same ECU as the ECU hardware abstraction.

Also the interface between the `SensorActuatorSoftwareComponentTypes` and the actual `AtomicSoftwareComponentTypes` representing the application is visible on the VFB. To describe the data that is going to be exchanged via this interface the standard AUTOSAR Interface description mechanisms are used (see chapter 2.4).

9.3.1 ECU Abstraction and its AUTOSAR Interfaces

Since the AUTOSAR standard is designed with the focus on the integration of software-components coming from different contractors, the interfaces between the different software-components obviously have to be compatible.

In the case of the sensors and actuators the interface is gathered in the ECU Abstraction. For each sensor and actuator there is one AUTOSAR `PortPrototype` that represents the AUTOSAR Signal that is delivered by the sensor or the AUTOSAR Signal that is consumed by the actuator. This relationship is depicted in figure 9.4

Each sensor and actuator has an AUTOSAR `PortPrototype` at the ECU Abstraction. Connected to this port is the `SensorActuatorSoftwareComponentType`. The `SensorActuatorSoftwareComponentType` has one `PortPrototype` to the ECU Abstraction (IF_2) where it gets the AUTOSAR signals from the hardware, and one `PortPrototype` to `AtomicSoftwareComponentTypes` (IF_3) where it provides the actual physical value to the rest of AUTOSAR on the RTE.

In addition, the Interfaces between the ECU Abstraction and the `SensorActuatorSoftwareComponentType` have to be compatible like defined in chapter 3.4.

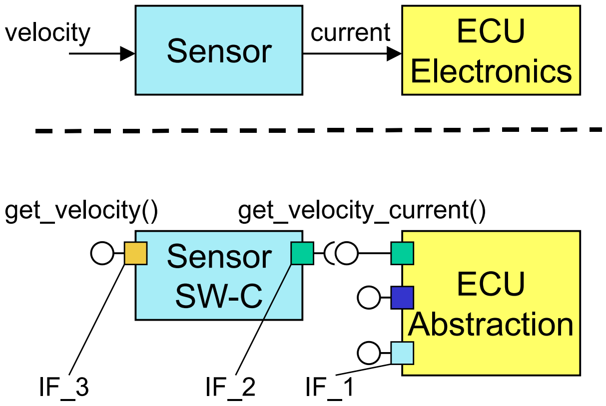


Figure 9.4: Interfaces of signals in software

9.4 Shipment of Sensors/Actuators

In the layered software architecture described in [2] each hardware sensor/actuator is coupled to a `SensorActuatorSoftwareComponentType` (see figure 9.5). Since the `Software Component Template` is going to be used to describe the `SensorActuatorSoftwareComponentType` as well, there is also a reference needed from the software representation of a sensor/actuator to the actual hardware element described in the ECU Resource description.

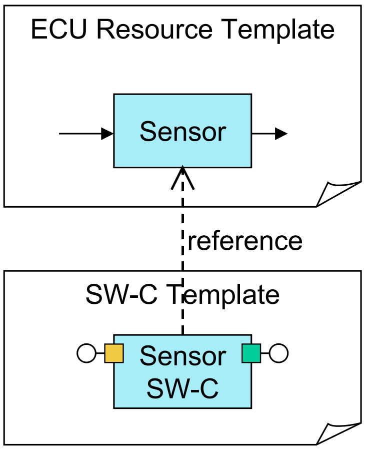


Figure 9.5: Shipment of a sensor

So each time a sensor/actuator is selected to be connected to an ECU also the corresponding `SensorActuatorSoftwareComponentType` is available.

Figure 9.6 depicts the reference of `SensorActuatorSoftwareComponentType` designed as a specialization of an `AtomicSoftwareComponentType` with an additional reference to a `SensorActuatorHW`.

Furthermore, a `SensorActuatorSoftwareComponentType` needs to be mapped and run on exactly that ECU that contains the `SensorActuatorHW` that it refers to in case it accesses the hardware via the I/O hardware abstraction layer. And in contrast to an `AtomicSoftwareComponentType`, an

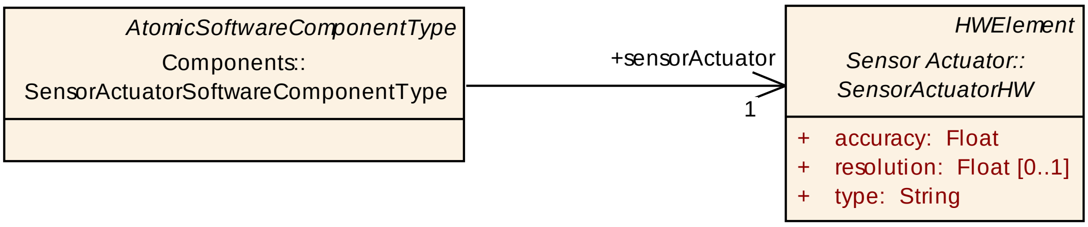


Figure 9.6: Sensor/actuator to Hardware Relationship

`SensorActuatorSoftwareComponentType` may use the I/O hardware abstraction directly (via ports/connectors). In case the sensor/actuator hardware is accessed via bus communication, e.g. is located on a LIN slave, no such mapping constraints apply (note that this is not handled via the IO hardware abstraction layer).

Class	⟨⟨**atpType**⟩⟩ **SensorActuatorSoftwareComponentType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Desc.	The SensorActuatorSoftwareComponentType introduces the possibility to link from the software representation of a sensor/actuator to its hardware description provided by the ECU Resource Template.			
Base Class(es)	AtomicSoftwareComponentType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
sensor Actuator	Sensor ActuatorHW	1	reference	Reference from the Sensor Actuator Software Component Type to the description of the actual hardware.

Table 9.1: SensorActuatorSoftwareComponentType

Class	⟨⟨**atpObject**⟩⟩ **SensorActuatorHW (abstract)**			
Package	M2::AUTOSARTemplates::ECUResourceTemplate::SensorActuator			
Class Desc.	The common attributes for sensors and actuators. The sensor and actuators can be connected via a Peripheral HW Port, a Communication HW Port or a Power Driver HW Port.			
Base Class(es)	HWElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
accuracy	Float	1	aggregation	Defines the error in the representation of the Technical Signal in the data format This applies only if the Technical Signal is encoded before it is transferred to the ECU Electronics (e.g. via Communication Transceiver HW Port).
cycleTime	Time Range	0..1	aggregation	The time the sensor/actuator must be accessed for correct information. It is possible to give a minimum, a maximum and a typical cycle time.
resolution	Float	0..1	aggregation	Defines the granularity of the representation of the Technical Signal in the data format. This applies only if the Technical Signal is encoded before it is transferred to the ECU Electronics (e.g. via Communication Transceiver HW Port).
type	String	1	aggregation	Defines the general type of the sensor/actuator type is a most common naming for a sensor/actuator and is an open list and is not restricted to the following items. Several sets of types exist. Type is mandatory for the usage of the template - Sensor: Temperature, Pressure, Distance, Hall - Actuator: DC Motor, Valve, Relay, Display

Table 9.2: SensorActuatorHW

9.5 I/O Hardware Abstraction

The `I/O Hardware Abstraction` interfaces on one side the MCAL drivers via `Standardized Interfaces` and on the other side the Sensor Actuator Software Component via `AUTOSAR Interfaces`. On the `VFB` the `I/O Hardware Abstraction` is represented by the `EcuAbstractionComponentType`. Depending on the complexity of an ECU, the `I/O Hardware Abstraction` might be sub structured. In this case the `I/O Hardware Abstraction` Layer is described by several different `EcuAbstractionComponentTypes` on M1.

Class	⟨⟨**atpType**⟩⟩ **EcuAbstractionComponentType**
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components

Class Desc.	The ECUAbstraction is a special AtomicSoftwareComponent that sits between a component that wants to access ECUperiphery and the Microcontroller Abstraction. The EcuAbstractionComponentType introduces the possibility to link from the software representation to its hardware description provided by the ECU Resource Template.			
Base Class(es)	AtomicSoftwareComponentType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
bswModuleDescription	BswModuleDescription	*	reference	Reference from the EcuAbstractionComponentType to the Basic Software Module Description describing the BSW part of the ECU Abstraction Component.
hardware Element	HWElement	*	reference	Reference from the EcuAbstractionComponentType to the description of the used HWElements.

Table 9.3: EcuAbstractionComponentType

The `I/O Hardware Abstraction` abstracts from the location of peripheral I/O devices (on-chip or on- board) and the ECU hardware layout and has therefore dependencies to ECU Hardware described by `HWElements`. In addition the `EcuAbstractionComponentType` is hybrid between `Software Component` and `Basic Software Module`. The `BSW` part is described by the means of the `Basic Software Module Template` and the `Basic Software Module Description` is referenced by the `EcuAbstractionComponentType`.

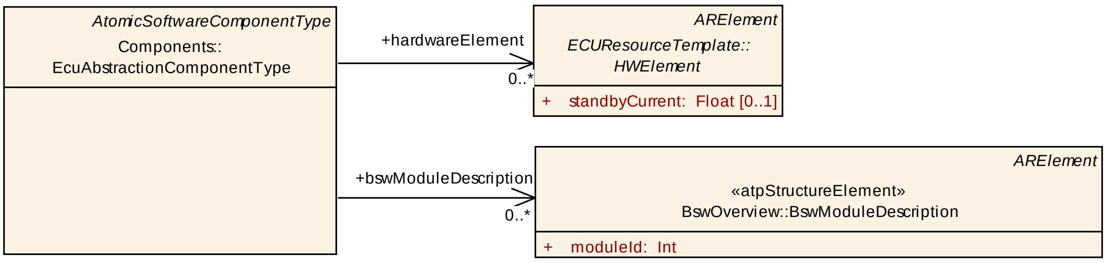


Figure 9.7: `ECUAbstractionComponentType`

9.6 Complex Driver

A `Complex Driver` implements complex sensor evaluation and actuator control with direct access to the Microcontroller using specific interrupts and/or complex Microcontroller peripherals to fulfill the special functional and timing requirements.

In addition it might be used to implement enhanced services / protocols or encapsulates legacy functionality of a `non-AUTOSAR` system. See also document [3].

On the `VFB` the `Complex Driver` is represented by the `ComplexDeviceDriverComponentType`. An ECU might have zero to many different `ComplexDeviceDriverComponentTypes`.

Class	⟨⟨**atpType**⟩⟩ **ComplexDeviceDriverComponentType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Desc.	The ComplexDeviceDriver Component is a special AtomicSoftwareComponent that has direct access to hardware on an ECUand which is therefore linked to a specific ECU or specific hardware. The ComplexDeviceDriver ComponentType introduces the possibility to link from the software representation to its hardware description provided by the ECU Resource Template.			
Base Class(es)	AtomicSoftwareComponentType			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
bswModuleDescription	BswModuleDescription	*	reference	Reference from the ComplexDeviceDriverComponentType to the Basic Software Module Description describing the BSW part of the Complex Device Driver Component.
hardware Element	HWElement	*	reference	Reference from the ComplexDeviceDriverComponentType to the description of the used HWElements.

Table 9.4: ComplexDeviceDriverComponentType

Similar to `EcuAbstractionComponentType` the `ComplexDeviceDriverComponentType` has dependencies to ECU Hardware described by `HWElements` and is a hybrid between `Software Component` and `Basic Software Module`. The `BSW` part is described by the means of the `Basic Software Module Template` and the `Basic Software Module Description` is referenced by the `ComplexDeviceDriverComponentType`.

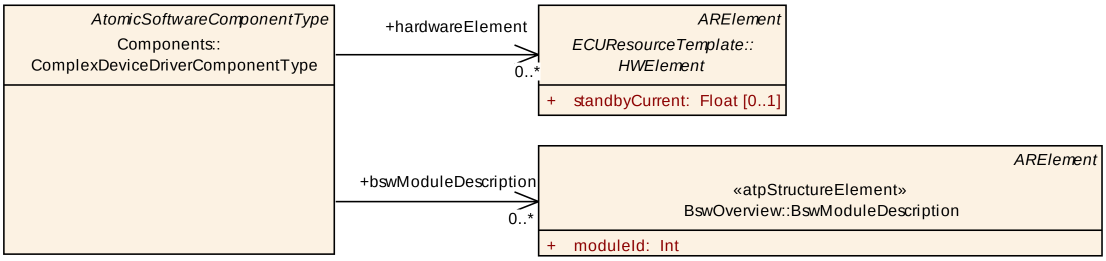


Figure 9.8: `ComplexDeviceDriverComponentType`

10 Services

10.1 Overview: Generation of Service-related Model Elements

This chapter covers the description and handling of `AUTOSAR Service` configuration.

`AUTOSAR Services` can be seen as a hybrid concept between `Basic Software Modules` and a `ComponentType`. `AUTOSAR Services` actually provide access to low-level and ECU-wide ”standard functionalities” commonly referred to as ”service”.

`AtomicSoftwareComponentTypes` requiring services use `Standardized AUTOSAR Interfaces` to communicate with these `AUTOSAR Services`.

Due to that special nature, the handling of such `AUTOSAR Services` requires a number of custom model elements, and also need to be handled specifically in the methodology [4]. The following list of paragraphs presents a short overview over the steps required for the configuration of `AUTOSAR Services`.

Note that most of these steps are performed by tools, and the model elements being created in these steps are rather specific to `Service` configuration and are not to be modeled manually within AUTOSAR authoring tools.

In particular, the following requirements apply:

1. The dependency of an `AtomicSoftwareComponentType` (or more precisely, one of its non-abstract derived meta-classes) from an `AUTOSAR Service` is modeled by aggregating required and provided `PortPrototypes`.

   The `PortInterface` being implemented by the `PortPrototypes` needs to be one of a number of standardized `Service Interfaces`, which is indicated by having its `isService` attribute set to `TRUE` and is referenced by `ServiceNeeds`.

   Additionally, the software components and `Basic Software Modules` shall specify `ServiceNeeds` containing further input information for the later `Service` configuration step.

2. When defining the software system, the `AtomicSoftwareComponentType` is used in the form of `ComponentPrototypes` within a `CompositionType`. In this step, the non-service ports of all required interfaces are being connected using `AssemblyConnectorPrototypes` and `DelegationConnectorPrototypes` in order to eventually form a top-level `SoftwareComposition` which can be referenced in an AUTOSAR `System`.

3. In `System Configuration Phase`, the mapping of all `AtomicSoftwareComponentType` instances to `ECUInstances` is done. The `ServiceNeeds` may be used by tools to check for available resources on the targeted ECUs.

4. The `ECU Extract` is extracted from the `System Configuration` for each ECU. As explained in the `AUTOSAR System Template` [10], this contains an

ECU-centric view onto the system description, including a reduced version of the system's `SoftwareComposition` where `ComponentPrototypes` not being mapped to the ECU are being left out.

5. Early on in ECU Configuration, for each `Service` required on the ECU exactly one `ServiceComponentType` is created based on the needs from the `AtomicSoftwareComponentTypes`: An adequate number of `PortPrototypes` are created on this `ServiceComponentType` for each needed port at the `AtomicSoftwareComponentType`. Thereby the specified communication pattern 1:1 or 1:n for a specific kind of `ServicePort` has to be considered. See also 10.2.2.

6. Per `Service` exactly one `ServiceComponentPrototype` is created based on the previously defined `ServiceComponentType`. Additionally, the connectors are constructed that connect the pairs of `PortPrototypes` belonging to the `ComponentPrototypes` requiring services and those belonging to the actual services.

7. For each `ServiceComponentType` an `InternalBehavior` is created or extended providing the information about `Port Defined Argument Values`, `RunnableEntities` and `RTEEvents` necessary for RTE generation. Further detailing of the service ports by filling in these `Port Defined Argument Values` is also done in ECU Configuration phase. See also chapter 5.5.3.

8. For the RTE module configuration an implementation of the `AUTOSAR Service` belonging to each `ServiceComponentPrototype` and described by a `Basic Software Module Description` has to be selected and the `bswModuleDescription` reference is set accordingly.

   For each `InternalBehavior` created in the previous step one `SwcImplementation` is being created. The information for `SWCImplementation` should be generated based on the available information of `BswImplementation`.

9. In ECU Configuration phase the remaining `Service` parameters are specified. Depending of the configuration of the Service `BSW` it might be necessary to update the `ValueSpecifications` belonging to the `Port Defined Argument Values` generated in a previous step.

Class	⟨⟨**atpObject**⟩⟩ **ServiceNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	This expresses the abstract needs that a Software Component or Basic Software Module has on the configuration of an AUTOSAR Service to which it will be connected. "Abstract needs" means, that the model abstracts from the Configuration Paramaters of the underlying Basic Software.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 10.1: ServiceNeeds

Class	⟨⟨**atpObject**⟩⟩ **EcuInstance**			
Package	M2::AUTOSARTemplates::SystemTemplate::Fibex::FibexCore::CoreTopology			
Class Desc.	ECUInstances are used to define the ECUs used in the topology. The type of the ECU is defined by a reference to an ECU specified with the ECU resource description.			
Base Class(es)	FibexElement			
Attribute	*Datatype*	*Mul.*	*Link Type*	*Description*
associated IPduGroup	IPduGroup	*	reference	With this reference it is possible to identify which IPduGroups are applicable for which CommunicationConnector/ ECU.
comConfigurationId	Integer	0..1	aggregation	This ID is returned by a call to Com_GetConfigurationId()
comProcessing Period	Float	1	aggregation	The COM scheduling time is used in order to be able to calculate the worst case bus timing. The processing period shall be specified AUTOSAR conform in seconds.
commController	Communication Controller	1..*	aggregation	CommunicationControllers of the ECU.
connector	Communication Connector	*	aggregation	All channels controlled by a single controller.
diagnostic Address	Integer	0..1	aggregation	An ECU specific ID for responses of diagnostic routines.
pduRConfigurationId	Integer	0..1	aggregation	unique PDURconfiguration identifier
response Address	Integer	*	aggregation	An ECU specific ID for responses of diagnostic routines.
sleepMode Supported	Boolean	1	aggregation	Specifies whether the ECU instance may be put to a ”low power mode” TRUE: sleep mode is supported FALSE: sleep mode is not supported Note: This flag may only be set to TRUE if the feature is supported by both hardware and basic software.
wakeUp OverBus Supported	Boolean	1	aggregation	Driver support for wakeup over Bus.

Table 10.2: EcuInstance

10.2 Service Related Model Elements in the Software Component Template

This chapter covers meta-model elements exclusively designed for the handling of `AUTOSAR Services`. Note that these model elements are not to be instantiated in the normal context of modeling `SoftwareComponentTypes`, but rather are reserved for the special purpose of `Service` configuration as part of the ECU configuration, a step occurring only after System Configuration phase.

Although these model elements are only added to the `EcuConfiguration` in ECU Configuration phase, they technically belong to the Software-Component Template because they are used for connecting `PortPrototypes` within `CompositionTypes`. However, authoring tools shall not allow for the users to manually create instances of these meta-model classes in software-component descriptions.

10.2.1 ECU Software Composition

As explained in chapter 10.1, Service Configuration takes place in ECU Configuration phase. In doing so, ECU Configuration creates a new model element of type `EcuSwComposition` as shown in figure 10.1 represents the whole Software Composition on an ECU, including both the software components mapped to the ECU by referencing the ECU Extract of the System Description, and the service components by owning one `ServiceComponentPrototype` per `AUTOSAR Service` to be used on the ECU.

Class	⟨⟨**atpPrototype**⟩⟩ **ServiceComponentPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Services			
Class Desc.	Each service in an ECU is represented by exactly one ServiceComponentPrototype. Instances of this class are only to be created in ECU Configuration phase for the specific purpose of the service configuration.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
serviceComponent	ServiceComponentType	1	reference to type	

Table 10.3: ServiceComponentPrototype

Special connectors of type `ServiceConnectorPrototype` are used for connecting service-requiring `PortPrototype` instances of `Application Software Components` with the actual `Service PortPrototype` instances defined in the `ServiceComponentType`.

Class	⟨⟨**atpStructureElement**⟩⟩ **EcuSwComposition**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Services			
Class Desc.	EcuSwComposition contains the complete Software Composition in an ECU, consisting both of application software components and service components.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

component	Service Component Prototype	*	aggregation	Service components used within one EcuSwComposition
connector	Service Connector Prototype	*	aggregation	The connectors used for connecting Service ports with the AtomicSoftwareComponents' service ports.
ecuExtract	System	1	reference	Represents the extract of the System Configuration which the referencing EcuSwComposition applies to, in particular the softwareComposition. As EcuSwComposition is only valid in the context of a given EcuConfiguration, this association needs to have the same target as the ecuExtract association from EcuConfiguration.

Table 10.4: EcuSwComposition

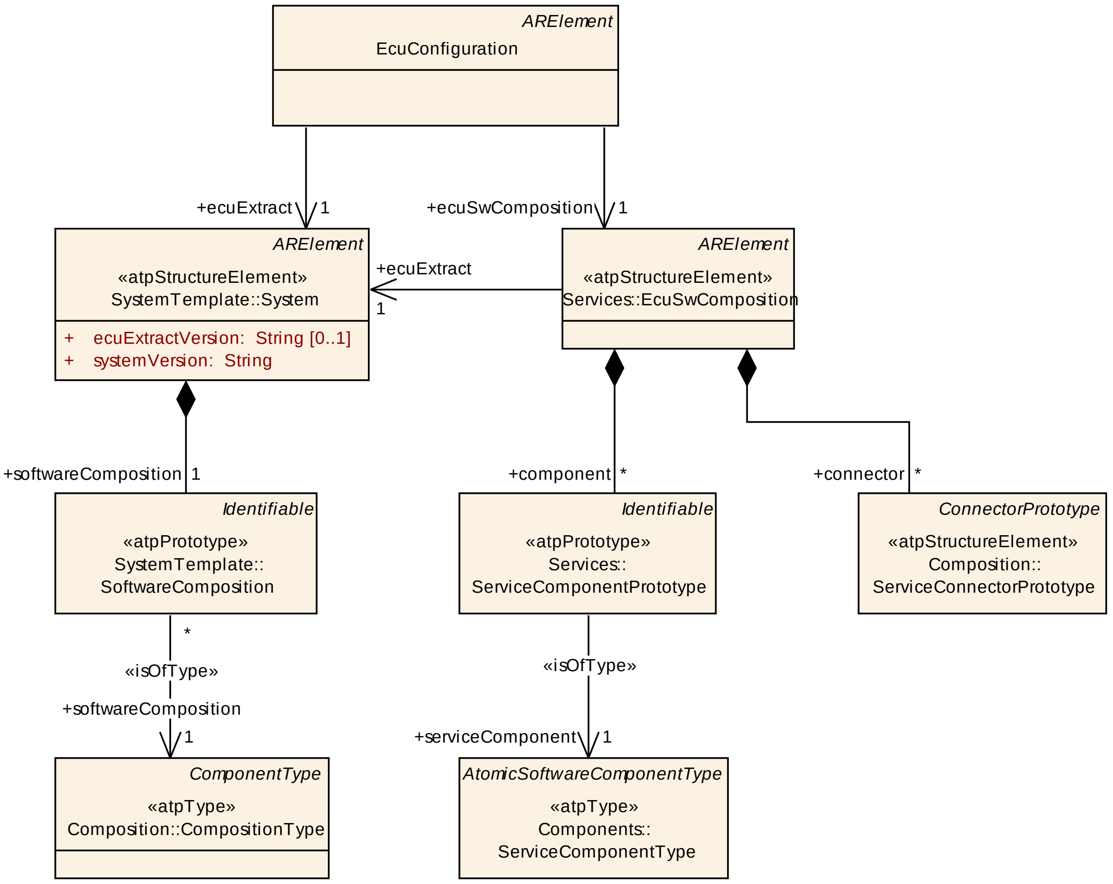


Figure 10.1: EcuSwComposition

10.2.2 Service Component Type

AUTOSAR Services are represented by a meta model class of their own, the ServiceComponentType. As can be seen in Figure 10.2 ServiceComponentType is a specialization of AtomicSoftwareComponentType.

Like any other ComponentType they can aggregate PortPrototypes, in the case of ServiceComponentType all aggregated PortPrototypes need to have an isOfType relationship to a PortInterface which has its isService attribute set to TRUE.

Similar to an EcuAbstractionComponentType and ComplexDeviceDriverComponentType the ServiceComponentType is a hybrid between Software Component and Basic Software Module. The BSW part is described by the means of the Basic Software Module Template and the Basic Software Module Description is referenced by the ServiceComponentType.

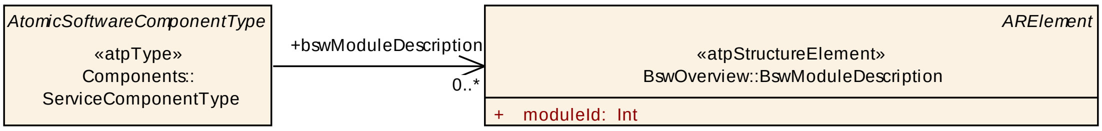


Figure 10.2: **ServiceComponentType**

ServiceComponentType must not be used when modeling application software using CompositionType; they are only added in ECU Configuration phase, where exactly one ServiceComponentPrototype per ServiceComponentType per ECU is added to the ECU Description model.

The Base ECU Config Generator tool needs to take care that for all service ports of ComponentPrototypes mapped to the ECU service ports at the appropriate ServiceComponentTypes are created. In the process the specified communication pattern 1:1 or 1:n for a specific kind of service port has to be considered.

In case of 1:1 communication for each service port of a ComponentPrototype one port on the ServiceComponentType is created.

In case of 1:n communication for each different type of service port one port on the ServiceComponentType is created.

Class	**⟨⟨atpPrototype⟩⟩ ServiceComponentPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Services			
Class Desc.	Each service in an ECU is represented by exactly one ServiceComponentPrototype. Instances of this class are only to be created in ECU Configuration phase for the specific purpose of the service configuration.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

service Component	ServiceComponentType	1	reference to type	

Table 10.5: ServiceComponentPrototype

More explicitly, all instances of `AtomicSoftwareComponentType` need to be checked for `PortPrototypes` of `PortInterfaces` with `isService` attribute set to `TRUE` and referenced by `ServiceNeeds`, and for each of these `PortInterface` instances belonging to the `AUTOSAR Service` to be configured one `PortPrototype` implementing the same or a compatible `PortInterface` needs to be created on the `ServiceComponentType`.

The roles of the `PortPrototypes` (required/provided) on the Application Component and the Service Component side obviously need to match, i.e. an `RPortPrototype` attached to an application `AtomicSoftwareComponentType` matches a `PPortPrototype` attached to a `ServiceComponentType`.

10.2.3 Service Connector Prototype

The `ServiceConnectorPrototype` (see figure 10.3) is exclusively used in ECU Configuration Phase for connecting software components requiring `AUTOSAR Services` to the `Services` they are requiring on. More detailed this means that for each instance of an `AtomicSoftwareComponentType` containing a `PortPrototype` that declares via its `PortInterface` that it needs to be connected to an AUTOSAR `Service` the `PortPrototype` needs to be connected to the respective `PortPrototype` on the `ServiceComponentType`.

Class	⟨⟨**atpStructureElement**⟩⟩ **ServiceConnectorPrototype**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Class Desc.	A ServiceConnectorPrototype connects a PortPrototype owned by an ComponentPrototype with the service PortPrototype owned by the ServiceComponentPrototype. A ServiceConnectorPrototype is only added to the model in ECU Configuration phase for the specific purpose of configuring services within an EcuSwComposition.			
Base Class(es)	ConnectorPrototype			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
application Port	PortPrototype	1	instanceRef	Service port to be connected on application component side
service Port	PortPrototype	1	instanceRef	Service port to be connected on service component side

Table 10.6: ServiceConnectorPrototype

Class	⟨⟨**atpType**⟩⟩ **ServiceComponentType**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Desc.	ServiceComponentType is used for configuring services for a given ECU. Instances of this class are only to be created in ECU Configuration phase for the specific purpose of the service configuration.			
Base Class(es)	AtomicSoftwareComponentType			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
bswModuleDescription	BswModuleDescription	*	reference	Reference from the ServiceComponentType to the Basic Software Module Description describing the BSW part of the Service Component.

Table 10.7: ServiceComponentType

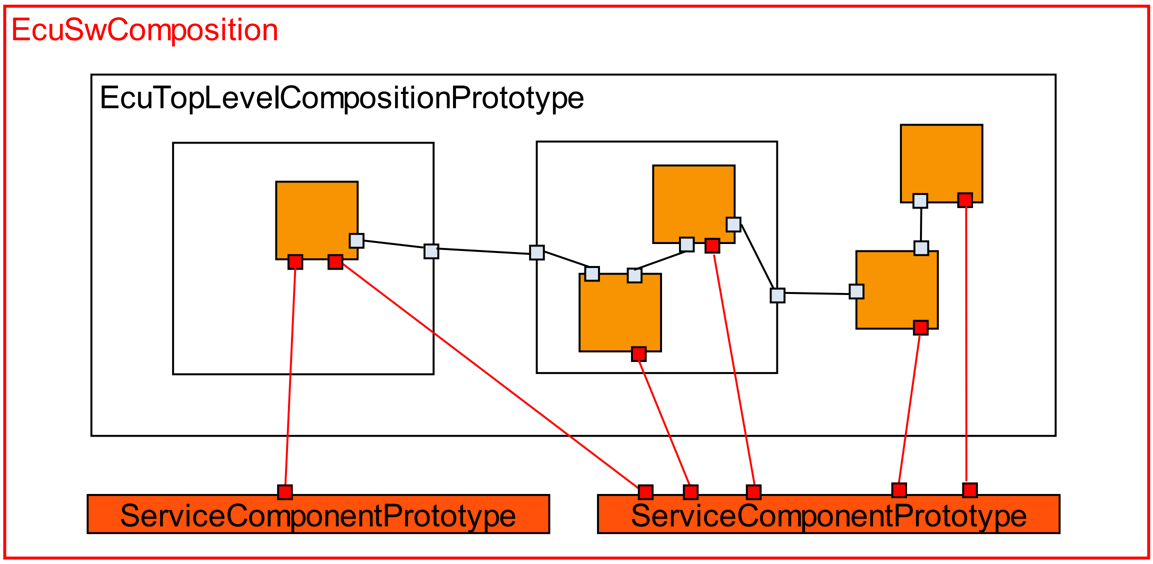


Figure 10.3: ServiceConnectorPrototypes connecting Application Component Service Ports to Service-ComponentPrototype Service Ports

Compared to the other connector types the `ServiceConnectorPrototype` is different in the way that the two `PortPrototypes` it connects have different contexts: On the one hand side a `PortPrototype` aggregated by an `AtomicSoftwareComponentType` can have an unlimited number of nested `ComponentPrototypes` forming a Composition hierarchy in the ECU Extract Software Composition.

On the other hand, the `ComponentPrototypes` representing the `ServiceComponentTypes` are flatly aggregated by the `EcuSwComposition`. A further constraint is that both connector ends need to connect `PortPrototypes` belonging to the same or compatible `PortInterface` which must have its `isService` attribute set to `TRUE`.

Please find an overview of `ServiceConnectorPrototype` in figure 2.6.