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

Document Version	2.2.0
Document Status	Draft
Part of Release	2.1
Revision	20

Document Change History

Date	Version	Changed by	Change Description
31.05.2010	2.2.0	AUTOSAR Administration	• Add missing InstanceRef from ModeSwitchComSpec to ModeDeclarationGropuPrototype • Legal disclaimer revised
31.01.2007	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 towards 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” revised • “Revision Information” added
18.05.2006.	2.0.0	AUTOSAR Administration	Second
09.05.2005	1.0.0	AUTOSAR Administration	Initial release

Release Notes

Errata and known deficiencies

All modifications planned in the scope of Release 2.1 for the incorporation into this document are completed. The document, however, has not yet undergone the necessary finalization.

Disclaimer

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

1 Introduction

1.1 Methodology for Defining Formal Template

Figure 1 illustrates the overall methodology used to define formal templates. As is explained in the "*Template UML Profile and Modeling Guide*", it is important to separate a precise and concise model of the information that needs to be captured from the concrete XML-DTDs, XML-Schemas or other technology that is used to define the actual templates.

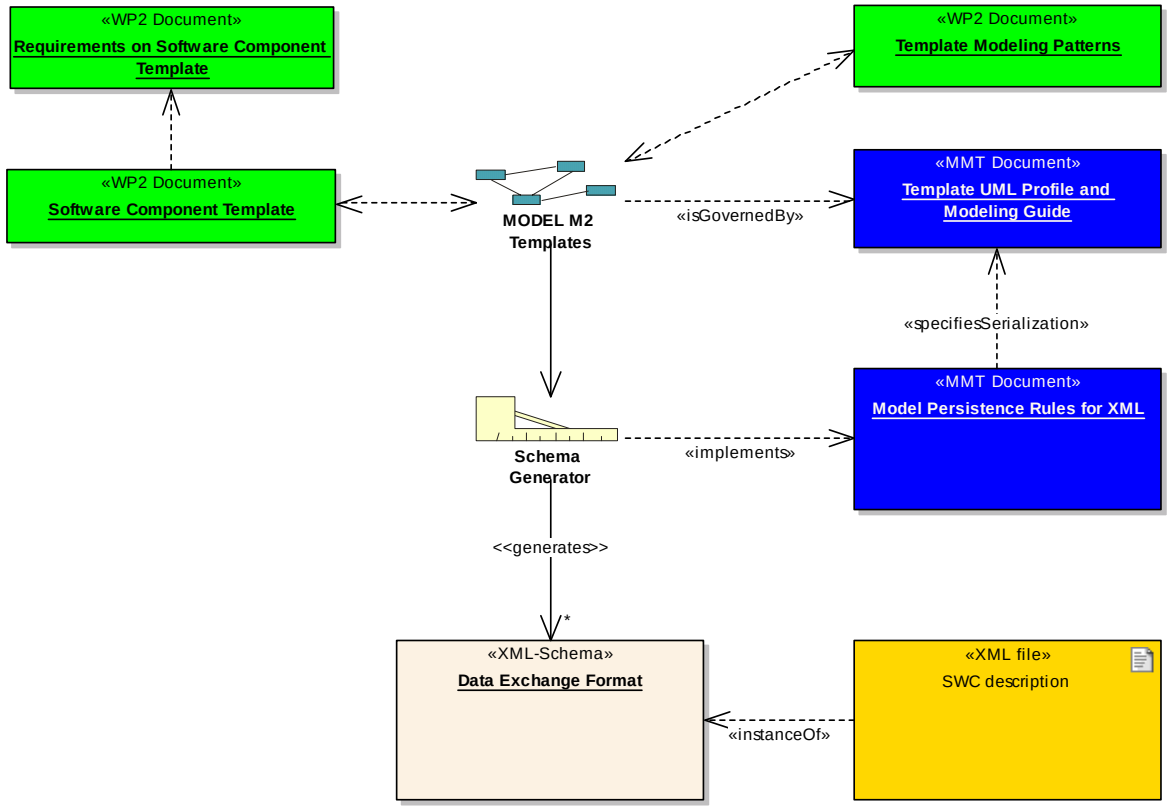

Figure 1: Methodology to define templates in AUTOSAR

The following documents describe the various aspects of the methodology:

1. The document called "*Software Component Template*" (this document) describes the information that can be captured in the "software-component" descriptions, independently from the mapping of this model on XML-technology. This document consists of the metamodel or "model of attributes" and an elaborate description of the semantics (the precise meaning) of all the information that can be captured with this metamodel
2. The document called "*Template UML Profile and Modeling Guide*" describes the basic concepts that should be used when modeling the information in the metamodel.

3. The document called "*Model Persistence Rules for XML*" describes how XML is used and how the metamodel 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 to be used for all data that is formally described in the metamodel. 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 "*Template Modeling Patterns*" are represented as predefined Classes in the metamodel 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 metamodel.
5. The concrete "Template" the "*Data Exchange Format*" is an XML schema which is generated out of the metamodel 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 tools. The M1 – level software component descriptions are XML files which can be validated against the schema. In that sense they are instances of the schema defining the XML representation of the template.

In Figure 2 the relationsship between the AUTOSAR templates and their associated template specification documents is illustrated.

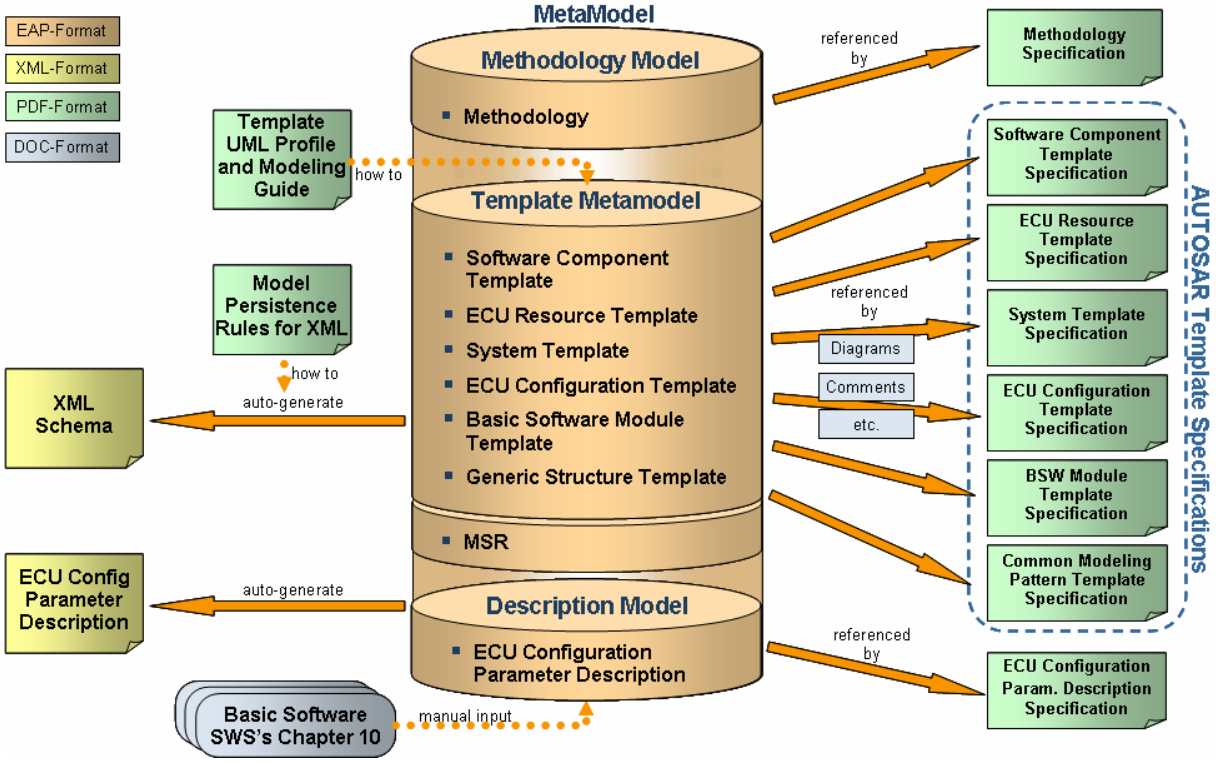

Figure 2: Structure and Dependencies of AUTOSAR Templates

1.2 Scope

The Scope of this document is the description of "components".

This work covers the following three aspects:

a. A general description of components using ports and interfaces
b. A description of "compositions", which are sub-systems consisting out of connected instances of components
c. A description of "atomic software-components" which is implemented as a piece of software that can be mapped to an AUTOSAR ECU

These aspects are described in more details in the next sub sections.

1.2.1 Scope 1: General Description of Components

This document defines the "component" as an entity which can be described through "ports" which provide or require "interfaces". This component concept can not only be used to describe an "atomic software-component" (a piece of software that can be mapped to an AUTOSAR-ECU) but also to describe "non AUTOSAR components".

1.2.2 Scope 2: Description of Compositions

Software components can exist in the form of hierarchical subsystems, which in turn consist of components again. The description of such hierarchical structures is in scope of this document.

1.2.3 Scope 3: Description of Atomic Software-Components

A special kind of component is the "atomic software-component", which is implemented as a piece of software that can be mapped to an AUTOSAR ECU.
An "atomic software-component" component therefore shows up in the ECU Software Architecture, shown in Figure 3. In this figure, the green (vertically striped) and blue (diagonally striped) borders show the aspects that are described by the Software-Component Template
This document describes the "AUTOSAR software-components" (which are the atomic software-components which lie above the RTE) completely. This includes a complete description of their resource needs.
This document will however not completely cover the description of the atomic software-components, which are not "AUTOSAR software-components". For example, a "complex device driver" is an atomic software-component which has direct interaction with local hardware. A description of this interaction (and the associated resources, like interrupts, that are being used) is not in the scope of this document.

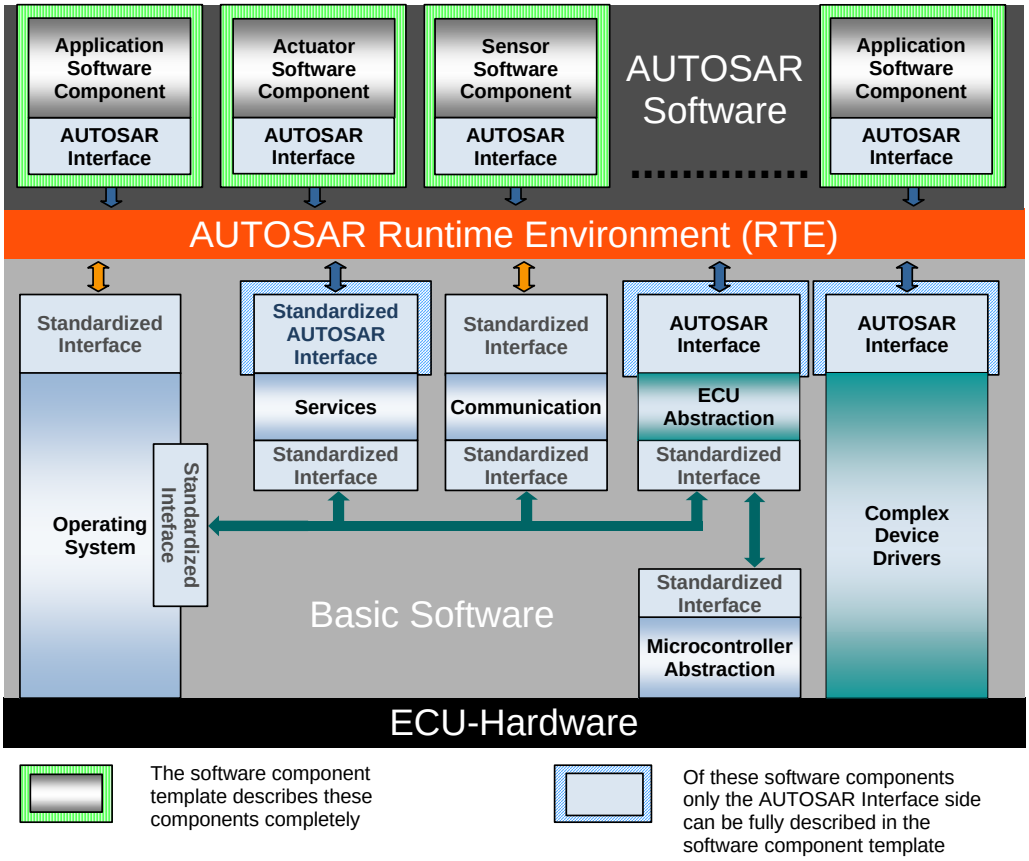

Figure 3: Scope of this document in the ECU SW Architecture

1.3 Organization of Meta-Model

Figure 4 shows the overall structure of the meta model, which formally defines the vocabulary required to describe AUTOSAR software components. As the diagram points out, also other work packages (ECU Resource and System Constraint) use the same modeling technique in order to find an overall consistent model of description attributes. The arrows in the diagram describe the relationships between the packages within the meta-model. For example the “SWComponentTemplate” refers to concepts defined in the packages “GenericStructure” and “ECUResourceTemplate”.
This specification however will only explain the elements defined in the packages “SWComponentTemplate” and “GenericStructure”. The latter package contains some fundamental infrastructure classes and common patterns, while the component package consists of all the software component specific attribute definitions.

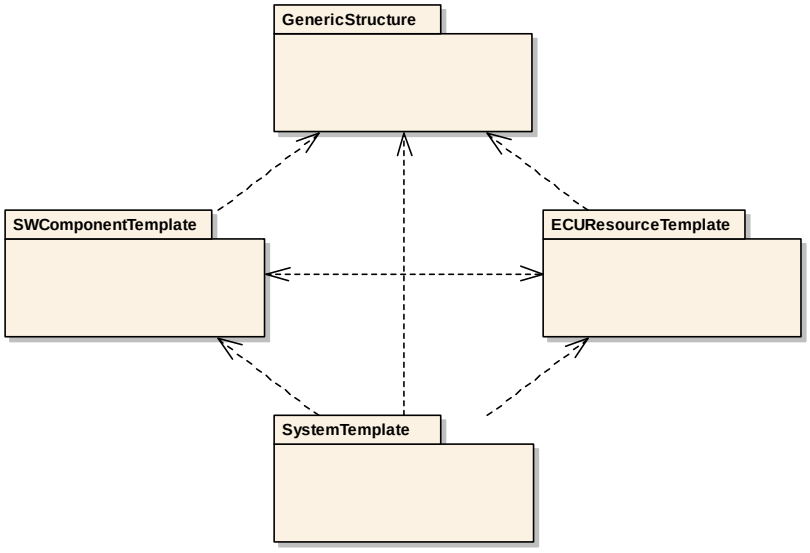

Figure 4: Packages in the full meta model.

The content of the other packages is out of scope of this document and is described in those templates' respective specifications.

1.4 Structure of the Template

AUTOSAR software components are described on three distinctive levels, as shown in Figure 5.

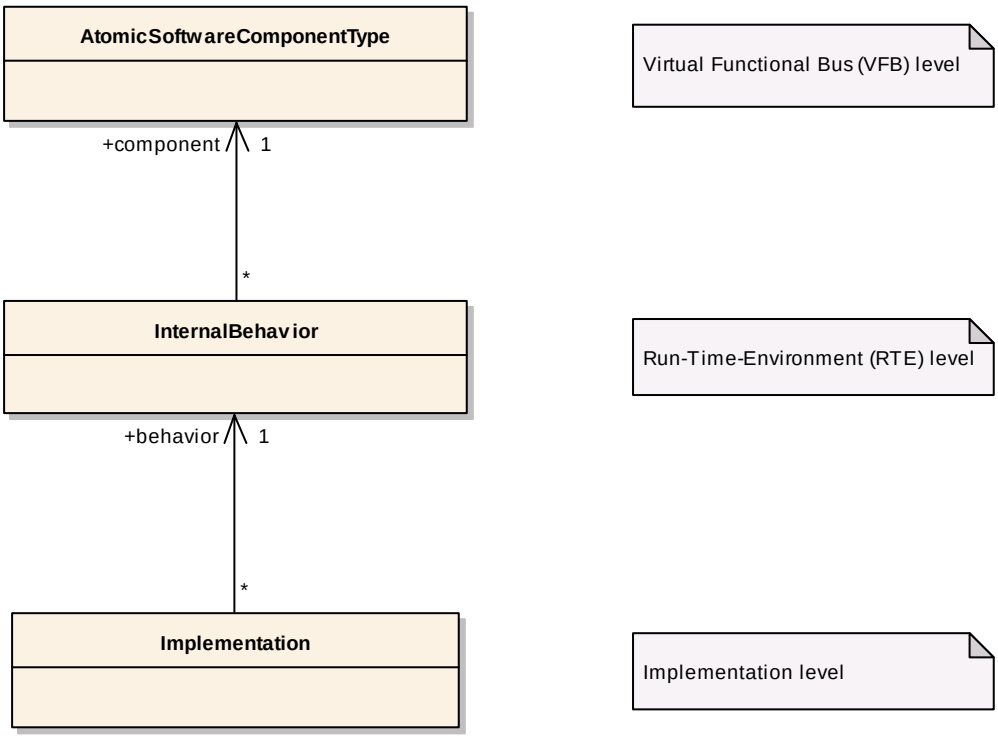

Figure 5: The description of a software component is done on three levels.

Description of components on VFB level
The highest (most abstract) description level is the Virtual Functional Bus. Here components are described with the means of datatypes and interfaces, ports and connections between them, as well as hierarchical components.
At this level, the fundamental communication properties of components and their communication relationships among each other are expressed. In the diagram above, this is expressed through the description of atomic software components[1].
The following chapters address this level:

- 2 Software Components
- 3 Compositions
- 4 Interfaces
- 5 Modes
- 6 Measurement & Calibration

Description of components on RTE level
The middle level allows for behavior description of a given software component. This behavior is expressed through the means of AUTOSAR RTE concepts, e.g. RTE events and in terms of schedulable units. For instance, for an operation defined in an interface on the VFB, the behavior specifies which of those units is activated as a consequence of the invocation of that operation.
Theoretically, there may be multiple behaviors for a given software component specification.
The following chapters address this level:

- 5 Modes
- 7 Interaction with the Run Time Environment

Descriptions of components on implementation level
The lowest (most concrete) level of description specifies the implementation of a given behavior description. More precisely, the schedulable units of such a behavior are mapped to code (source code or object code). While the two layers above constrain the RTE API that a component is offered, the implementation now utilizes this API.
There may be different implementations per behavior description, e.g. in different programming languages, or with differently optimized code.
Finally, the following chapters describe this third specification level:

- 8 Component Implementation
- 9 Resource Consumption of AUTOSAR SW-Components

1.5 Relation of AUTOSAR Templates to the MSR DTDs and ASAM

The MSR Consortium[2] has developed several XML-DTDs which define structures for the exchange of data in the domain of embedded electronic systems (in particular in the automotive). Due to the experience already present in these structures it was decided to take over major concepts (described in the MSR-TR-CAP document [13]).

[1] To avoid clutter and require additional up-front information about the meta model, compositions have not been added to the diagram.
[2] Manufacturer Supplier Relationship, a consortium that specified how to exchange engineering artifacts and how to describe them. See: http://www.msr-wg.de.

In the meantime MSR is continued by ASAM. One part is specified in “Meta Data Exchange Format for Software Module Sharing (MDX)”.
Also ASAM had harmonized common parts of MSR and ODX. These agreed parts are called “ASAM 2 object Harmonized Data MCD”.
For the software component template also some parts of the MSR-SW DTD were identified to be included. In order to achieve this, the MSR software DTD was reverse engineered according to “*Template Formalization Guide*” such that it could be incorporated in the AUTOSAR meta-model seamlessly. So the AUTOSAR template is a mixture of newly generated parts and take-over parts from the MSR-SW.

1.6 Requirements Tracing

The following table contains the requirements that are listed in the document “*Requirements on Software Component Template*” and the references to the sections in this current document that contain the corresponding descriptions.

Requirement	Satisfied by
[AR_CONTENT_0010] Compositions	3 Compositions
[AR_CONTENT_0020] Interfaces	4 Interfaces
[AR_CONTENT_0030] Libraries	8.7 Dependencies
[AR_CONTENT_0035] Integration on object code level	Not yet covered.
[AR_CONTENT_0040] Behavior	Not formally supported.
[AR_CONTENT_0050] Schedulability	7 Interaction with the Run Time Environment
[AR_CONTENT_0060] Ordering	Not formally supported.
[AR_CONTENT_0070] Resource	9.1 SW-Component Resources
[AR_CONTENT_0080] Timing	7 Interaction with the Run Time Environment Reaction time is not covered.
[AR_CONTENT_0090] Sensors / Actuators	10.4 Shipment of Sensors/Actuators
[AR_CONTENT_0100] Variant	Not yet covered.
[AR_CONTENT_0110] Modes	5 Modes
[AR_CONTENT_0120] Dependency on Modes	5 Modes

2 Software Components

2.1 Introduction

In AUTOSAR, application software is organized in independent units, called *software components*. Such components hide the implementation of the functionality and behavior they provide and simply expose very well defined connection points, called ports.
The software components of an AUTOSAR system do depend on each other with respect to utilizing each other's functionalities. However, those kind of dependencies are described precisely in form of interfaces and ports, and no internal, hidden dependencies may exist. Therefore, components are in theory exchangeable as long as they implement the same logic and provide the same public communication interface to the remaining system.
Once a component is defined with the help of the software component template, a new component type has been defined. This implies that such a component can be used an arbitrary number of times within the same system as well as in different systems. Here, *use* refers to the assignment of a certain role to a component type (e.g. a windshield wiper component is assigned the *left wiper* role, another one the *right wiper*).
Finally, components are developed against the virtual functional bus, an abstract communication channel without direct dependency on ECUs and communication busses. This has consequences on the design of components, which must not directly call the operating system or the communication hardware. As a result, components are transferable and can be deployed to ECUs very late in the development process.
This chapter explains the fundamental concepts around AUTOSAR software components.

2.2 Components, Ports and Interfaces

As mentioned above, AUTOSAR software components have well defined interaction points to describe the possible kinds of (data as well as service oriented) communication with other software components, called the ports of the component. Figure 6 shows those fundamental classes in the meta model of the template.

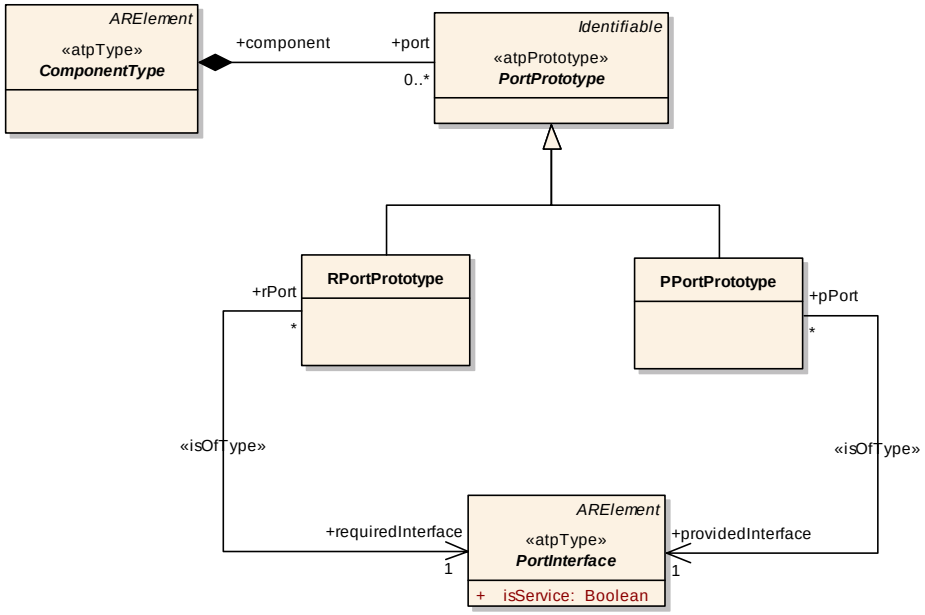

Figure 6: Metamodel of AUTOSAR Software Components.

A “ComponentType” is a class that *describes* a *component type* that will be assigned a role and hence instantiated at a later time. Such components have an arbitrary number of ports. Ports are described in the class “PortPrototype”.
Different kinds of component types are possible. The most direct type is the “Atomic-SoftwareComponentType”. Only atomic software components can eventually be assigned an actual implementation in a programming language.
The model shows that it is allowed to define a component without any ports, which seems counterintuitive, since what could this component possibly do?
In fact, this is only reasonable for components that are totally self-contained and don’t need any interaction with other components. Since components allow for hierarchical structure (see below), this may be the case for the top level component, that describes the complete logical software system of the vehicle. If basic software entities are not part of that system, then even the top level composition will have ports, e.g. for communication with the ECU abstraction.
Ports are either *require*- or *provide*-ports. A require-port (short: r-port) requires certain services or data, while a provide-port (p-port) on the other hand provides those services or data.
Two components are eventually connected by hooking up a p-port of one component to a compatible r-port of the other component, as shown in Figure 7.

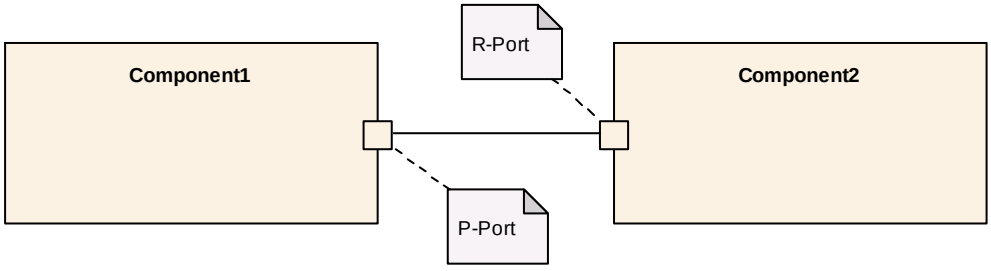

Figure 7: Two components connect via compatible r- and p-port.

Whether ports are actually compatible is described by a “PortInterface”. As will be shown later, these port interfaces declare services or data elements that are required and provided by the respective ports.
A port interface has a single direct attribute (opposed to further subclasses and aggregations shown elsewhere), called “isService”. This flag indicates, whether service or data described in the interface is actually provided by AUTOSAR services instead of another AUTOSAR software component[3].
The visual representation of components, ports and interfaces in a component model is shown in Figure 8. The figure has been taken from the specification of the VFB.

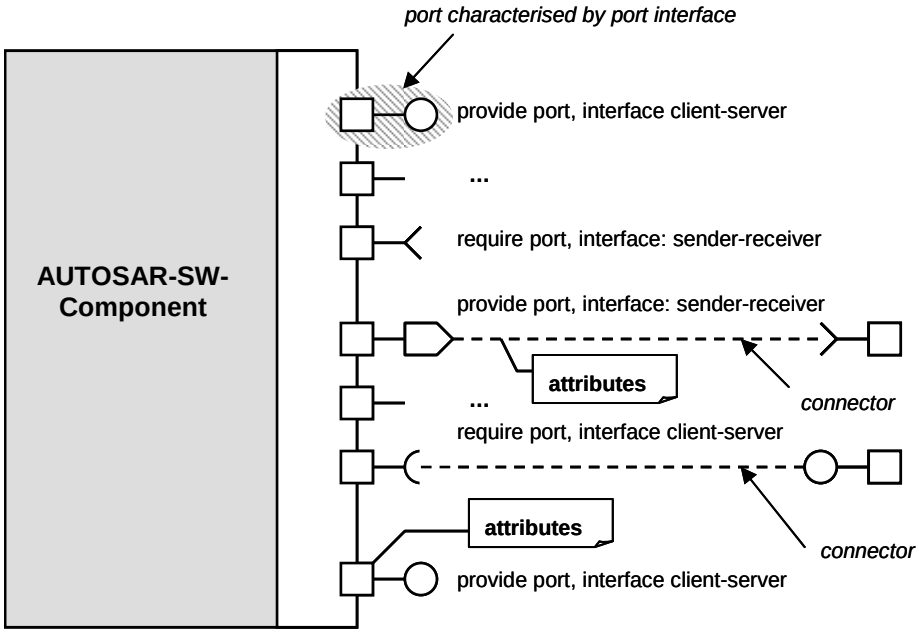

Figure 8: Graphical representation of components, ports and interfaces.

The following tables list the formal parameters of components, ports and interfaces in the AUTOSAR component description meta model.

Class	ComponentType {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Description	Base class for AUTOSAR software components.			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement			
Attribute	Datatype	Mul.	Link Type	Attribute Description
port	PortPrototype	0..*	aggregation	The ports through which this component can communicate.

Class	AtomicSoftwareComponentType
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components

[3] Sometimes these kind of interfaces are also called “vertical interfaces”, due to the visual fact that a component sits on top of the RTE, and hence connects vertically to RTE provided services.

Class Description	An atomic software component is atomic in the sense that it cannot be further decomposed and distributed across multiple ECUs.			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, ComponentType			
Attribute	Datatype	Mul.	Link Type	Attribute Description
measurable	Measurable	0..*	aggregation	The measurable is used to allow the delivery of component types with CalprmElementPrototype entities referencing DataPrototype entities in PortPrototype entities as input values. Furthermore, this aggregation determines the actual PortPrototype of the DataPrototype in an InterfaceType.

Class	PortPrototype {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Description	Base class for the ports of an AUTOSAR software component.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
annotation	PortAnnotation	0..*	aggregation	Formal annotations of the port that have no direct influence on communication and RTE generation.

Class	RPortPrototype			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Description	Component port requiring a certain port interface.			
Base Class(es)	Identifiable, PortPrototype			
Attribute	Datatype	Mul.	Link Type	Attribute Description
requiredComSpec	RPortComSpec	0..*	aggregation	Required communication attributes, one for each interface element.
requiredInterface	PortInterface	1	reference to type	The interface that this port requires, i.e. the port depends on another port providing the specified interface.

Class	PPortPrototype			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Description	Component port providing a certain port interface.			
Base Class(es)	Identifiable, PortPrototype			
Attribute	Datatype	Mul.	Link Type	Attribute Description
providedComSpec	PPortComSpec	0..*	aggregation	Provided communication attributes per interface element (data element or operation).
providedInterface	PortInterface	1	reference to type	The interface that this port provides.

Class	PortInterface {abstract}

Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Description	Abstract base class for an interface that is either provided or required by a ports of a software component.			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement			
Attribute	Datatype	Mul.	Link Type	Attribute Description
isService	Boolean	1	aggregation	This flag is set, if the port interface is to be used for comunication between an atomic software component and a component of the basic software (namely an AUTOSAR Service, ECU abstraction or Complex Driver) located on the same ECU. Otherwise the flag is not set.

2.3 Communication and Application Level Attributes

AUTOSAR software components communicate via the Virtual Functional Bus. They need ways to express requirements and capabilities with respect to exchanging data, which is currently possible through two kinds of attributes:

(a) Communication attributes, detailed in 7.4, allow specifying parameters of the communication that affect the generation of the RTE or the actual communication taking place at runtime. An example for such an attribute is the aforementioned transfer time over a connector.

(b) Application level attributes, allow describing properties of exchanged data that do *not* affect the RTE generation, but are indicate to the developer how data needs to be processed. An example for this kind of attribute is a flag, whether data is “filtered” or “raw”. More detailed information on this type can be found in chapter 4.6.

2.4 Multiple Instantiation of Software Components

Instantiation in this context refers to the creation of M0 objects (aka instances) on the basis of "AtomicSoftwareComponent" on M1 modeling level. Please note that the creation of instances of "AtomicSoftwareComponent" depends on the selection of a particular "InternalBehavior" referencing "AtomicSoftwareComponent" and the selection of a particular "Implementation" referencing "InternalBehavior".
Following the cardinalities of the reference chain it is essentially the combination of "Implementation" and "AtomicSoftwareComponent" that determines the basis for instantiation. In other words: instances are derived from a particular combination of "AtomicSoftwareComponent" and "Implementation". The latter shall be referred to by the term *component implementation*.
The process of creating more than a single instance on the basis of a particular combination of "AtomicSoftwareComponent" and "Implementation" is called multiple instantiation.
Please note that the existence of multiple instances of such a combination breeds special traits that have to be observed carefully. Chapter 8.8 gives an overview of all aspects that must be taken into account to understand the issue of multiple instantiation.

3 Compositions

3.1 Overview

The purpose of an AUTOSAR composition is to allow encapsulation of functionality by aggregating existing software components. Since a composition is also a kind of component, it again may be aggregated in even further compositions. This recursive relation is shown in the following extract from the AUTOSAR meta model.

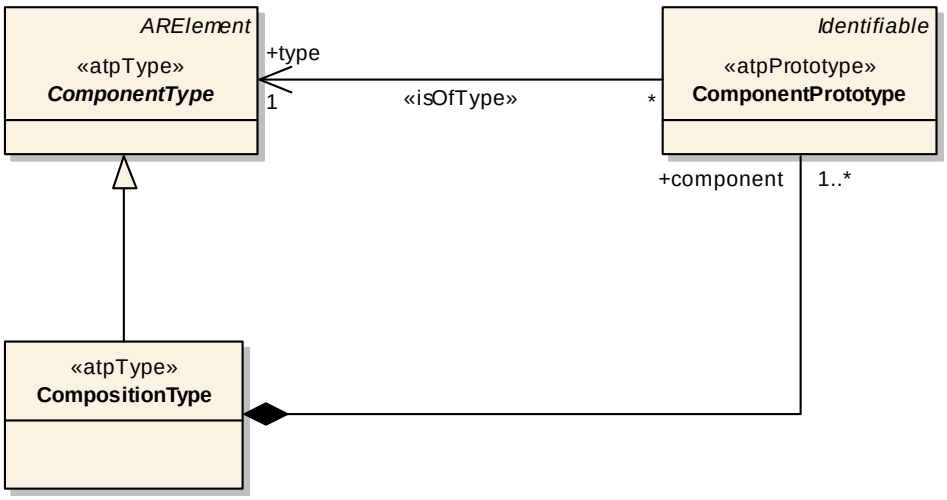

Figure 9: The recursive relation of software components and compositions.

It is important to note that while compositions allow for (sub-) system abstraction, they are solely an architectural element supporting model scalability. They simply group existing components and thereby take away complexity when viewing or designing logical system architecture. More information on the concept of compositions is given in [3].
Therefore, compositions have no effect on how components interact with the Virtual Functional Bus (VFB). Compositions do not add any new functionality to what is already provided by the components they aggregate – instead they build a so-called order-hierarchy, i.e. they simply add structure. Furthermore, compositions have no binary representation (like e.g. a “mini-RTE”), i.e. there is no component hierarchy in the final runtime system.

3.2 Structure of a Composition

As outlined above, a composition is an aggregating component type. In short, compositions contain a number of sub component prototypes, the connections between their ports, and the knowledge, which ports are explicitly delegated and thereby made publicly known to the outside of the composition. This is shown in the following class diagram.

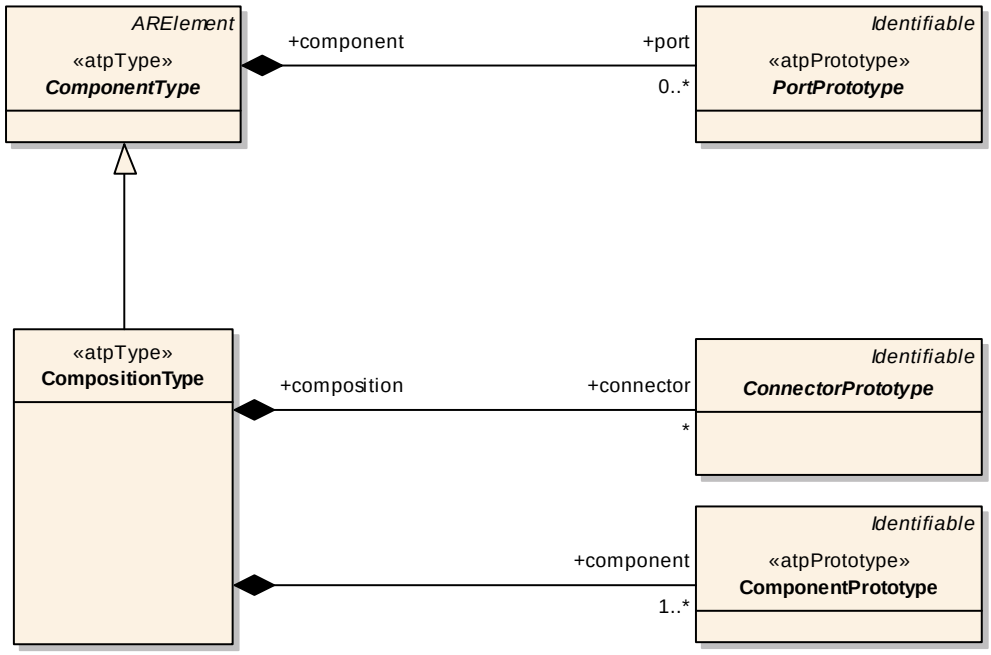

Figure 10: Composition and the classes it aggregates.

It has to be clearly understood that defining a composition will define a new component type, which is not instantiated by itself. However, the components that are part of a composition are assigned roles within the component's context. Refer to the modeling guide for precise definition of type, role and instance.
Here are the attributes of meta class "CompositionType":

Class	CompositionType			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Class Description	Aggregating component type to encapsulate and abstract subsystem functionality. Compositions contain instances of the component parts, as well as the connections between them.			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, ComponentType			
Attribute	Datatype	Mul.	Link Type	Attribute Description
component	ComponentPrototype	1..*	aggregation	The instantiated components that are part of this composition.
connector	ConnectorPrototype	0..*	aggregation	The connectors that link component instances within composition.

3.2.1 Component Prototypes

First and foremost, compositions accumulate and thereby group prototypes of component types. For completion, here is the explicit attribute once more:

Class	ComponentPrototype
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition

Class Description	Role of a software component within a composition.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
type	ComponentType	1	reference to type	Type of the instance.

3.2.2 Ports

A composition exposes inner component's ports explicitly by means of delegated ports. Only ports that are not yet connected can be delegated this way. Also, delegation is an explicit decision whether or not an inner port is actually available from the outside of the composition.
The semantics of this concept are similar to encapsulation mechanisms like public and private members in object-oriented programming languages.

3.2.3 Connectors

Compositions contain two kinds of connectors: assembly connectors to interconnect ports of components that are part of the composition as well as delegation connectors from inner ports to delegated ports. A third kind of connector, the service connector is exclusively used for in the context of ECU configuration phase, and must not be used within "CompositionTypes" of software applications. The different kinds of connectors are shown in the following diagram.

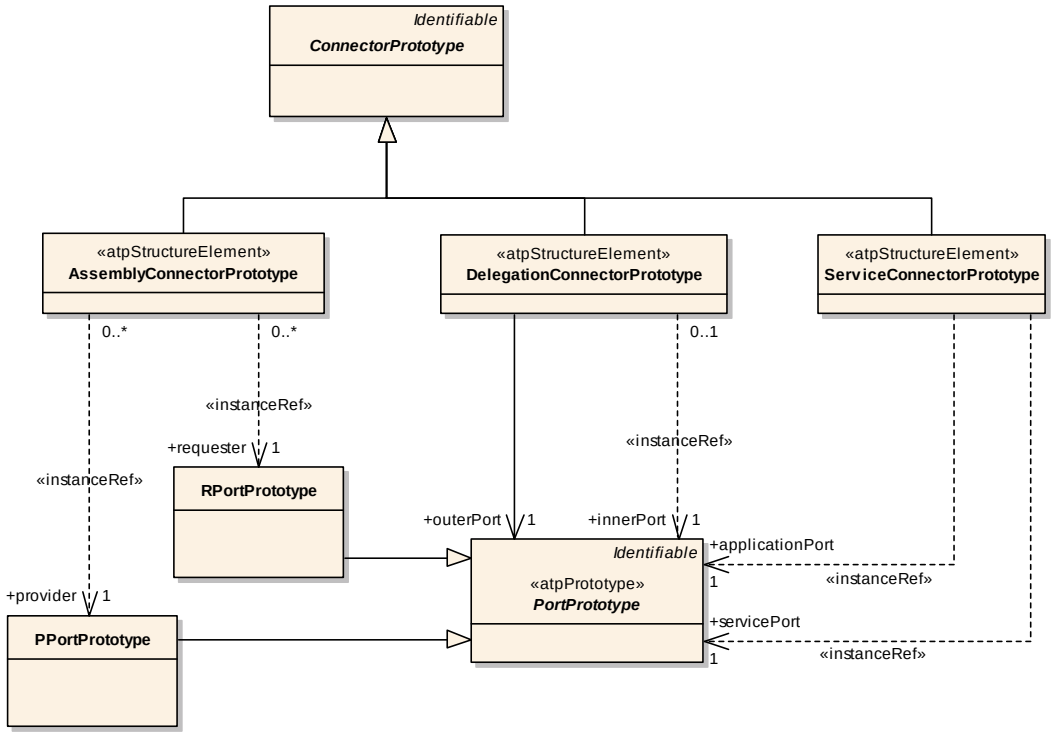

Figure 11: Connector types and the ports they reference.

3.2.3.1 Assembly Connector

An assembly connector is required to connect one or more r-ports to one or more compatible p-ports. All of those ports are part of component instances within the composition.

Class	AssemblyConnectorPrototype			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Class Description	An assembly connector connects a p-port to an r-port. Unlike a delegation connector, assembly connectors are "real" in the sense that there will be some kind of physical representation (bus, shared memory, ...) they are mapped to.			
Base Class(es)	Identifiable, ConnectorPrototype			
Attribute	Datatype	Mul.	Link Type	Attribute Description
connectorComSpec	ConnectorComSpec	0..*	aggregation	Communication attributes of connectors, one per element in the connector's common interface (the interface that p- and r-port agreed upon).
provider	AssemblyConnectorPrototype_-provider	1	reference to instance	Instance of providing port.
requester	AssemblyConnectorPrototype_-requester	1	reference to instance	Instance of requiring port.

The communication related attributes are once more explained in the RTE section.

Many to Many Connectors
While already explained in the VFB specification the allowed types of many to many connections are listed here once more for sake of clarity. A single assembly connector has exactly one R- and also exactly one P-Port. However, those ports connect to multiple connectors at the same time. Here is a list of the scenarios allowed by the VFB communication model:

1. Many clients to one server.
2. Many receivers to one sender.
3. One receiver to many senders.

Scenario 3 seems somewhat obscure but is required for services like error management. For more information, refer to the VFB specification.

3.2.3.2 Delegation Connector

A delegation connector is used to connect a single port of an inner component prototype to one of the composition's ports. It does not have any quality attributes since the connection is purely logical.

Class	DelegationConnectorPrototype
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition
Class Description	A delegation connector simply delegates one inner port (a port of a component that is used inside the composition) to a port of identical type that belongs directly to the composition (a port that is owned by the composition). Just as compositions they are solely for structuring the model and have no physical meanig. In particular, delegation ports cannot merge or dispatch data.

Base Class(es)	Identifiable, ConnectorPrototype			
Attribute	Datatype	Mul.	Link Type	Attribute Description
innerPort	DelegationConnector-Prototype_innerPort	1	reference to instance	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

3.2.3.3 Service Connector

The “ServiceConnectorPrototype” is exclusively used in ECU Configuration Phase in order to connect application components requiring Service ports to their counterparts, the provided service ports. In particular, when modeling ordinary “CompositionType” model elements it is not allowed to use this kind of connector, and tools shall forbid the user to create them manually. See 11.2.3 in chapter AUTOSAR Services for details.

3.3 Top Level Compositions

One purpose of using compositions is to aggregate functionality at higher abstraction levels. If this is done over and over again, one will end up with a single top-most composition which contains the functionality of the full system.
This top-level composition effectively describes the complete *software component architecture* of this system and would typically be called something in analogy to “Car”, “System”, etc….
Note, that this does not necessarily mean that such a top-level composition does not have any ports: there may still be open ends, e.g. due to connection to the ECU abstraction.

4 Interfaces

4.1 Overview

4.1.1 Purpose of Interfaces

A Port interface defines, which information is exchanged between software components, or to be more precise between ports of those components. It does this by formally describing the names and signatures of operations and data elements exchanged between the two components.
Interfaces are used to support *design-by-contract*, i.e. they provide means to formally verify structural and dynamic compatibility between today's as well as tomorrow's components. As such, interfaces are a pivotal point in AUTOSAR.

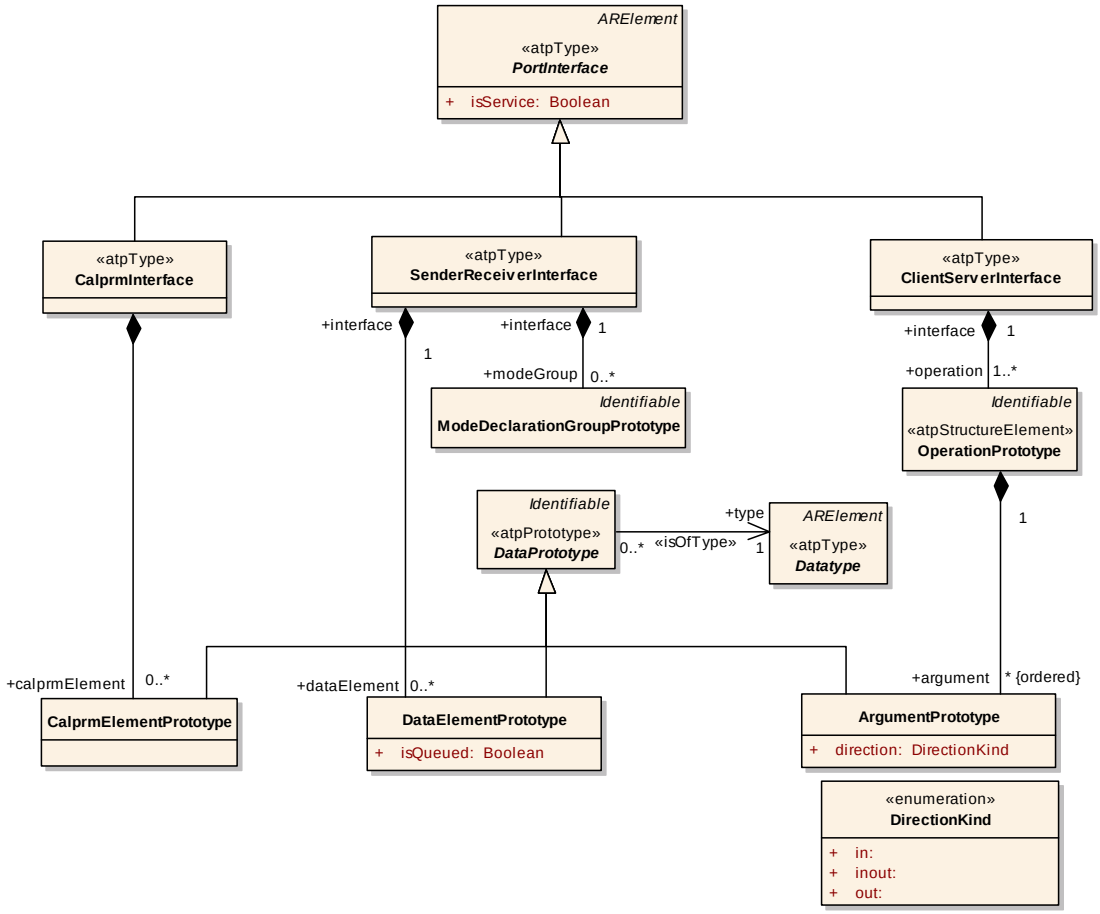

Figure 12: Interfaces in the AUTOSAR meta model.

The RTE or more precisely, the communication layer ensures that single data elements as well as the arguments of single operations are transmitted as atomic units.

4.1.2 Interface is a Namespace

An interface creates a namespace for the elements it contains. A client/server interface for instance (explained below) contains operations. The identifiers (names) of those operations are declared within the interface's namespace. Therefore, different interfaces may have operations with identical names.

4.1.3 Interfaces for AUTOSAR Services

"AUTOSAR Services" provide standardized functionality that is available on each AUTOSAR ECU. The VFB specification defines the concept in detail. "AUTOSAR Services" communicate with applications through "standardized AUTOSAR Interfaces". These consist of PortInterfaces that are standardized by AUTOSAR. Such a standardized PortInterface, which is intended for the communication between an application and an AUTOSAR Service, must have the flag "isService" set to TRUE.

4.1.4 Extensibility

It will be necessary to extend existing interfaces over time, e.g. simply due to product evolution. The compatibility rules mentioned in 4.6.5.3 do not simply rely on interface identifiers, but also check the interface data elements or operations if required. Therefore, there is no need for explicit extensibility rules that ensure backward/forward compatibility: simply define a new interface and apply the compatibility rules to see whether the old and the new interface match.
In particular, interfaces do not define any inheritance semantics. Here are several reasons why:

1. Over time a large inheritance tree will be created, which becomes hard to manage: changing base interfaces results in the *fragile base class syndrome*[4].
2. Merging interfaces introduces base interfaces multiple times in derived interfaces. What does this mean if the merged interfaces use different versions of the base interface?
3. While not yet modeled in AUTOSAR, interfaces also define a certain protocol. These protocols would also need to be inherited and extended, which is not trivial and not yet state of the art either.

4.1.5 Differentiation of Communication Paradigms

The specification of the Virtual Functional Bus (VFB) explains the two main paradigms for communication between two components: client/server, which is operation based, and sender/receiver, which is data based. The nature of those paradigms is quite different, and so are the attributes for interfaces operating within them. Therefore we distinguish pure client/server interfaces (c/s-interfaces) and pure sender/receiver interfaces (s/r-interfaces).

[4] An negative side effect of deep inheritance trees is the difficulty to maintain this tree. If a fundamental base class is changed, the whole inheritance tree is affected. The consequences are typically not predictable, unless the whole tree is maintained by a single CCB.

4.2 Sender/Receiver Communication

4.2.1 Sender/Receiver Interfaces

Sender/receiver interfaces (s/r interfaces) allow for the specification of the common 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 ports, s/r interfaces allow to formally describe what kind of data is sent and received.
Interfaces currently are limited to describe the static structure of this exchanged information, while the communication relevant dynamic attributes are attached to ports (see 7.4).
Those elements that are described in s/r interfaces are called data elements and mode groups. These are introduced in the next sections.

Class	SenderReceiverInterface			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Description	A sender/receiver interface declares a number of data elements to be sent and received.			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, PortInterface			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dataElement	DataElementPrototype	0..*	aggregation	The dataelements of this sender/receiver interface.
modeGroup	ModeDeclarationGroupPrototype	0..*	aggregation	modes which may be communicated via tha interface

The following semantic constraints are defined for the SenderReceiverInterface (from (SWComponentTemplate::PortInterface)

ID	*Constraint Description*	*OCL Expression*
0001	A SenderReceiverInterface must contain at least one data element or mode group	self.dataElement->size() + self.modeGroup->size() > 0

4.2.2 Data Elements

Data elements are declared within the context of a “SenderReceiverInterface”. They serve as the data units that are exchanged between sender and receiver. A data element, or more precisely a “DataElementPrototype” is created by choosing a data type – the datatype of the data element – and a name for the data element.

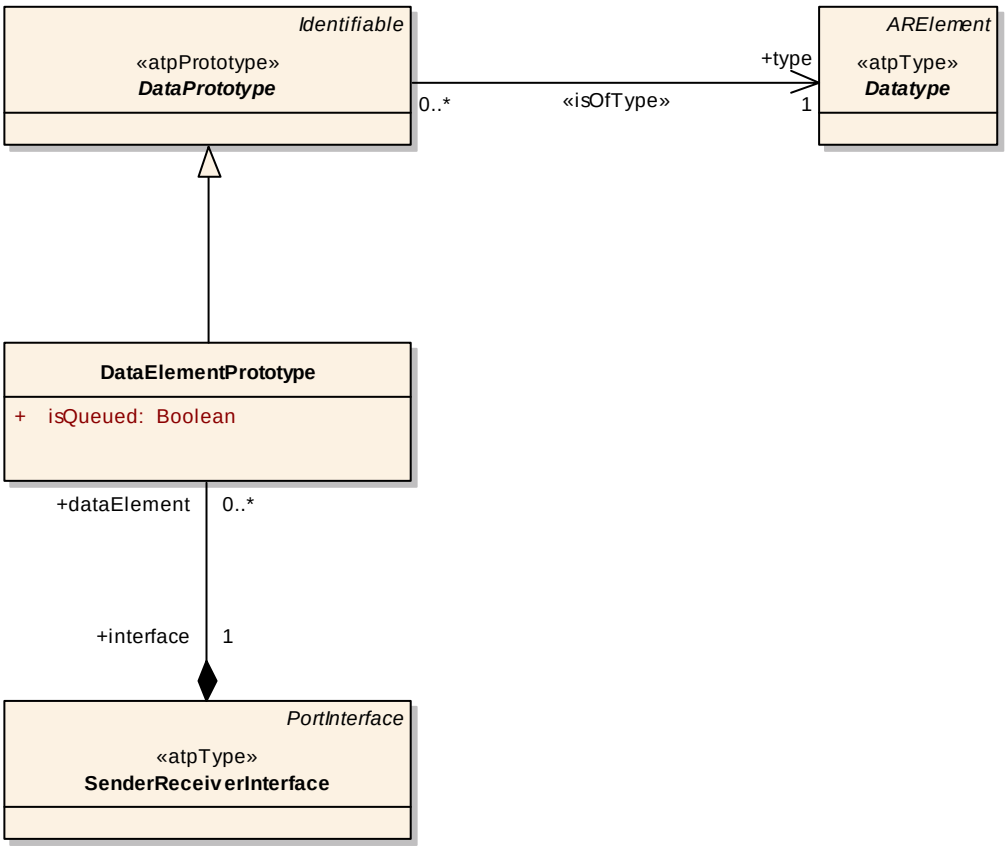

Figure 13: Data elements of a sender/receiver interface.

The datatype may be a predefined AUTOSAR datatype or a user defined datatype, it may be a primitive one or a composite one (see 4.4). The datatype in turn may be associated with certain data semantics like physical units.
The isQueued attribute of the DataElementPrototype qualifies whether its content 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.
Data elements may have associated communication properties that are not captured on the interface level and therefore are not discussed here. See 7.4 for more information on this topic.
While this definition of data elements may come very close to the reader's idea of a *signal*, note that signals have a very specific meaning in AUTOSAR (see AUTOSAR glossary). Therefore, a different term was chosen.

Class	DataElementPrototype
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface
Class Description	A data element of a sender-receiver interface, supporting signal like communication patterns.
Base Class(es)	Identifiable, DataPrototype

Attribute	Datatype	Mul.	Link Type	Attribute 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.

4.2.3 Mode Groups

In addition to the data elements a “SenderReceiverInterface” can define mode groups, which describe the collection of mode switches that can be communicated via the interface. For more details on modes please see chapter 5.

4.3 Client/Server Communication

4.3.1 Client/Server Interfaces

C/S interfaces aggregate a number of operations. They are the service oriented counterpart to the s/r interfaces mentioned above.

Class	ClientServerInterface			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Description	A client/server interface declares a number of operations that can be invoked on a server by a client.			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, PortInterface			
Attribute	Datatype	Mul.	Link Type	Attribute Description
operation	OperationPrototype	1..*	aggregation	
possibleError	ApplicationError	0..*	aggregation	Application errors that are defined as part of this interface.

4.3.2 Definition of Operations

This section describes in detail how operations are defined in the software-component template and what the semantics of such a definition are. It then motivates some of the design choices made and discusses how these operations might be mapped on programming languages.

4.3.2.1 Meta-Model: Summary

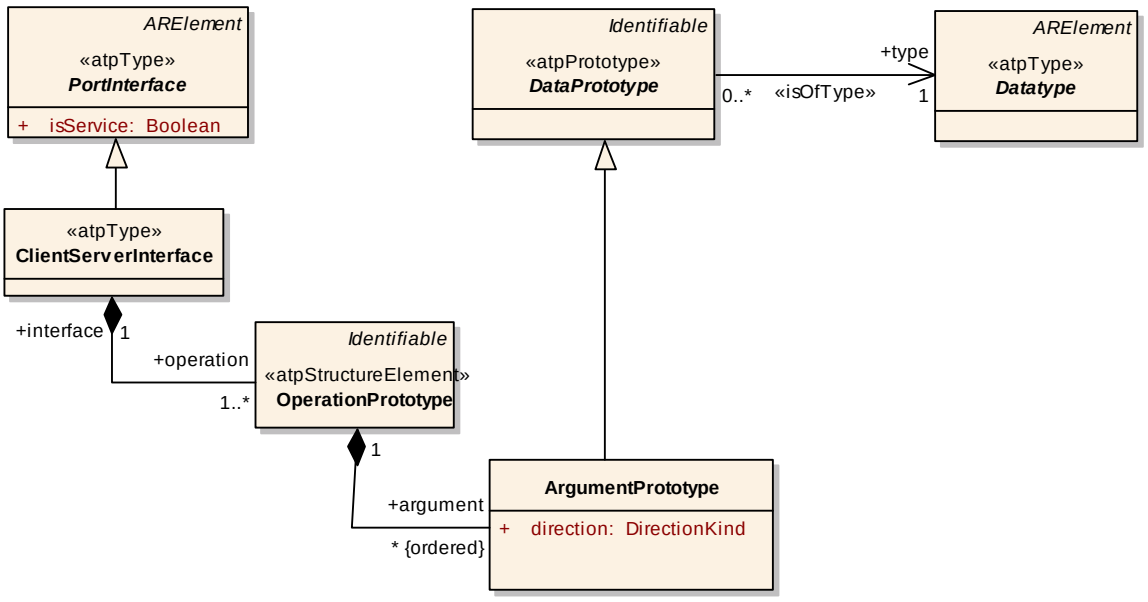

Figure 14: Excerpt of Meta-Model related to "Operation"

Figure 14 shows an excerpt out of the meta-model. A "ClientServerInterface" consists of "OperationPrototypes". An "OperationPrototype" consists of "ArgumentPrototypes" which define the in- and out-arguments that are passed between the client and the server. The direction of an "ArgumentPrototype" is defined through its “direction” attribute which is of type “DirectionKind”, see the class table of “DirectionKind” below. The "ArgumentPrototype" is a special kind of "DataPrototype" and is defined trough a reference to a "Datatype".

4.3.2.2 Metamodel Details

Each "OperationPrototype" is a Namespace and therefore has a unique identifier, which identifies the operation within the "ClientServerInterface". The operations have no ordering within the ClientServerInterface (there is no such thing as the “first” operation of an Interface)[5].
"OperationPrototypes" are not themselves “ARElement”s; they do not exist outside of the ClientServerInterface of which they are part but can only be specified as part of the definition of an interface.
Each OperationProtype consists of "ArgumentPrototypes". It is allowed that an operation has no arguments at all. Each "ArgumentPrototype" of the operation has a

[5] In different parts of the interface-definition, a "calling-order" of the operations might be prescribed: the client might be required to use the operations in a certain logical ordering. However, this ordering has nothing to do with the order in which the operations are listed in the interface.

unique identifier with respect to the operation. The ordering of the arguments is relevant (an operation has a “first” argument)[6].
Each "ArgumentPrototype" inherits from "DataPrototype" and therefore has a reference to a "Datatype". The direction of an argument can either be “in”, “out” or “inout”.
The following tables show the attributes of the various meta-classes in this model:

Class	ClientServerInterface			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Description	A client/server interface declares a number of operations that can be invoked on a server by a client.			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, PortInterface			
Attribute	Datatype	Mul.	Link Type	Attribute Description
operation	OperationPrototype	1..*	aggregation	
possibleError	ApplicationError	0..*	aggregation	Application errors that are defined as part of this interface.

Class	OperationPrototype			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Description	An operation declared within the scope of a client/server interface.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
argument	ArgumentPrototype	0..*	aggregation	
possibleError	ApplicationError	0..*	reference	Possible errors that may by raised by referring operation.

Class	ArgumentPrototype			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Class Description	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)	Identifiable, DataPrototype			
Attribute	Datatype	Mul.	Link Type	Attribute Description
direction	DirectionKind	1	aggregation	

[6] Giving the arguments of an operation 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 is an arbitrary choice made when implementing of the invocation. In “C”, only the datatypes and ordering of the arguments constitute the signature, NOT the names of the arguments.
Nevertheless, we decided that arguments are BOTH ordered and named. Whether operation-definitions that have arguments with same datatypes but with different names are “compatible” or not, is a different issue that is handled in sub-chapter INTERFACE-COMPATIBILITY (see 4.6.5.3).

4.3.2.3 Semantics of a Client-Server Interface

The client-server interface defines the interaction between a software-component that provides the client-service interface (called the "server" in the following) and a software-component that requires the client-service interface (called the "client" in the following).
When the client invokes an operation, it needs to provide a value for each argument that is of direction "in" or "inout". This value needs to be of the correct "Datatype". The client expects to receive a response to the invocation of the operation. As part of that response, it receives a value (of the correct datatype) for each argument that is of direction "out" or "inout".
Note that the client-server interface 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"). These issues are further touched upon in section 4.3.2.5 but are not in the responsibility of the Software Component Template work package.
Operations may also be associated with a number of application errors they possibly raise. Those errors are defined as part of the client/server interface. See section 4.3.3 for further details on VFB level errors.

4.3.2.4 Background Information and Motivation

The AUTOSAR operation model is kept very simple and does not have the features present in many common programming languages or component models. This section describes these specific aspects of the AUTOSAR operation model and motivates why these decisions were taken.

4.3.2.4.1 The Arguments have NO Default Values

The operations of a Client-Server Interface currently do not have the possibility to set “default values” for arguments. Default values can complicate mappings to programming languages and can confuse the issue of “extensibility”.

4.3.2.4.2 An Operation or an Interface is NO Datatype

In many common programming languages (like “C”), an operation is another datatype. This makes it for example possible to pass a reference to an operation as an argument in another operation.
This is NOT allowed in AUTOSAR: it is not possible to pass a reference to an operation as an argument in another operation. Essentially all arguments in an AUTOSAR operation 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 argument). 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).

4.3.2.5 Mapping the AUTOSAR Operations on Programming Languages

The description mechanism and semantics defined in the software-component template leave a lot of freedom in mapping the operations on specific programming language mechanisms.
Further AUTOSAR WPs will define these mappings. The detailed mapping will not only depend on the client-server interface as defined in this section but also on:

1. the programming language used (a very different style might be used for C, C++ and Java)
2. whether the software-components in client- and server-role are multiply instantiatable or not
3. whether the client makes a synchronous or asynchronous invocation of the operation
4. etc.

Therefore, it is enitrely left open to other work-packages to define for example:

1. how arguments with direction "in" or "inout" are passed from the client to the RTE (by value, through pointers, through separate arguments in a "C"-function-call, as one large structure which is passed as one argument,…)
2. how arguments with direction "out" or "inout" are received by the client (as a return-value of the "C"-function-call, through arguments, in one large structure,…)
3. how memory is managed (who is responsible for providing space to store the arguments: the software-component or the RTE).
4. etc.

The only important requirement is that the semantics defined in section 4.3.2.3 are respected.

4.3.3 Support for Error Handling

This section describes how the software component template supports handling of errors occuring either within a software component or during the communication across the VFB.

4.3.3.1 Error Types

This template only addresses errors that are either generated by or relevant for software components. Errors that are created and consumed by basic software modules are not in scope.
Those errors are divided into two simple classes: infrastructure errors and application errors.

Infrastructure 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.
Those errors are listed in the VFB specification, 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 deter-

mine infrastructure faults and communicate the corresponding errors to the relevant software components.
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
Application errors on the other hand are specific to the functionalty or information that is described in form of a port interface. It is not possible to define such errors up front, instead they are defined at design time of a certain interface.
In principle, such errors could be part of all interface kinds, but as of now, AUTOSAR supports application errors only for client/server interfaces.

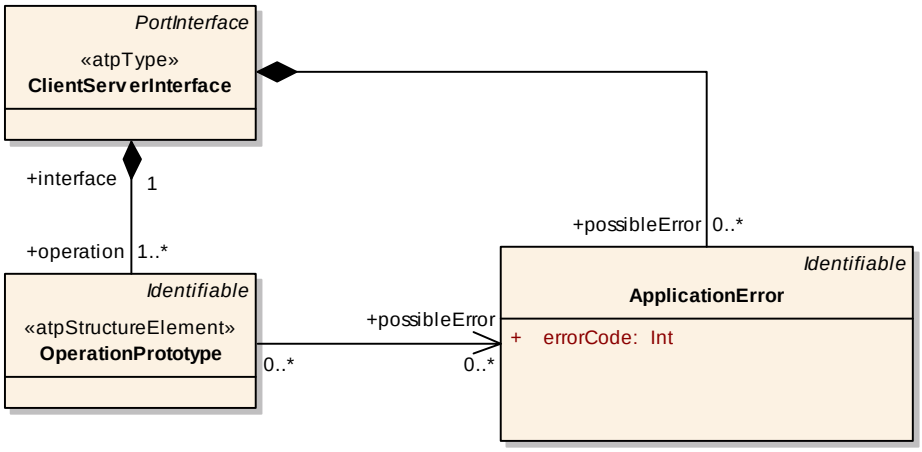

Figure 15: VFB error metamodel.

4.3.3.2 VFB Error Model

Figure 15 shows the metamodel of VFB errors. As infrastructure errors are inherently defined as part of the VFB, only application errors have to be described as part of the software component template.
They are defined as part of a client/server interface, which also contains a set of operations. Each of those operations can specify a number of application errors it possibly raises.

Class	ApplicationError			
Package	M2::AUTOSARTemplates::SWComponentTemplate::VFBErrors			
Class Description	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	Attribute 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).

4.4 Data Types

4.4.1 Overall Concepts

4.4.1.1 Levels of Abstraction in Defining Data Types

In the context of defining data types, AUTOSAR distinguishes between the levels shown in Figure 16.

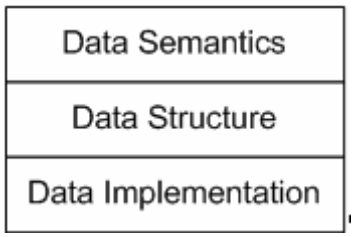

Figure 16: Levels of abstraction

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 "ints" and "floats") is defined. On top of this, it is usually possible to build various structures with these primitive types.
The Data-Implementation level is the implementation of these 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 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 components are 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.

4.4.1.2 Usage of Data Types

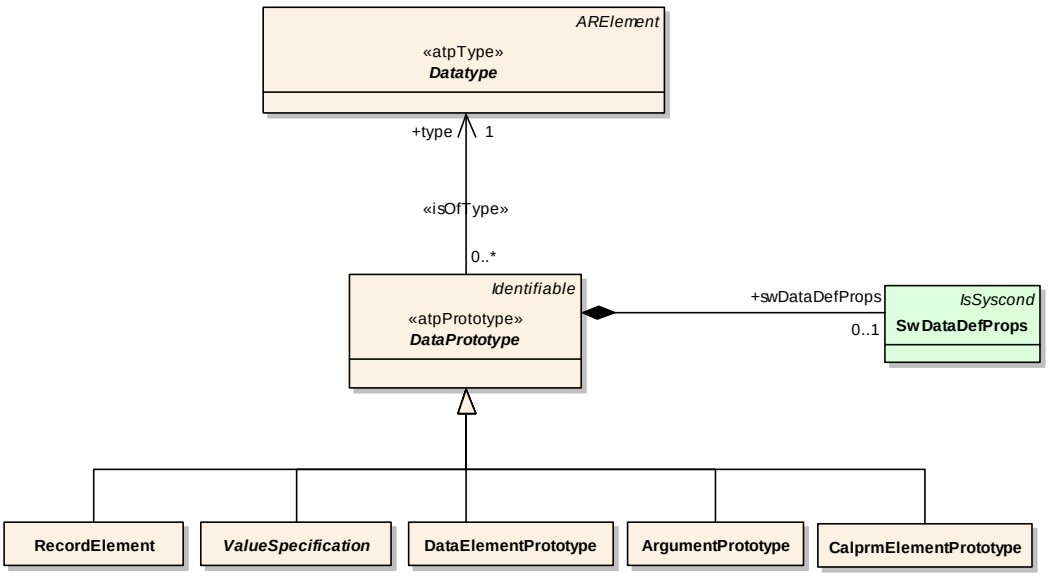

Figure 17: Data type usage

The above figure indicates some of the usages of a datatype in the model. E.g. they define

- elements in a record structure,
- constant values,
- data-elements inside a sender-receiver interface, or
- arguments of the operations in a client-server interface.

Note that the data types do *not* contain any information on the evolution of the values in the data type 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.

4.4.1.3 What is Defined Where

This sub-chapter deals with the "Data Structure": it explains what primitive Data-Types are available at this level within AUTOSAR and how new Data-Types can be constructed.
The following sub-chapter deals with the optional "Data-Semantics" which describe the correct interpretation of the values stored in the Data-Structures.
The "Data Implementation" is not necessarily described in the Software-Component Template but depends on the medium on which the Data-Structure is used. Note that for measurement and calibration this can be specified using baseType.

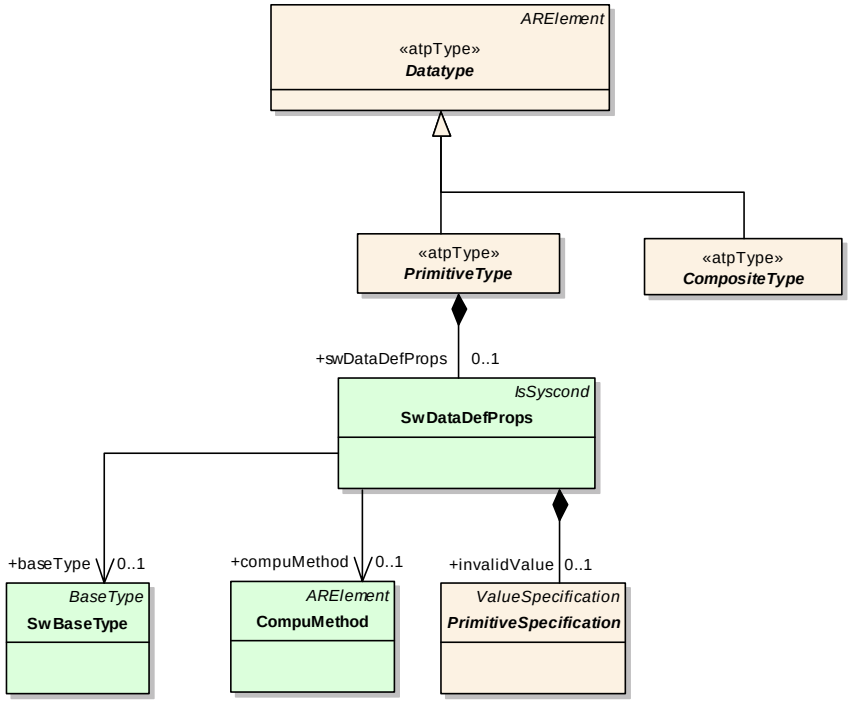

Figure 18: Data type summary

AUTOSAR provides a set of primitive data type-description and a mechanism to describe arrays and records. Semantics (as described in the next chapter) can be attached to primitive types.

4.4.2 Primitive Data Types in AUTOSAR

4.4.2.1 Rationale

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 SW-components must be as efficient as possible. So 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 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, AUTOSAR needs to define a fixed mapping between the structural data types and their implementations in a specific programming language.

4.4.2.2 Overview of the Definition of Primitive Types

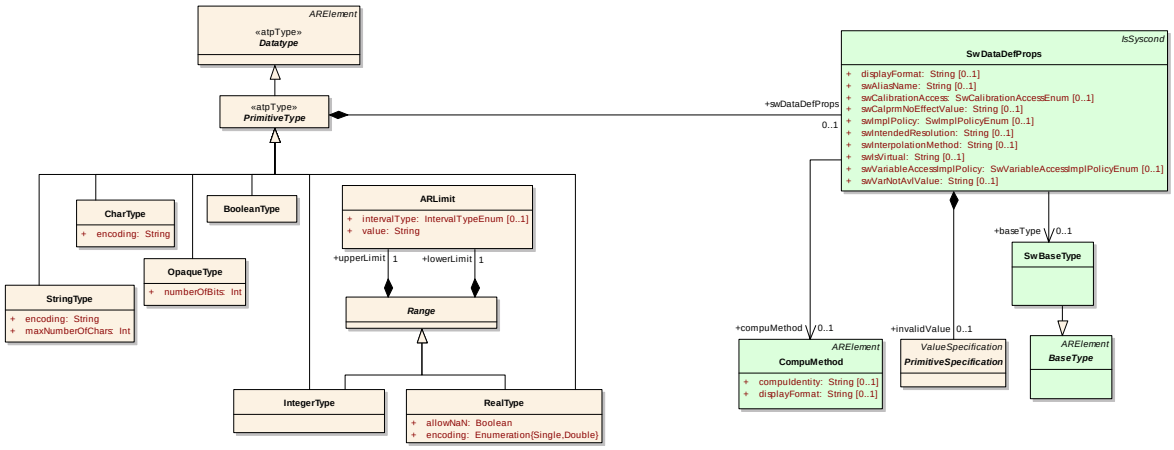

Figure 19: Primitive datatype

4.4.2.3 Range

When defining a data type, it is often necessary to specify an open or closed range of values.

Class	Range {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Description	Abstract class for specifying a range from lower limit to upper limit.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute 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.

Semantically, the range is all real numbers which are defined by:

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

Note that lowerLimit and upperLimit can be "+INFINITY" or "-INFINITY".

4.4.2.4 Description of Primitive Data Types

The following sections describe how primitive data types can be defined in AUTOSAR.

4.4.2.4.1 Boolean

Class	BooleanType
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes
Class Description	This datatype represents a set containing the logical value true and false
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, Datatype, PrimitiveType

4.4.2.4.2 Opaque

Class	OpaqueType			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Description	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)	Identifiable, PackageableElement, NonSplitableElement, ARElement, Datatype, PrimitiveType			
Attribute	Datatype	Mul.	Link Type	Attribute Description
numberOfBits	Integer	1	aggregation	The number of bits that are used to make up the opaque type.

4.4.2.4.3 Integer

Class	IntegerType
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes
Class Description	This data-type are the integers in the interval defined by the Range.

Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, Datatype, PrimitiveType, PerInstanceMemory, NVRAMMapping, ARObject, Range

“IntegerType” inherits from “Range” (see section 4.4.2.3) and “PrimitiveType”. Therefore the attributes “upperLimit” and “lowerLimit” are defined implicitly.
Semantically a range of type “IntegerType” is the intersection of the range of real numbers as defined section 4.4.2.3 and the numbers that can be expressed by the data type integer.
For example, the following values of the “IntegerType” attributes define a data-type containing the integers 0, 1, 2 and 3.

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

4.4.2.4.4 Real

Class	RealType			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Description	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)	Identifiable, PackageableElement, NonSplitableElement, ARElement, Datatype, PrimitiveType, PerInstanceMemory, NVRAMMapping, ARObject, Range			
Attribute	Datatype	Mul.	Link Type	Attribute Description
allowNaN	Boolean	1	aggregation	Denotes whether this data type permits for "not a number" being represented by the type
encoding	RealTypeEncodingEnum	1	aggregation	Denotes whether single or double precision is used.

When encoding is "Single" or "Double", the values in this data type are the reals 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”. In other words: A range of type “RealType” is the intersection of the range of real numbers as defined section 4.4.2.3 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, the ones used by special DSPs).

4.4.2.4.5 Char

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

For the definition of the encoding of “CharType” and “StringType” the names described in the following table 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*[7]. 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.

The table was created by

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

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

4.4.2.4.6 String

Class	StringType
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes
Class Description	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:

[7] http://www.iana.org/assignments/character-sets

	bytes required to represent the string = maxNumberOfChars * (max bytes per character using the given encoding) + 1 (terminating null)			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, Datatype, PrimitiveType			
Attribute	Datatype	Mul.	Link Type	Attribute 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.

The naming conventions for the description of encoding are described on section 4.4.2.4.5.

4.4.2.5 Note on Enumeration

Enumeration is not a primitive datatype. Rather a range of integers can be used as a structural description. The mapping of the integers on "labels" in the enumeration is seen as Data-Semantics and not as part of the structural description.

4.4.3 Composite Data Types

The composite data types "array" and "record" provide the means to build new data types. It is possible to combine these two, so that an "array" could be an element of a "record" and in the same manner a "record" could be an element type of an "array". Nesting these data types is also possible.

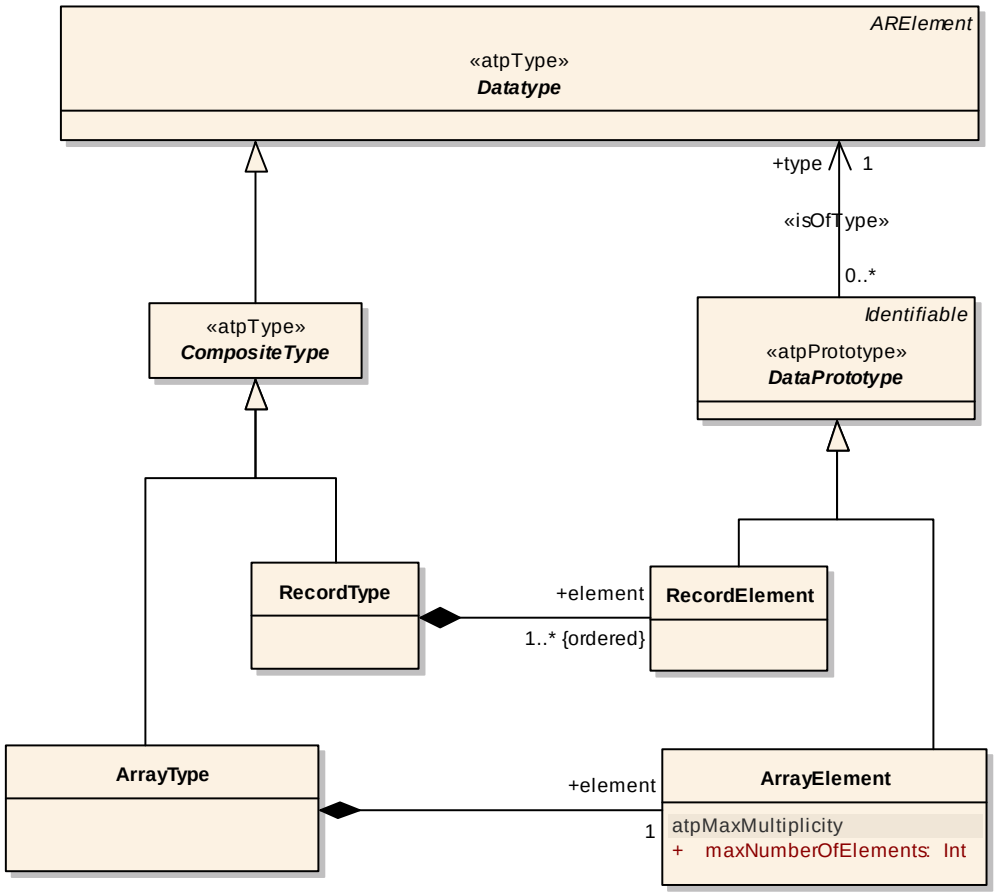

Figure 20: Composite data type

4.4.3.1 Array

An array can contain maxNumberOfElements elements that each have the same type. When referring to an element of an array within the software-component descriptions, the element-index runs from 0 to maxNumberOfElements-1.

Class	ArrayType			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Description				
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, Datatype, CompositeType			
Attribute	Datatype	Mul.	Link Type	Attribute Description
element	ArrayElement	1	aggregation	

Class	ArrayElement
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes
Class Description	

Base Class(es)	Identifiable, DataPrototype			
Attribute	Datatype	Mul.	Link Type	Attribute Description
maxNumberOfElements	Integer	1	aggregation	The maximum number of elements that the array can contain.

4.4.3.2 Record

A declaration of this type describes a nonempty set of objects, each of which has a unique identifier with respect to the record-type and a data-type. The name of each object must be unique within this set of objects and the data type could be any AUTOSAR data type.

Class	RecordType			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Class Description				
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, Datatype, CompositeType			
Attribute	Datatype	Mul.	Link Type	Attribute Description
element	RecordElement	1..*	aggregation	

Class	RecordElement
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes
Class Description	An element in a record.
Base Class(es)	Identifiable, DataPrototype

4.4.4 Constants

The software component template allows utilizing constant values in two ways: (a) by referencing a publicly defined constant (represented as class “ConstantSpecification”), or through an *inline* aggregation of a constant value (class “ValueSpecification”).
The structure of a constant value is defined by its datatype. Specialized value classes allow the definition of values for the different kinds of datatypes, e.g. “BooleanValue” specifies the value for a “BooleanType” and an “ArraySpecification” does the same for an “ArrayType”.
A specific value specification is the “ConstantReference”: it passes the definition of the constant value on to another constant (“ConstantSpecification”) that is defined as part of an AUTOSAR package.

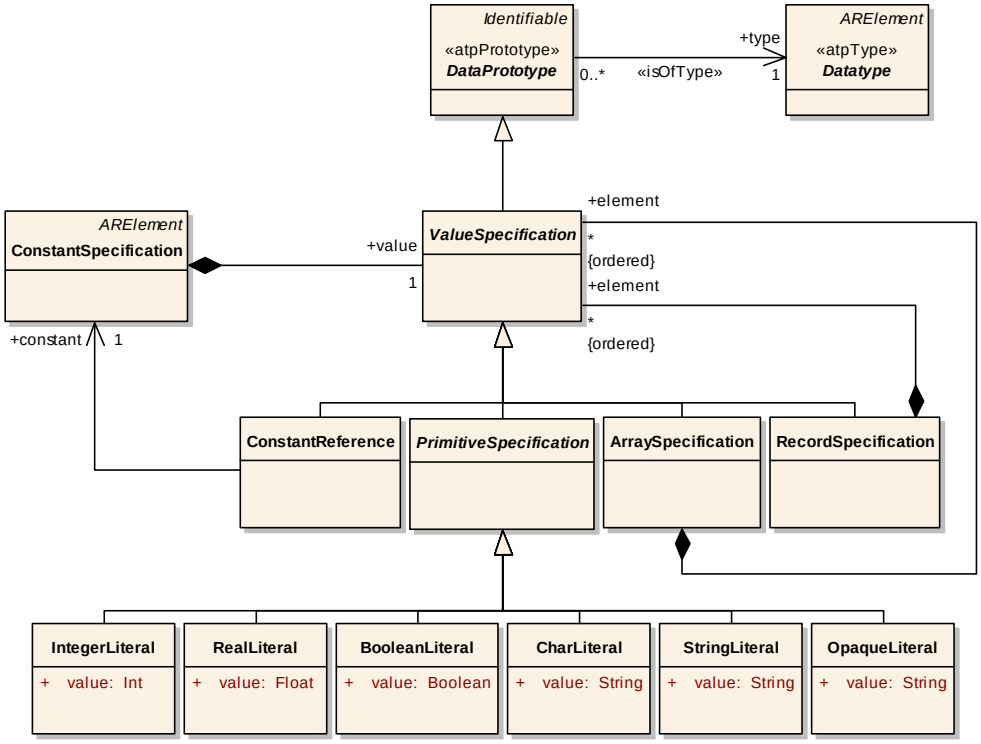

Figure 21: Model of constants.

4.5 Data Semantics

4.5.1 Overview

To ensure compatibility between communicating software components, not only the datatypes 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 the following sections those two representations are referred to as *physical* and *internal*.

4.5.2 Data Types with Semantics

It does not make sense to specify semantics and therefore a physical meaning to all of the datatypes explained in the previous section. More precisely, data semantics may be assigned to primitive datatypes only. An array or a record of datatypes can-

not be given a particular semantic meaning besides the one occasionally specified for the contained primitive data elements.
Since primitive datatypes with specified semantics may often be reused,it is possible to assign additional properties to a PrimitveType using swDataDefProps. The actual semantics class is called “CompuMethod”, due to compatibility with MSR-SW.

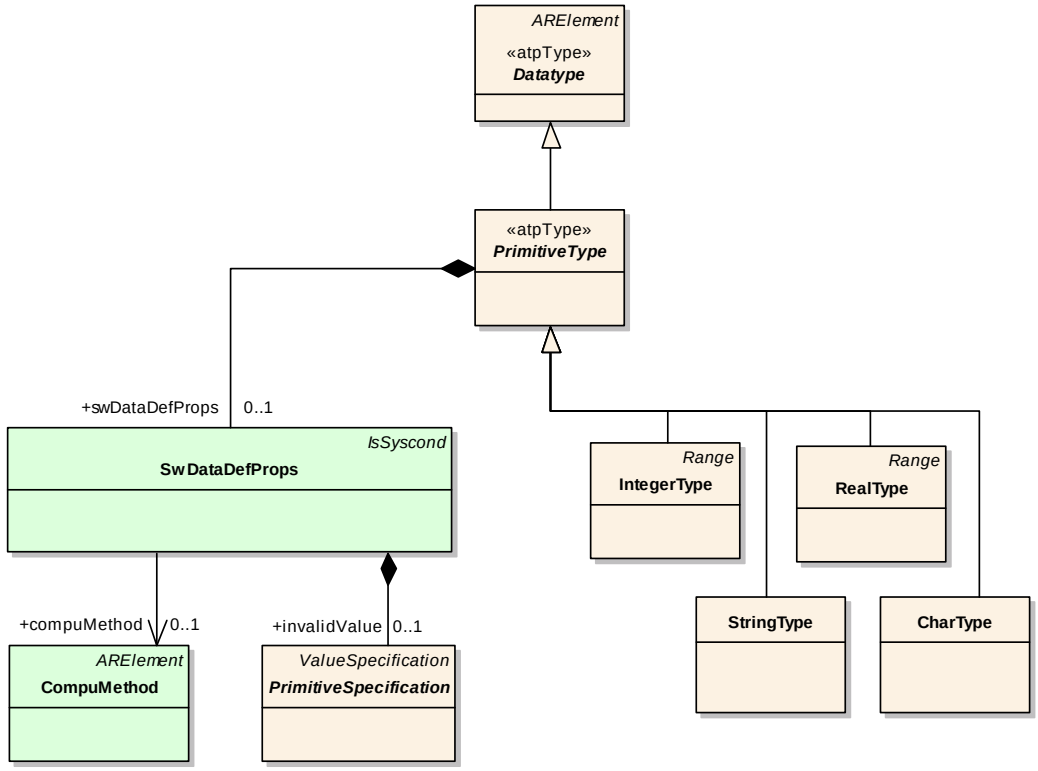

Figure 22: Primitive types and semantics.

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, 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 primitive type (not shown in diagram). Constants are explained in the previous chapter on data types.
The following section explains the usage of the class “CompuMethod” in order to allow specification of the data semantics of a primitive datatype.

4.5.3 CompuMethod

This class was taken from the ASAM standard’s harmonized data objects, this is also indicated by the green color of the classes in the figure 29.
CompuMethods 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 that is 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 is used in both directions. For simple functions like identical or linear functions this is sufficient because the inverse function 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.
As CompuMethod specify the conversion between the physical world and the numerical values, they must refer to a unit.

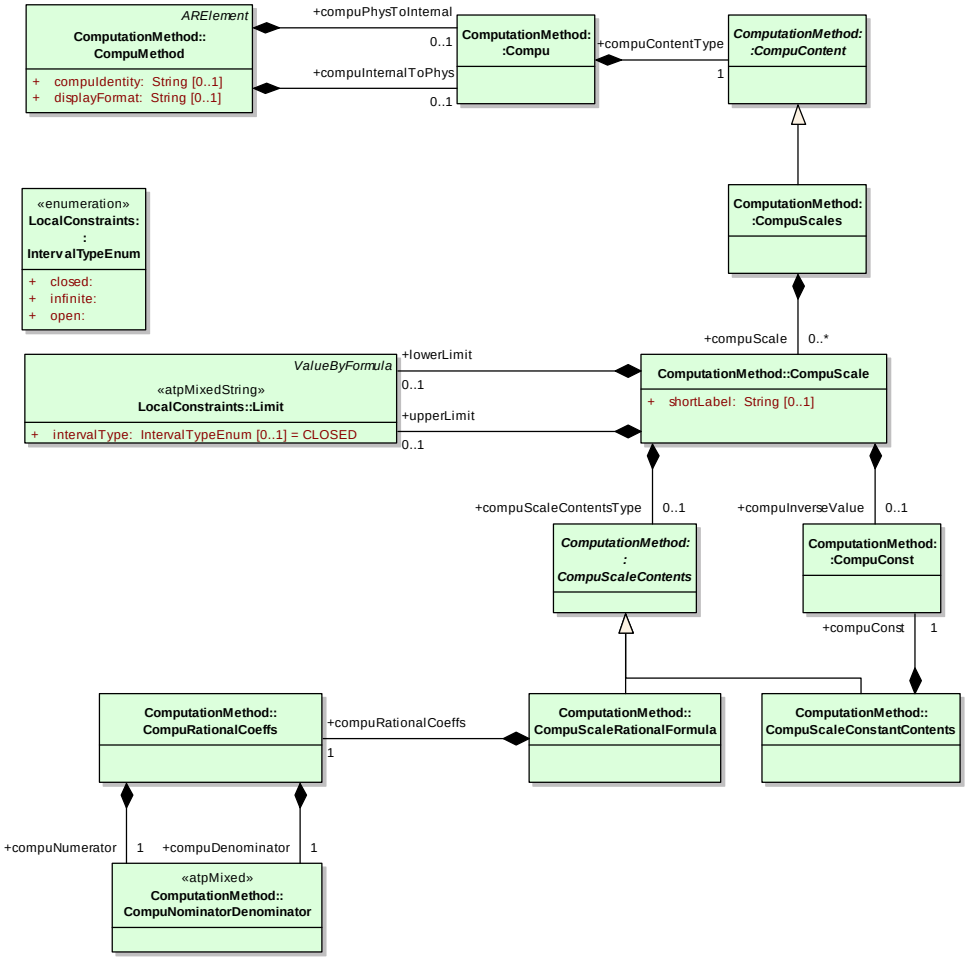

Figure 23: A CompuMethod and its attributes define data semantics.

Figure 23 shows 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 assotiated. Note that quantities like “%” are not 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 class. A number of derived 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). 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 x_0, the second <V> is the coefficient for x_1, etc. With this sequence the values of the exponents can be entirely represented.

For a detailed description of compuMethods, please refer to the ASAM MCD 2 Harmonized Data Objects[8].

In particular, the following categories apply:

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, PHYSCONSTR and INTERNALCONSTR 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	In case of SCALE-LINEAR more than one COMPU-SCALE can be defined. Additionally there have to be the UPPER-LIMIT and LOWER-LIMIT elements, which define the region

[8] At the time of this writing the file was available on the AUTOSAR server in .../04PapersAndResults/04WPs/06WP2.1.1.1/trunk/06InfoSources/10ASAM+MSR/MCD2-HDO-V1[1].0.zip

		of validity for the linear function. The boundaries of the regions must not overlap.
RATFUNC	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, … 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.
SCALE-RATFUNC	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.

4.5.3.1 Example how to Specify an 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>
```

4.5.3.2 Example how to Specify an Linear Conversion

The following example illustrates how an 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>
```

4.5.4 Physical Units

An important part of the semantics associated with a datatype 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[9] (e.g. veloc-

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

ity 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.
Figure 24 shows the concept of how units are defined.

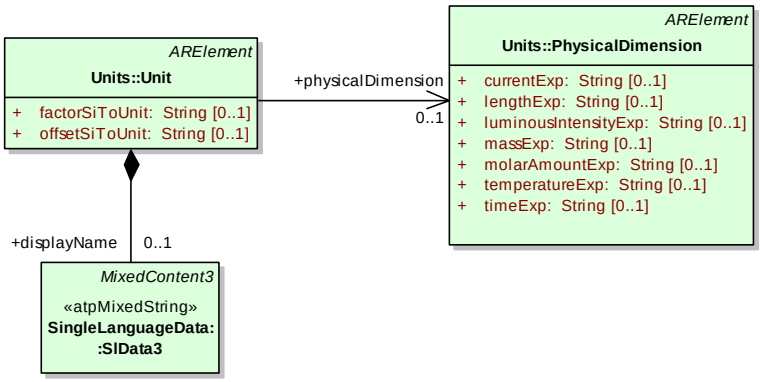

Figure 24: Definition of SI based units.

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[10].
To base a new unit directly on 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.
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[11].

4.5.5 BaseType

BaseType is used to specify in detail the Data Implementation level mentioned in chapter 4.4.1.1. For a detailed description of BaseTypes, please refer to the ASAM MCD 2 Harmonized Data Objects[12]. The information is necessary to create an A2L-File.

[10] At the time of this writing the file was available on the AUTOSAR server in .../04PapersAndResults/08InterWP/02WP10.x/02Documents/03UnitsAndDataTypes
[11] At the time of this writing the file was available on the AUTOSAR server in .../04PapersAndResults/08InterWP/02WP10.x/02Documents/03UnitsAndDataTypes
[12] At the time of this writing the file was available on the AUTOSAR server in .../04PapersAndResults/04WPs/06WP2.1.1.1/trunk/06InfoSources/10ASAM+MSR/MCD2-HDO-V1[1].0.zip

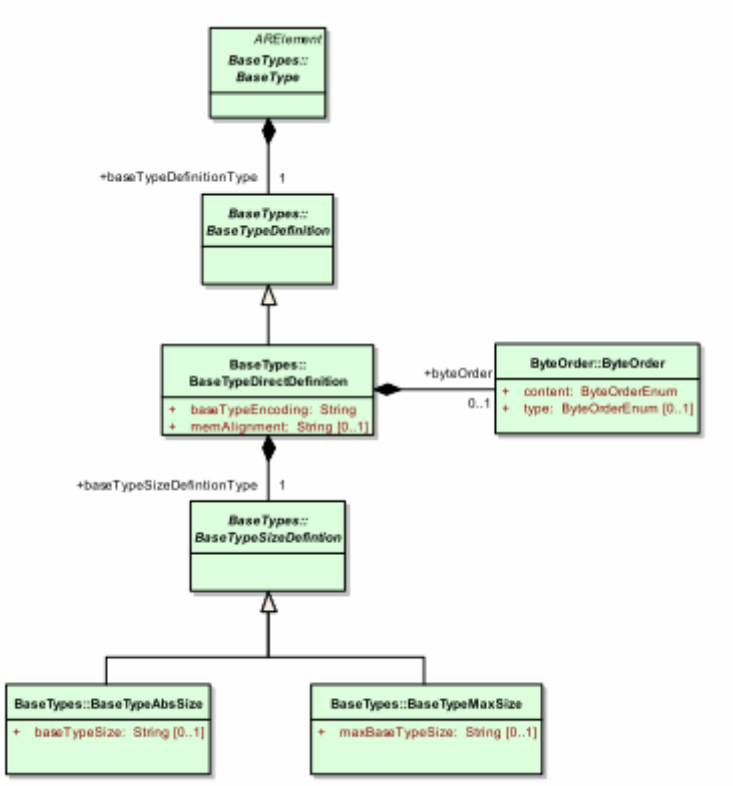

Figure 25: Base Type

The properties of BaseType are:

- For CATEGORY only the values "FIXED_LENGTH" and "VARIABLE_LENGTH" are supported. In case of "FIXED_LENGTH" BaseTypeSize is filled. In case of "VARIABLE_LENGTH" MaxBaseTypeSize 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
  - · 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. E.g. if MemAligment is 16, the data object in question is aligned to a memory adress dividable by 2.

- ByteOrder specifies the ordering of bits in memory. Possible values are MOST-SIGNIFICANT-BYTE-FIRST and MOST-SIGNIFICANT-BYTE-LAST.

4.6 Application Level Port Annotations

4.6.1 Introduction

Embedded Automotive Software is used to implement open- 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.
The various port annotations described in the next chapters are by no means exhaustive. This situation is reflected by the meta-model element extAnnotation aggregated by the abstract class PortAnnotation. This allows to specify additional annotations via name and free text.

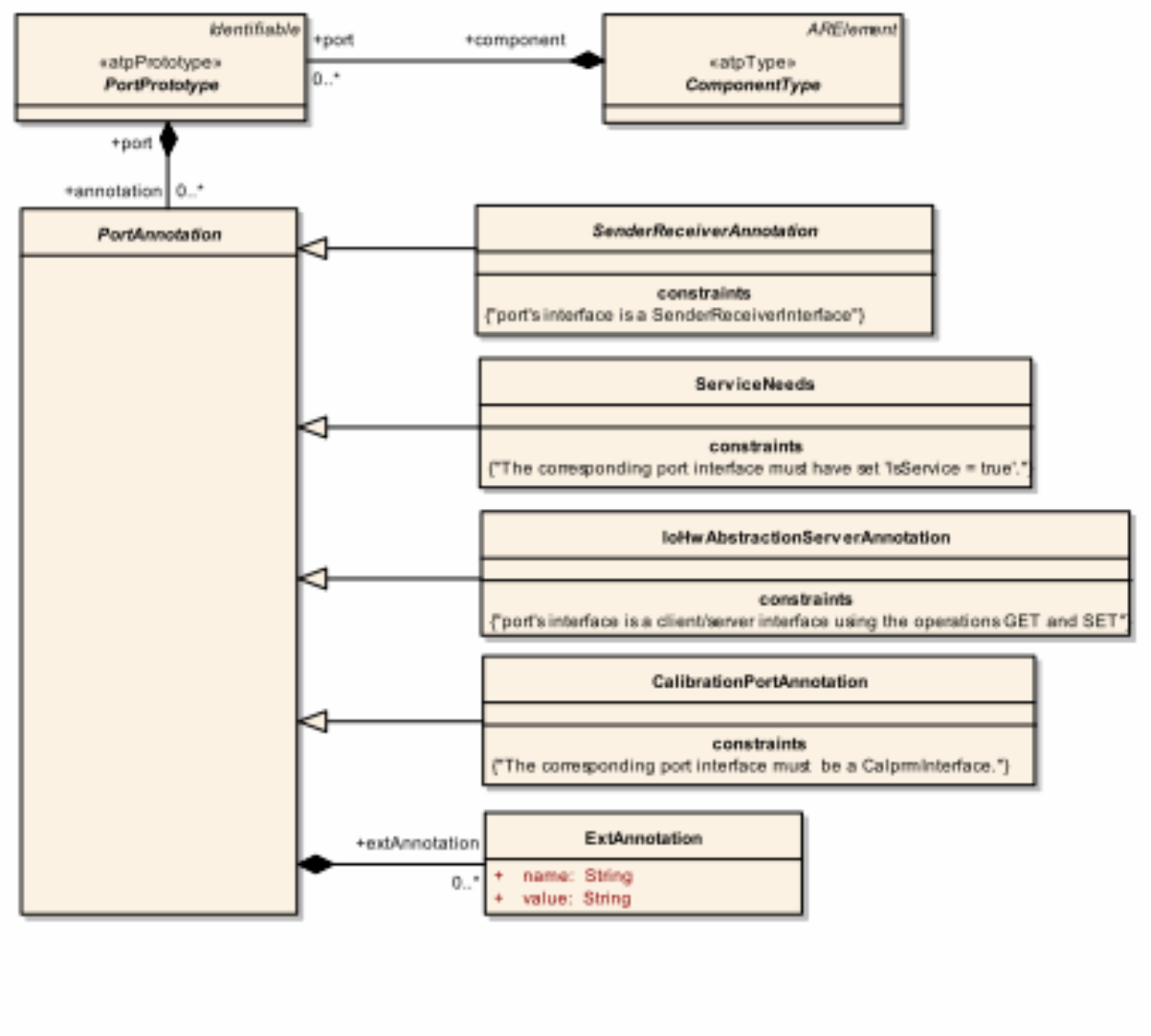

Figure 26 Application Level Port Annotations Overview

Class	PortAnnotation {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Description	Base class for annotations of ports. Annotations are attributes that formally specify features and requirements of ports that have currently no influence on the generation of the RTE or the communication in general.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
extAnnotation	ExtAnnotation	0..*	aggregation	Additional annotations types that are not yet part of the software component template and can be specified on a pure text basis.

Class	ExtAnnotation			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Description	Allows the specification of port annotations not yet part of the software component template.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
name	String	1	aggregation	Name of the annotation.
value	String	1	aggregation	Value of the annotation.

4.6.2 Sender/Receiver Annotation

Typical annotations for sender/receiver are:

- Signal Age: The attribute signal age expresses that the associated software-component will only work correctly provided it took no longer than a certain time that the signal was read from the sensor. Of course, this cannot 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 data-element in a R-Port of a software component using this annotation indicates to the control engineer, developing a control-algorithm for this component, that the signal has to be filtered (This relationship holds for Sender-Receiver interfaces).
- 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 annotations 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 annotations 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.

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. 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 compu-methods.

The application level port annotations for sender/receiver have to be associated to each data element in a port, e.g. there might be a “raw” data element and a “filtered” data element in the same port! Furthermore, if two data-elements use the same application level port annotation, a reference from the annotation to the data-elements will be established by an appropriate tool.

As shown in Figure 27 the port annotations for sender/receiver 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 a receiver port, the signal age of the value, carried by the associated connector, can be specified. Each of these groups can be interpreted as a property of the signal-quality.

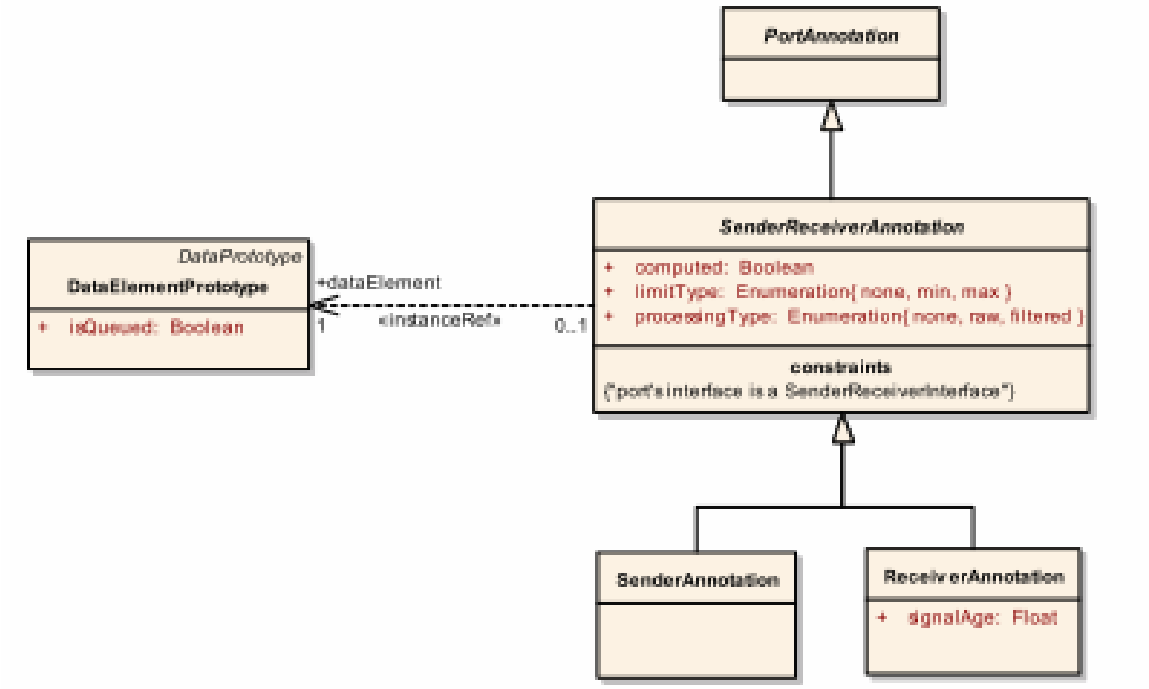

Figure 27 Application Level Port Annotations for Sender/Receiver

Class	SenderReceiverAnnotation {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Application-Attributes			
Class Description	Annotation of the data elements in a port that realizes a sender/receiver interface.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PortAnnotation			
Attribute	Datatype	Mul.	Link Type	Attribute 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.
limitType	SenderReceiver-AnnotationLimit-TypeEnum	1	aggregation	Indicates whether the data element carries a minimum or maximum value, thereby limiting the current range of another value. 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 softwar component's runnable is transmitted to another software component whose runnable will use this value as a limit, e.g. the max.power which can be used by that component, or the current min. slip.
processingType	SenderReceiver-Annotation-ProcessingType-Enum	1	aggregation	Kind of processing applied to the data element. "raw" specifies that a signal is taken directly from the basic software modules, i.e. from the ECU abstraction layer. It indicates to a developer that the control algorithm in the software has to provide filters. "filtered" indicates that a raw signal has been manipulated by some application software components by using filters. "none" is specified if none of the previous two options apply.
senderReceiverAnnotation_data-Element	SenderReceiver-Annotation_data-Element	1	reference to instance	

Class	SenderAnnotation
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes
Class Description	Annotation of a sender port, specifying properties of data elements that don't affect communication or generation of the RTE. The given attributes are assertions on the provided data.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PortAnnotation, SenderReceiverAnnotation

Class	ReceiverAnnotation			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PortAnnotation, SenderReceiverAnnotation			
Attribute	Datatype	Mul.	Link Type	Attribute 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.

4.6.3 Annotations for the I/O Hardware Abstraction Layer

Remark: The concept being described in the following section is likely to be revised in AUTOSAR Phase II. In particular, the attributes BswRangeMin, BswRangeMax, BswResolution and Unit of physical signals are currently being described by attributes of class IoHwAbstractionServerAnnotation. 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. This way, the Range and Unit attributes will be expressed by ordinary Datatype semantics as detailed in chapter 4.5.2
Within the ECU-Abstraction Layer there are ECU-signals defined. These signals represent the electrical signals as they arrive in the µCs 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 sensor- or an actuator software component. The operations prove an argument prototype where several annotations can be assigned to. They are shown in the IoHwAbstractionSeverAnnotation class in Figure 28. A detailed description of the attributes can be found in the IoHwAbstraction Layer software specification document. 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 sender-receiver port annotation is more generic. Especially, there is no relationship with the µC's peripherals.

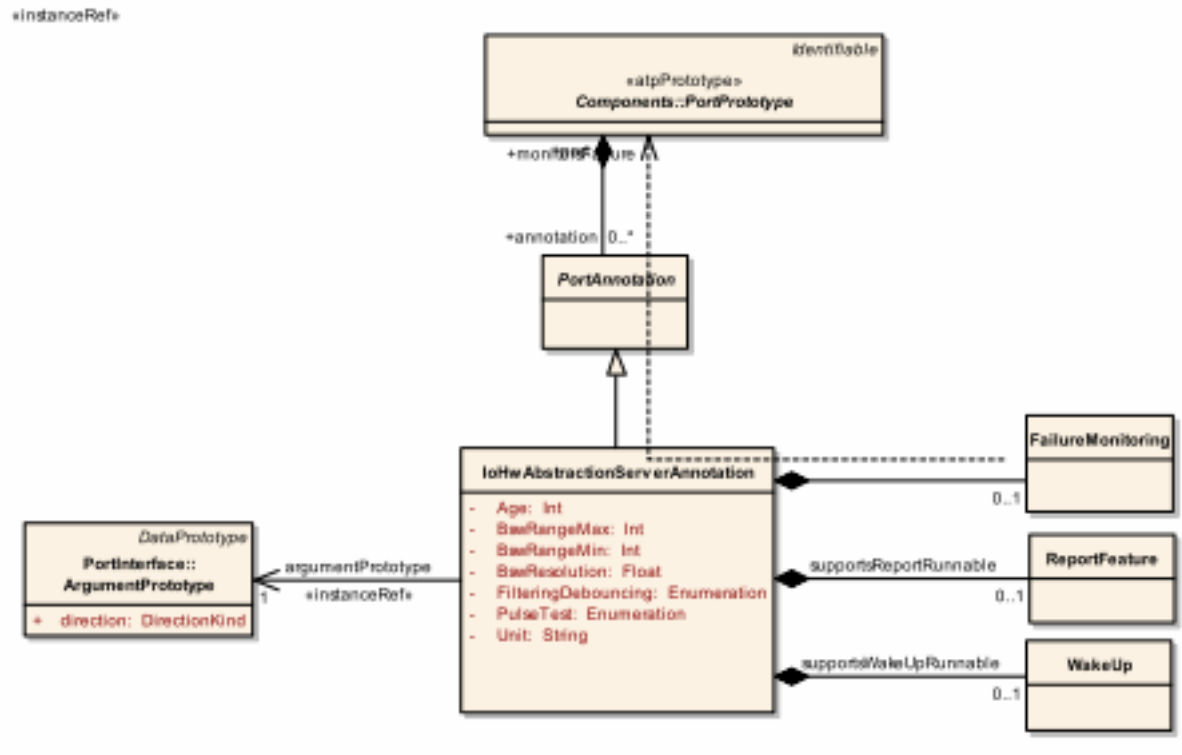

Figure 28: IO-Hardware-Abstraction-Server-Annotation

Class	IoHwAbstractionServerAnnotation
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes

Class Description	The ClientServer Port Annotation will only be used from a sensor- or an actuator component while interacting with the IoHwAbstraction layer			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PortAnnotation			
Attribute	Datatype	Mul.	Link Type	Attribute 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
argumentPrototype	IoHwAbstractionServer-Annotation_argument-Prototype	1	reference to instance	
BswRangeMax	Integer	1	aggregation	Specifies the maximum value of the Range the ECU-Signal is supposed to have
BswRangeMin	Integer	1	aggregation	Specifies the maximum value of the Range the ECU-Signal is supposed to have.
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) / (2^datatypelength - 1)
dataElementPrototype	IoHwAbstractionServer-Annotation_dataElement-Prototype	1	reference to instance	
failureMonitoring	FailureMonitoring	0..1	aggregation	
FilteringDebouncing	IoHwAbstractionServer-AnnotationFiltering-DebouncingEnum	1	aggregation	This attribute is used to indicate what kind of filtering/debouncing has been put to the signal in the IoHwAbstraction layer. RAW_DATA means that no modification of the signal has been applied.This is the default value DEBOUNCE_DATA means that the signal is a mean value WAIT_TIME_DATE means that the signal is delivered by a GET operation after a certain amount of time
PulseTest	IoHwAbstractionServer-AnnotationPulseTestEnum	1	aggregation	This attribute indicates to the connected Actuator Software component whether the data-element can be used to generate pulse test sequences using the IoHwAbstration layer
reportFeature	ReportFeature	0..1	aggregation	
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
wakeUp	WakeUp	0..1	aggregation	

Class	ReportFeature
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes
Class Description	This feature is used for input signals only. Each time the level of the associated data-element changes a dedicated GET operation has to be invoked by the sensor software component, i.e. an appropriate runnable has to be started.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject

Class	WakeUp
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes
Class Description	This attribut indicates wheter the software component reading this value provides dedicated runnables to handle a wake-up call by the IoHwAbstraction layer.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject

Class	FailureMonitoring			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Description	This attribute is only applicaple in SET operations. If it is enabled, the IoHwAbstraction layer will monitor the result of the opeation 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			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
monitorsFailure	FailureMonitoring_monitors-Failure	1	reference to instance	The referenced port monitors a failure in the data-element to be monitored of the IoHwAbstraction layer. The referenced port has to be an Actuator Component.

4.6.4 Calibration Port Annotation

The Calibration Port Anotation can be used to define that the date provided at the port is derived using some calibration parameters. The CalibrationPortAnnotation provides a reference to the CalprmElementPrototype.
The main use-case is to allow easy access to the information which calibration parameters influence the data on the port.

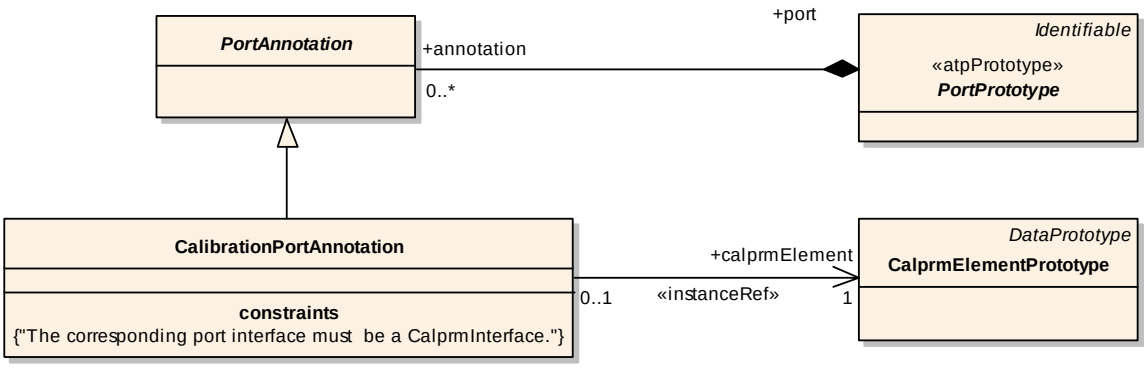

Figure 29: Calibration Port Annotation

Class	CalibrationPortAnnotation			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Description	Annotation to a port used for calibration regarding a certain CalprmElement			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PortAnnotation			
Attribute	Datatype	Mul.	Link Type	Attribute Description
calprmElement	CalibrationPortAnnotation_calprm-Element	1	reference to instance	The instance of calprm element annotated.

4.6.5 Service Port Annotation

Application SW-Components are designed to be independent of their mapping to actual ECU Hardware. However, each SW-Component might need services which are provided by the ECU's Basic Software through AUTOSAR Services. The "ServiceNeeds" are used to provide detailed information what the SW-Component expects from the AUTOSAR Services when integrated on an actual ECU.

Note that in addition to the annotations for the services described in the following chapters, it is also possible to use ExtAnnotation to describe service needs, which are not yet explicitly part of the meta model.
When integrating application SW-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 SW-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). It is not in the scope of this document, to specify a relation between the ServiceNeeds and the ECU configuration parameters, for further information see [14].

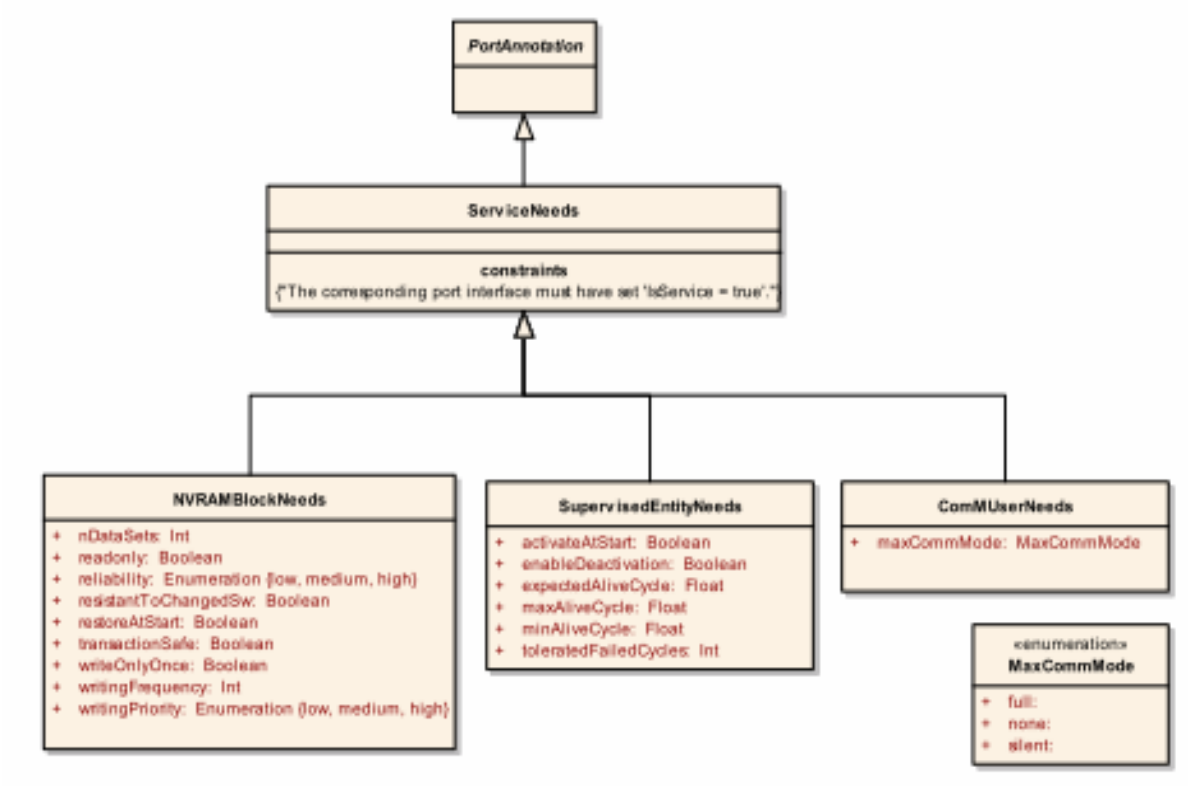

Figure 30: ServiceNeeds.

Class	ServiceNeeds
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes
Class Description	Annotation to a service port of an atomic SWC. It expresses the needs the SWC has on the configuration of the infrastructure (AUTOSAR Service, ECU abstraction or Complex Driver) to which it will be connected via this port.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PortAnnotation

4.6.5.1 Service Needs for the NvRam Manager

Class	NVRAMBlockNeeds
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes
Class Description	Specifies the SWCs needs on the configuration of an NVRAM block associated with this specific Service Port. 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 The needs for configuring callbacks can also be derived from the existence of additional ports required for that.

	The optional association of a RAM mirror block and/or a ROM default block with this NVRAM block is modeled by references from the corresponding elements of the InternalBehavior to this port annotation.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PortAnnotation, ServiceNeeds			
Attribute	Datatype	Mul.	Link Type	Attribute 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	NVRAMBlockNeeds-ReliabilityEnum	1	aggregation	Reliability against data loss on the non-volatile medium. low: Data loss is uncritical medium: Data loss should be avoided high: Data loss is critical These requirements give only a relative indication, for example on the required degree of redundancy for storage. They do however not specify by which means (e.g. software or hardware) the reliability is actually achieved.
resistantToChangedSw	Boolean	1	aggregation	Defines whether a NVRAM 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.
transactionSafe	Boolean	1	aggregation	true: If the current write access fails, the last data of this block is still available and stored on the non volatile medium. false: If the current write access fails, the current and/or last data of this block may be lost.
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	NVRAMBlockNeeds-WritingPriorityEnum	1	aggregation	Requires the priority of writing this block in case of concurrent requests to write other blocks.

4.6.5.2 Service Needs for the Watchdog Manager

Class	SupervisedEntityNeeds			
Package	M2::AUTOSARTemplates::SWComponentTemplate::ApplicationAttributes			
Class Description	Specifies the SWCs needs on the configuration of the Watchdog Manager for the Supervised Entity (SE) associated with this specific Service Port.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PortAnnotation, ServiceNeeds			
Attribute	Datatype	Mul.	Link Type	Attribute 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.

4.6.5.3 Service Needs for the Communication Mode Manager

4.7 Compatibility

4.7.1 Overview

In order to connect ports of software components, the compatibility of those ports' interfaces needs to be verified. This section will define the basic[13] rules, how a formal compatibility assessment can be achieved.

[13] *Basic*, because we are not yet allowing the complex extension mechanisms of ISO 11404. Later specification releases will address this.

Compatibility will be defined bottom-up, i.e. first the rules for compatible datatypes are set up, then the rules for the different interface types are derived. Figure 31 shows the classes that are relevant for the interface compatibility discussion.

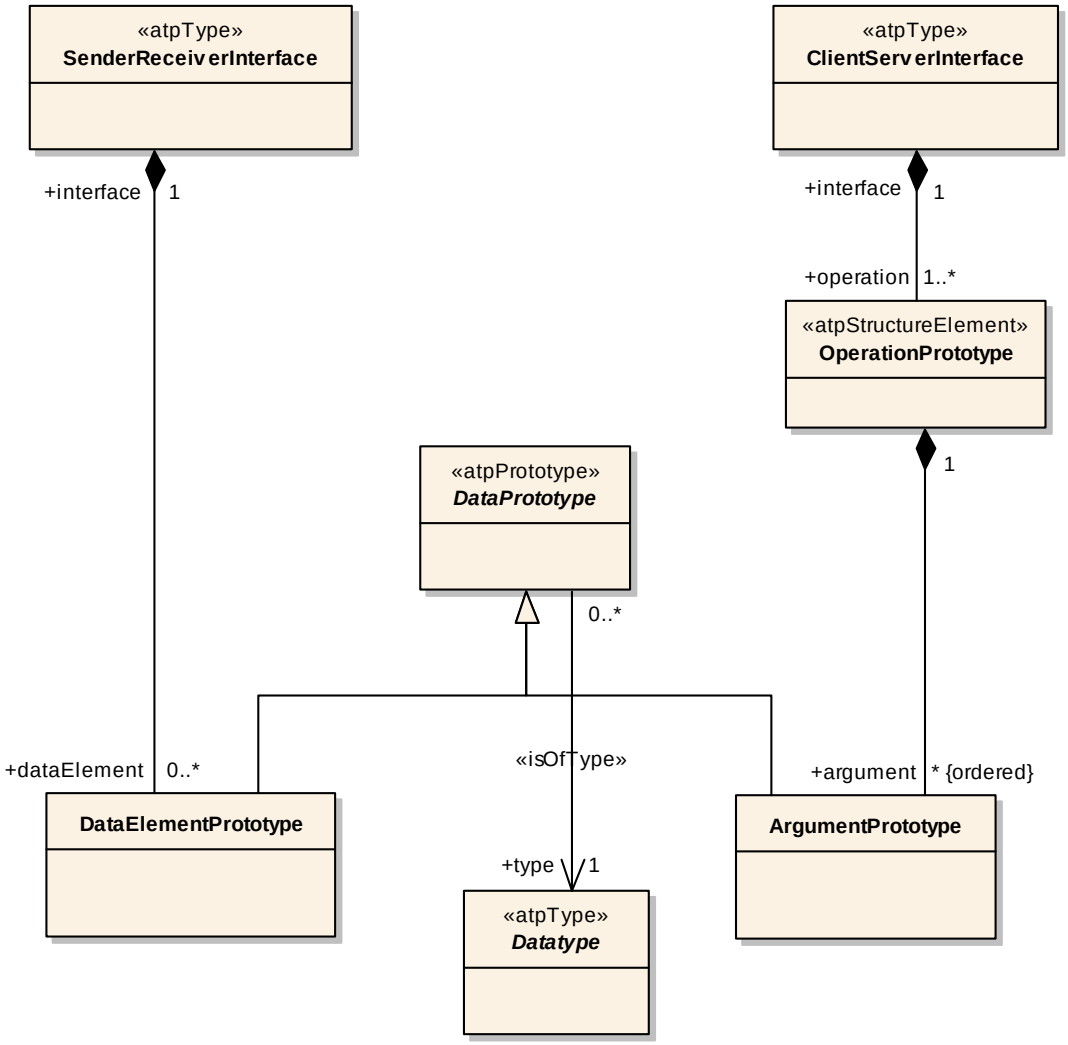

Figure 31: Meta classes relevant for interface compatibility.

4.7.2 Compatibility of Datatypes

To fully discuss compatibility rules for datatypes, the different types and objects in the datatypes meta models have to be cleanly distinguished.
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”). Note the difference between class and instance!
When generally speaking of datatypes (on M1), we are referring to those instances of datatype meta classes (on M2).
AUTOSAR already defines a number of standard datatypes by instantiating a couple of datatype objects.
At this point, no extension of datatypes is planned.

4.7.2.1 Primitive Datatypes

Primitive datatypes are compatible if and only if

1. The examined datatype instances are of the same meta class (type).
2. All attributes match exactly, with one exception: the datatype name[14]. This rule also covers aliases, which by definition differ only in name from the original.
3. The semantics of the datatypes are compatible.

4.7.2.2 Composite Datatypes

Two composed datatypes are compatible if and only if

1. They instantiate the same datatype meta class.
2. They are composed of compatible datatypes (either composed or primitive) in the same order (e.g. for RecordType).
3. All attributes match exactly, with the exception of the datatype name.

4.7.3 Compatibility of Semantics

Primitive datatypes may have associated semantics, e.g. the physical unit they represent. Figure 32 shows the classes in the metamodel that are relevant for comparing semantics.

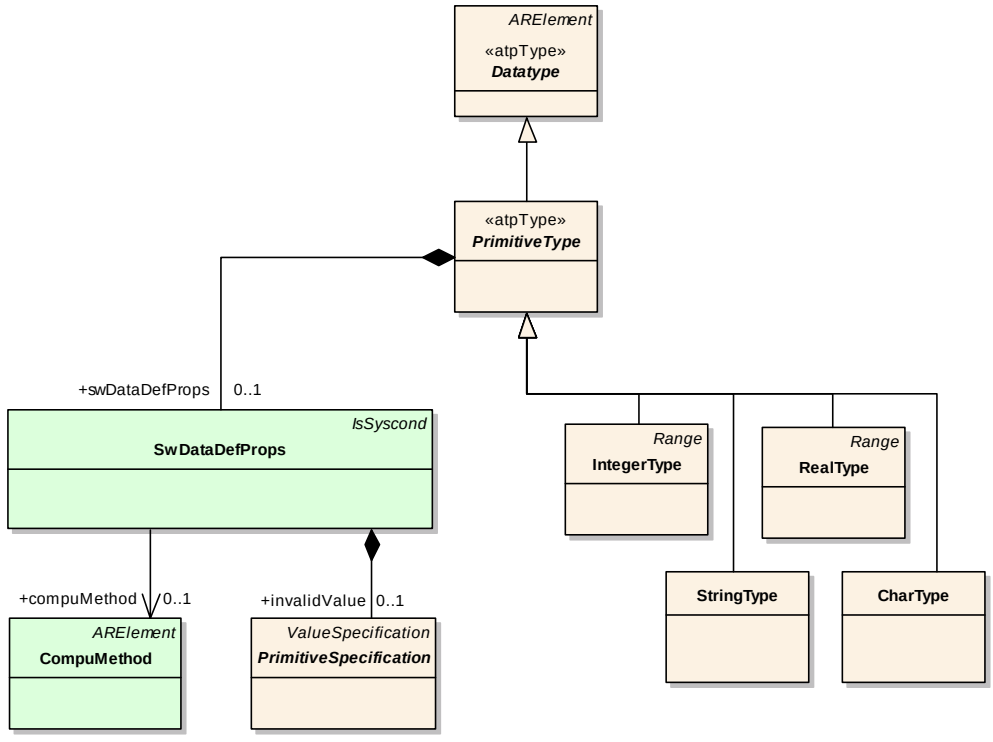

Figure 32: Classes relevant for compatibility of semantics

[14] Byte order is not an attribute of any datatype but of the underlying processor architecture.

A primitive type may have semantics associated (in form of class “CompuMethod”). As the diagram shows, semantics consist of three main characteristics that all need to be compatible to satisfy the overall compatibility requirement. This is automatically the case if both datatypes refer to the same semantics object. in addition to the semantics defined through the “CompuMethod”, also an “invalidValue” can be specified. Semantics of primitive data types 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 has such a method associated.
3. They contain identical conversion methods “compuInternalToPhys” from internal to physical values, or neither of them has such a method associated.
4. They contain the same “invalidValue”, if any.

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

4.7.4 Compatibility of Unit

Figure 24 shows the metamodel class of “Unit” which may optionally reference a “physicalDimension”. Two “Unit” definitions are compatible if and only if:

1. Their have identical “shortName”s.
2. They have identical attributes “factorSiToUnit” and “offsetSiToUnit”.
3. They either refer to identical definitions of “PhysicalDimension” or neither of them has a “PhysicalDimension” associated.

Two “PhysicalDimension” definitions are identical if they have identical “shortName”s and attributes.

4.7.5 Compatibility of Data Elements

Though data elements typically do not exist stand-alone, they are discussed separately to enable the following assessment of interface compatibility.
Two data elements are compatible if and only if

1. They are typed by (read “refer to”) compatible datatypes.
2. They have identical names. This is required to map data elements in unordered interfaces (see section 4.7.7).
3. For each such pair, the values of their ‘isQueued” attributes are equal.

4.7.6 Compatibility of Mode Groups

Mode groups, typed by the meta model class “ModeDeclarationGroup” are compatible if and only if

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

4.7.7 Compatibility of Sender/Receiver Interfaces

The compatibility of sender/receiver interfaces is considered for connection of PortPrototypes. PortPrototypes, of two distinct S/R interface types, are compatible to be connected only if for the two S/R interfaces the following rules apply

1. For each data element present in the S/R interface of the *required* PortPrototype, there exists one compatible data element in the S/R interface of the *provided* PortPrototype. The data element names are used to identify the pair.
2. For each mode group present in the S/R interface of the *required* PortPrototype, there exists one compatible mode group in the S/R interface of the *provided* PortPrototype. The mode group names are used to identify the pair.
3. For each such pair, the values of their “isService” attributes are equal.

This compatibility requirement currently 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).

4.7.8 Compatibility of Operations’ Arguments

Two arguments of two operations are compatible if and only if

1. They are typed by compatible datatypes.
2. They have the same direction (“in”, “out” or “inout”).

4.7.9 Compatibility of Application Errors

Two application errors are compatible if and only if

1. They have the same name.
2. They have the same attributes. Especially the error-code must be identical in both errors.

4.7.10 Compatibility of Operations

Two operations are compatible if their signatures match. I.e. they are compatible if and only if

1. They have the same number of operation arguments.
2. The n-th arguments of both operations are compatible. This implies ordering of operation arguments.
3. They have the same name (again allows for mapping in interfaces).
4. The required operation specifies a compatible application error for each application error that is possibly raised by the provided operation, maybe more.

4.7.11 Compatibility of Client/Server Interfaces

C/S interfaces are compatible if and only if

1. For each *required* operation, a compatible *provided* operation exists. The pair is found by matching operation names.
2. For each such pair, the values of their “isService” attributes are equal.

This compatibility requirement currently 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).

5 Modes

In general the Software Component template doesn't define the kind of modes, which must be supported by State Managers or Components explicitly. However the Software Component template provides generic mechanisms for describing modes. In this section the general relationship between modes, interfaces and SW-Components is discussed.
The assumption from the SW-Component point of view is that State Managers are using a Standardized AUTOSAR Interface[15] to influence the SW-Component and also provide an interface to get requests and confirmations from the SW-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 SW-Components.

5.1 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 initial mode is active before any mode switches occured.This is shown in Figure 33.

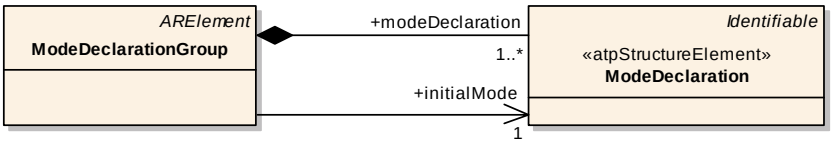

Figure 33: Mode Declaration

The class "ModeDeclarationGroup" has been introduced to support the grouping of modes and to provide predefined sets of modes that could be standardized and reused.

5.2 Communication of Modes

The SW-Component template describes the communication of modes similar to the communication of data elements: The collections of modes that are required or provided by a "SoftwareComponentType" are defined through its interfaces as shown in Figure 34.
This allows for explicitly defining connectors which communicate modes between "ComponentPrototype" and to define service interfaces for communication with the

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

basic software. Due to the compatibility rules of Interfaces (see chapter 4.6.5.3) each software component can rely on the availabilty of required mode activations.

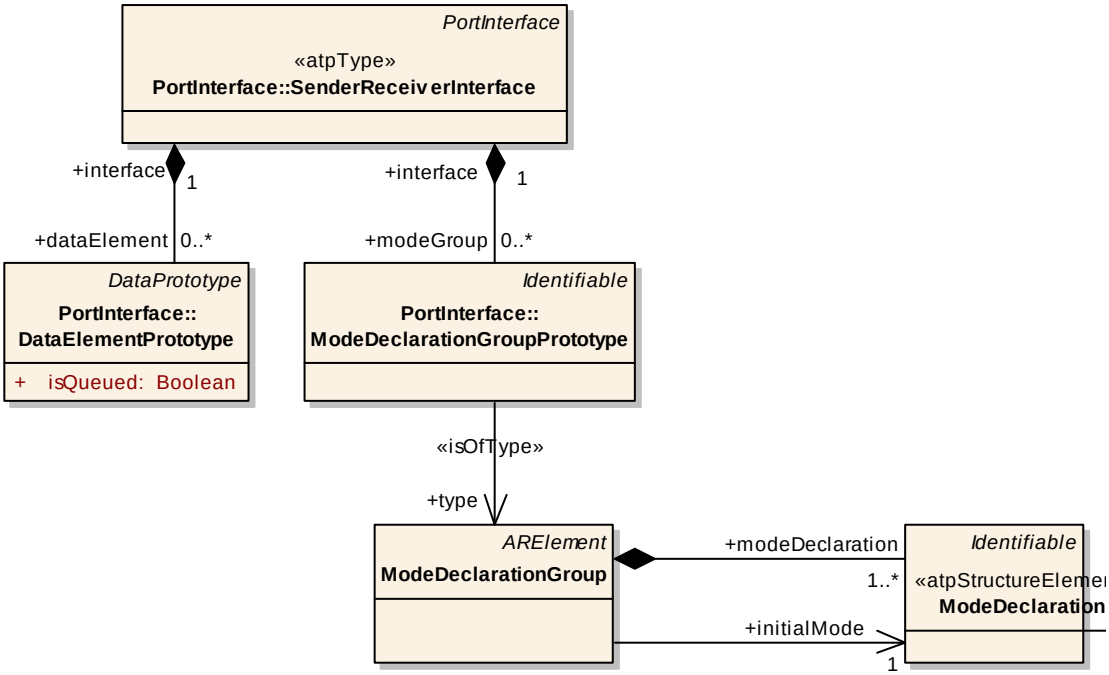

Figure 34: Communication of modes

Please note, that each "ComponentType" - AtomicSoftwareComponents as well as Compositions - can provide (via their ports and interfaces) a list of required and provided "ModeDeclarationGroup"s.
The composition requires and provides the modes that are required or provided by its contained AtomicSoftwareComponents. The delegation of these modes from component prototypes to the containing composition is explicitly described by delegation connectors.
The Software-Component description does not make any assumptions about the semantics of the required and provided modes. It just requires and provides the modes by name.

5.3 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 SW-Component is shown in Figure 35. Since the behavior of SW-Components is mainly determined by the runnable entities contained in the SW-Component it is necessary to configure the response to mode changes on the level of runnable entities.

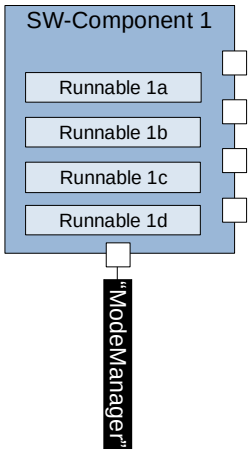

Figure 35: State Managers and SW-Component

Figure 36 shows an excerpt of the meta-model illustrating how the relationship between the current mode and the internal behavior of the SW-Component can be described.

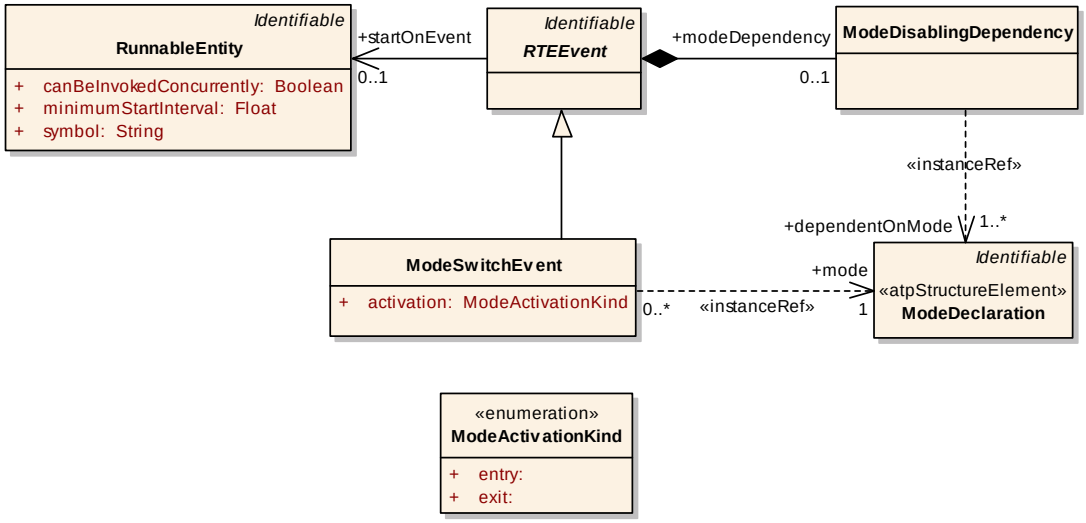

Figure 36: Modes and Events

The SW-Component can use two mechanisms to define how its internal behavior should depend on the mode.
Using the first mechanism (ModeSwitchEvent), the SW-Component can define an event to specify that a specific runnable must be started whenever a mode is entered or exited.
Using the second mechanism (ModeDisablingDependency), the SW-Component can indicate whether an RTEEvent that starts an associated Runnable 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 Runnable is *not* started when the mode referenced by the ModeDisablingDependency is active.

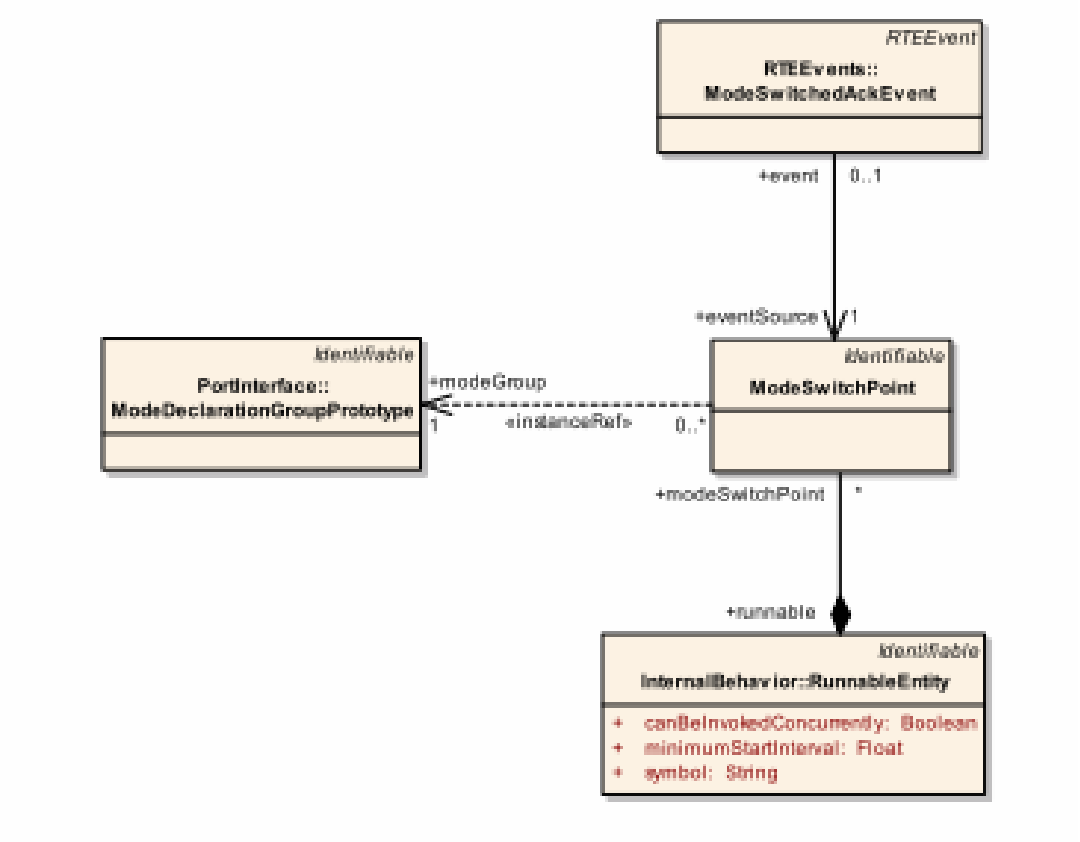

Figure 37: ModeSwitchEvent

A “RunnableEntity” can also have “ModeSwitchPoints” that specified, that a runnable has explicitly received a certain mode.

Class	ModeSwitchPoint			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::ModeDeclaration-Group			
Class Description	A mode switch point specifies that a runnable explicitly received a certain mode.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
modeGroup	ModeSwitchPoint_modeGroup	1	reference to instance	

The "ModeSwitchPoint" also allows for the definition of a "ModeSwitchedAckEvent".. This event is raised when the referenced mode have been received or an error occurs.

Class	ModeSwitchedAckEvent
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents
Class Description	The event is raised when the referenced mode have been received or an error occurs.

Base Class(es)	Identifiable, RTEEvent			
Attribute	Datatype	Mul.	Link Type	Attribute Description
eventSource	ModeSwitchPoint	1	reference	Mode switch point that triggers the event.

5.4 Initialization / Finalization

AUTOSAR must support the execution of initialization code for every SW component. Most SW components 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.
AUTOSAR must also support the execution of finalization code for every SW component. Most SW components 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 mode change driven activation of runnable entities it is easily possible to define a mode "Initialization". When "Entering" this state initialization runnable entities can be activated. When all initialization runnable entities have finished the mode manager can change to further modes.
Also the equivalent can be realized for the finalization of SW-Components.
Please note: The initial modes of SW components 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.

5.5 Summary Meta-Model Excerpt Related to Modes

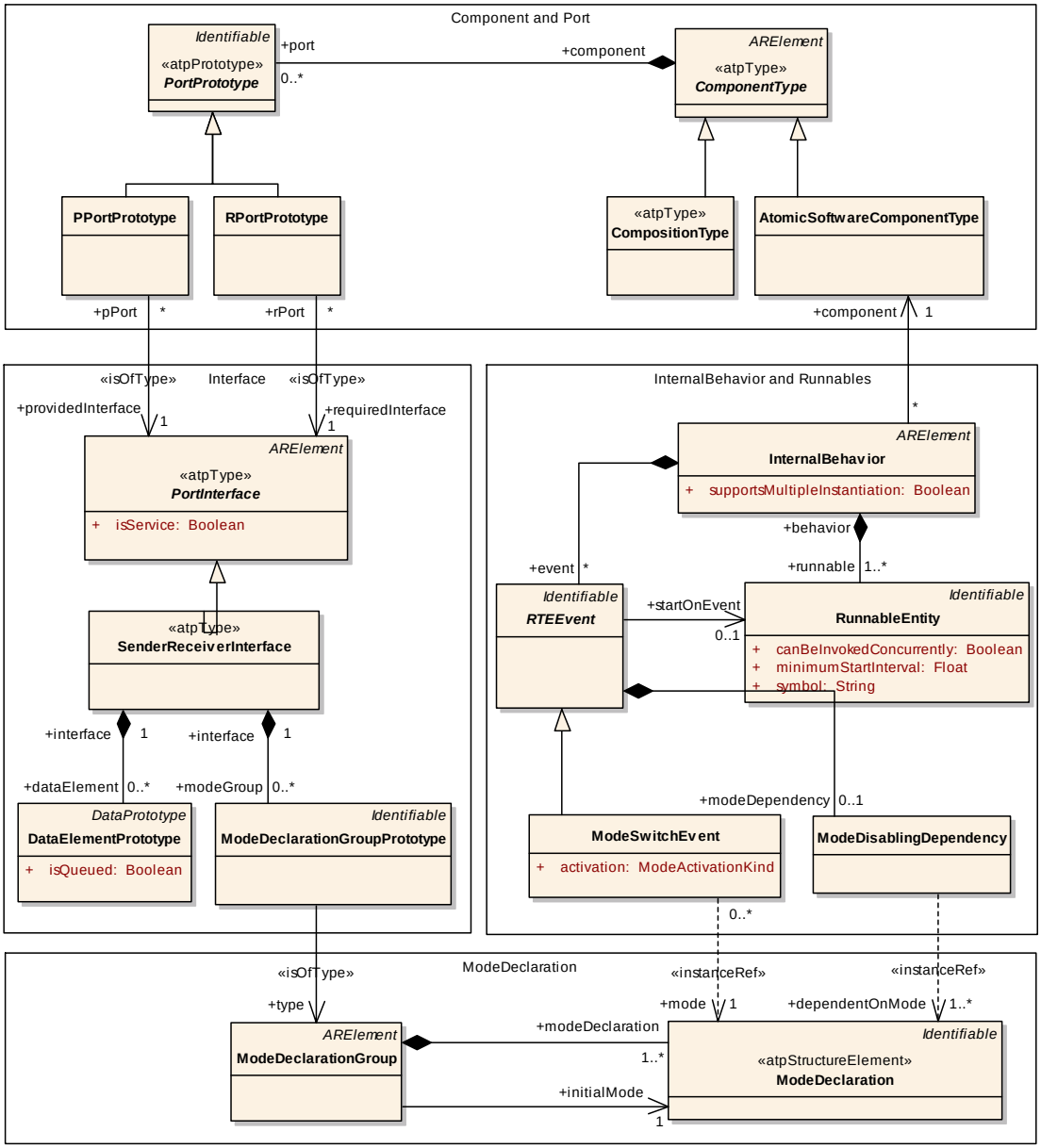

Figure 38: Summary meta-model excerpt related to modes

6 Measurement & 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. Therefore, the class tables contain sometimes more structural features than currently used in AUTOSAR. 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.
With this approach, all information necessary to generate appropriate A2L and ASAP2 files, which can be read by commercial measurement & calibration tools, is available.

6.1 Basic Idea

A calibration paramter (sometimes also called “characteristic value”) 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 “SwComponentPrototypes” (using the port-/interface-concept).

6.2 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 indicate all properties of a particular data element, e.g. how a “DataPrototype” can be measured or a parameter can be calibrated.
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. For details on SwDataDefProps on type level, see 4.5.2 Data Types with Semantics.
In AUTOSAR “SwDataDefProps” are attached to derivations of “DataPrototypes”, namely

1. DataElementPrototypes and ArgumentPrototypes in their respective context of PortPrototypes and ComponentPrototypes.
2. InterrunnableVariable,
3. PerInstanceMemory Variable, and
4. CalprmElementPrototype

to set the “swCalibrationAccess” to READ resp. READ-WRITE in the first three cases or to define the properties of Calibration Parameters in case four.

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.

Class	SwDataDefProps			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::DataDefProperties			
Class Description	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> .			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
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	An address schematic <swAddrMethod> describes how variables or parameters are addressed in the ECU. Address methods can take the form of direct addressing or indirect addressing using a vector. The principle role of this element is the declaration of keywords (in the subelement <shortName>) for the address calculation process. The process itself is given a verbal description (in <swAddrMethodDesc>) rather than a formal one. However, a formal reference <swCpuMemSegRef> to the memory segment concerned is issued. In future versions further formal components may be added to this, if required. . This approach was adopted as the number of possible address schematics is relatively small. A complete formal description would create unnecessary work.
baseType	SwBaseType	0..1	reference	

swBitRepresentation	SwBitRepresentation	0..1	aggregation	Description of the structure of a bit variable: comprises of the <bitPosition> in a memory object (e.g. <swHostVariable>, which stands parallel to <swBitRepresentation>) and the <numberOfBits> . In this way, interrelated memory areas can be described. Non-related memory areas are not supported.
swCalibrationAccess	SwCalibration-AccessEnum	0..1	aggregation	Describes the applicability of parameters and variables. Valid values are: * NOT-ACCESSIBLE : The element will not appear in an ASAP file. * READ-ONLY : The element will only appear as read-only in an ASAP file. * READ-WRITE : Both read and write access.
swValueBlockSize	SwArraysize	0..1	aggregation	<swValueBlockSize> is used to specify the size of a VAL_BLK
swCalprmAxisSet	SwCalprmAxisSet	0..1	aggregation	This element specifies the input parameter axes (abscissas) of parameters (and variables, if these are used adaptively).
swTextProps	SwTextProps	0..1	aggregation	
swCodeSyntax	SwCodeSyntax	0..1	reference	Code syntax objects fix the representation of the variables and parameters in the program source file. These code syntax objects are defined centrally in <swCodeSyntaxes> and are connected to variables and parameters by means of <SwCodeSyntaxRef> . The principle role of this element is the declaration of keywords (in the subelement <shortName>) for the code generation procedure. The process itself is given a verbal description (in <swCodeSyntaxDesc>) rather than a formal one. This approach was adopted as the number of possible code syntax schematics is relatively small. A complete formal description would create unnecessary work. It is possible however, to store a

				process-dependent micro syntax in <swCodeSyntaxDesc>, for example within <verbatim> and to label this through a suitable data assignment to the attribute SI .
compuMethod	CompuMethod	0..1	reference	
dataConstr	DataConstr	0..1	reference	
swDataDependency	SwDataDependency	0..1	aggregation	This element describes the interdependencies of variables and parameters. These can be used for example, to treat virtual parameters.
swHostVariable	SwVariableRef	0..1	aggregation	This element contains a reference <swVariableRef> to a variable, which serves as a host-variable for a bit variable <swBitRepresentation> .
swImplPolicy	SwImplPolicyEnum	0..1	aggregation	<swImplPolicy> enables the specification of an implementation strategy. In this way, the direction of the implementation to come, can be specified in advance in an earlier phase. The contents of the element comprise of a key word which is to be declared in the process. Examples of possible values are * MEMORY_OPTIMIZED * CPU-OPTIMIZED * INPLACE * SINGLE-INSTANCE * MULTI-INSTANCE
swPointer	SwPointer	0..1	aggregation	This element indicates, that the data object (which is specified by the parent element) is a reference to another data object. The properties of the referred data object are described in the <swDataDefProps> contained in the <swPointer> .
swRecordLayout	SwRecordLayout	0..1	reference	Defines how the data objects (variables, calibration parameters etc.) are to be stored in the ECU memory. As an example, this definition specifies the sequence of axis points in the ECU memory. Iterations through axis values

				are stored within the subelements <swRecordLayoutGroup> . These subelements might be stored embedded.
unit	Unit	0..1	reference	Use <unit> to enter the unit of a parameter.
swVariableAccessImplPolicy	SwVariableAccess-ImplPolicyEnum	0..1	aggregation	

Section 6.3 describes how “SwDataDefProps” are attached to ”DataPrototypes” for measuring purposes while sections 6.4 and6.5 describe the construction of charactistics based on the combination of “SwDataDefProps” with “DataPrototypes”. Section 6.6 describes in which context characteristics can be defined. Finally, sections 6.7, 6.8, and 6.9 show how characteristics are used in “Runnables” and show the link to an actual ECU implementation.

6.3 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. This is reflected by introducing derived data prototypes reflecting the context, namely:

- “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 runnables within a “ComponentPrototype”.

The Measurement corresponds to SW-VARIABLE in ASAM-MDX. Various categories of Measurement 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
BIT	A single bit	swHostVariable as instanceRef of the variable hosting the particular bit
BOOLEAN	A Boolean value	
STRUCTURE	A Structure of Values	Must refer to an RecordType.

		Category within RecordElement must be “VALUE”. DataDefProps within RecordElement must be specified. 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 for Measurement

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. Possible values are: STANDARD – no specific protection measures are taken. Usually applies to innerRunnableVariable MEASUREMENT-POINT – The data element is never read within the ecu software. It is written for measurement only MESSAGE – the access to the measurement must be implemented using protection mechanisms. This mainly applies to

	interRunnableVariables.
unitRef	The physical unit if not specified by the compuMethod

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 setting of “swCalibrationAccess”:

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

The Metamodel has to reflect that “SwDataDefProps” can be set for each resulting[16] instance of “DataPrototype” entities. For example, if a connector references a “SenderReceiverInterface” with a “DataElementPrototype” of “RecordType”, and the “RecordType” consists of two “RecordElements”, there will be two “SwDataDefProps” generated. If this “PortInterface” is referenced twice[17] in different “ConnectorPrototype” entities in an AUTOSAR VFB representation, four “SwDataDefProp” entities have to be generated.
The pattern to measure “DataPrototype” entities is that an arbitrary number of “Measurable” elements can be attached to a “ConnectorPrototype”. The “Measurable” element has an InstanceRef to the associated “DataPrototype”. The actual number of “Measurable” elements is determined by the number of resulting instances minus sharing opportunities[18] given by the actual system mapping. “InterRunnableVariable” elements and “PerInstanceMemoryVariable” elements follow the same pattern. However, both variables are related to the “InternalBehavior”, which means that in case of multiple instantiation of the associated component, the RTE will resolve the instances by using an “InstanceHandle”. This means that the association of the “Measurable” elements, as defined in the authoring tool, has to be attached to the appropriate instance in the RTE-contract phase. Otherwise, the RTE-generator will not be able to evaluate the “SwDataDefProps” entity correctly.
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 contraints than the type but not vice versa. To keep it simple for 2.1 it is recommended to avoid the multiple definition of the same data definition property. For example compuMethod might be defined on type

[16] When the actual instance tree is constructed from the top-level composition
[17] Via its connected “PortPrototype” entities.
[18] In case of a “PPortPrototype”, associated to a sending “ComponentPrototype”, connected to several “ComponentPrototype” entities via “RPortPrototype” entities of their associated “ComponentPrototype” entities, using a dedicated “ConnectorPrototype” for each connection, it is up to the designer to attach either a “SwDataDefProps” entity to every “ConnectorPrototype” or attaching “SwDataDefProps” to just one “ConnectorPrototype” and then trust the system-generator and the RTE-generator that the properties will be evaluated properly in the resulting E/E-Architecture, i.e. the appropriate memory cell will be found. However it is, all definitions must be processed and must be consistent.

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.

Class	Measurable			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-MeasurementProperty			
Class Description	this class assignes data def properties to particular prototypes. This pattern is necessary, since the interrunnable Variable might be duplicated in case the software component referenced by the InternalBehavior is instantiated more than once on the same ecu.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
swDataDefProps	SwDataDefProps	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> .

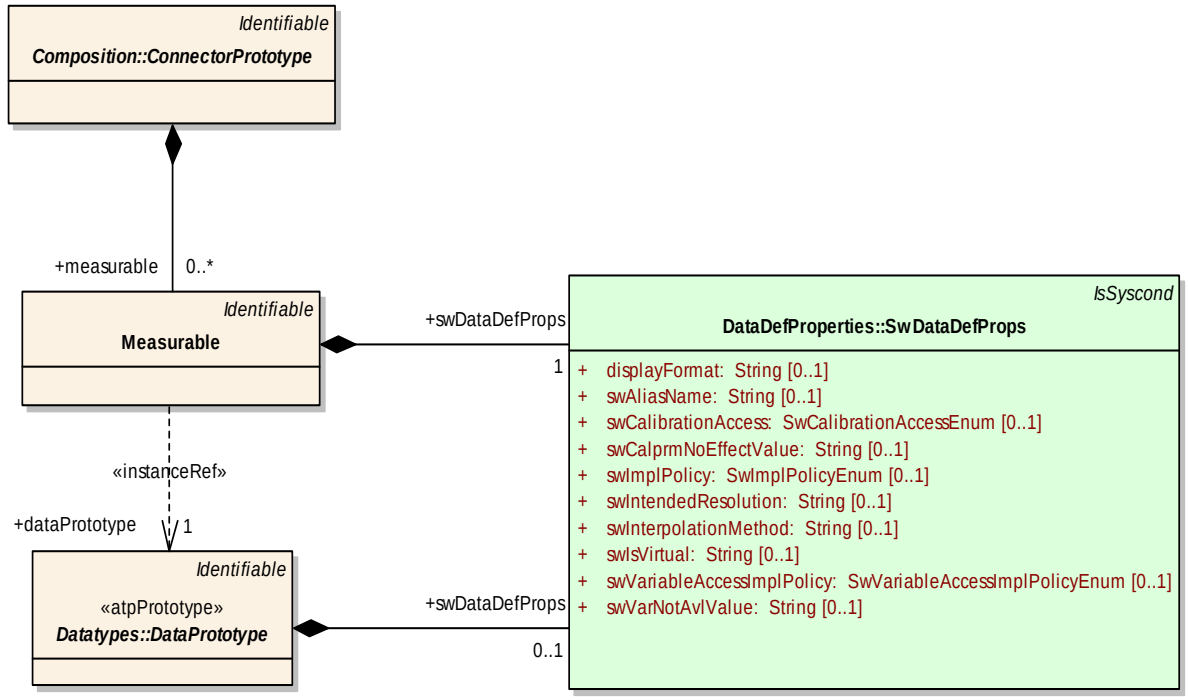

Figure 39: Data-Def-Properties in Connector Context

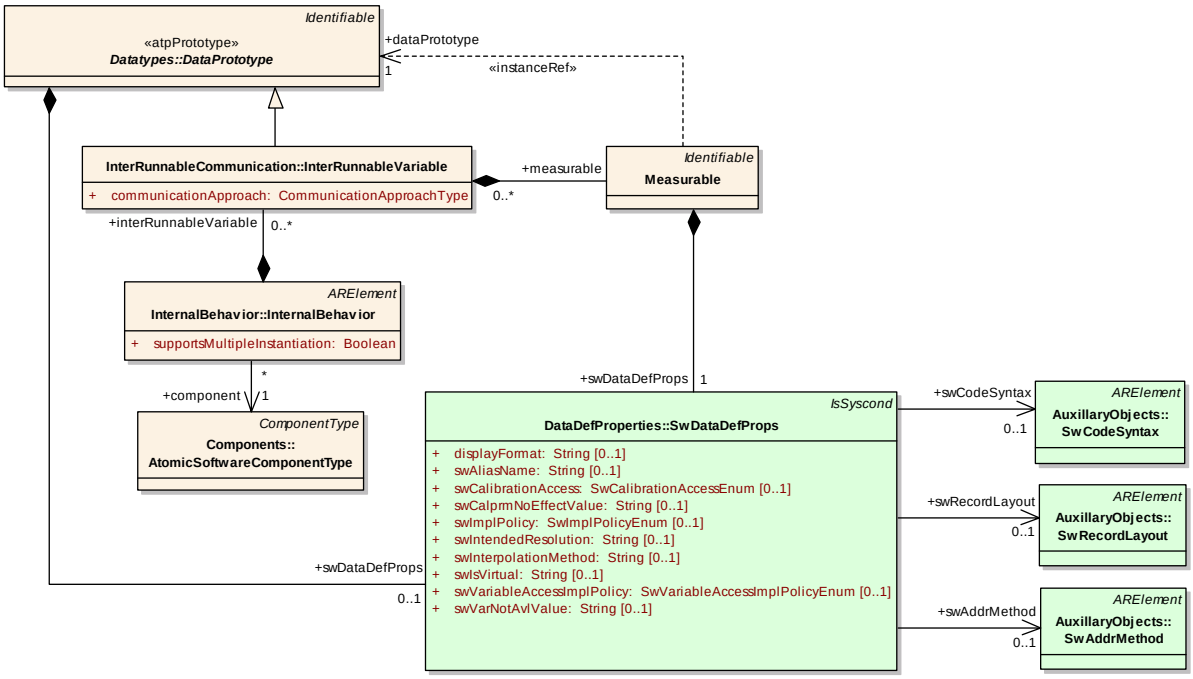

Figure 40: Data-Def-Props in Inter-Runnable-Variable Context

6.4 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 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 according ASAM – MDX categories are shown in Table 1.

ASAM Category	Meaning	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 homgenous 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 a 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 an 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, cause 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 an RecordType. Category within RecordElement must be set accordingly. DataDefProps within RecordElement must be specified. 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 1: ASAM Categories to represent different types of characteristiristics

Section 6.5 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.

6.5 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[19] 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 “SwCalprmAxisAxis” 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 maxmimum 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

[19] The MSR-style of defining a characteristic is described here in an attempt to make the AUTOSAR easier to understand.

when the type of the referenced input value (swVariableRef) has been set. For further details see Section 6.7.

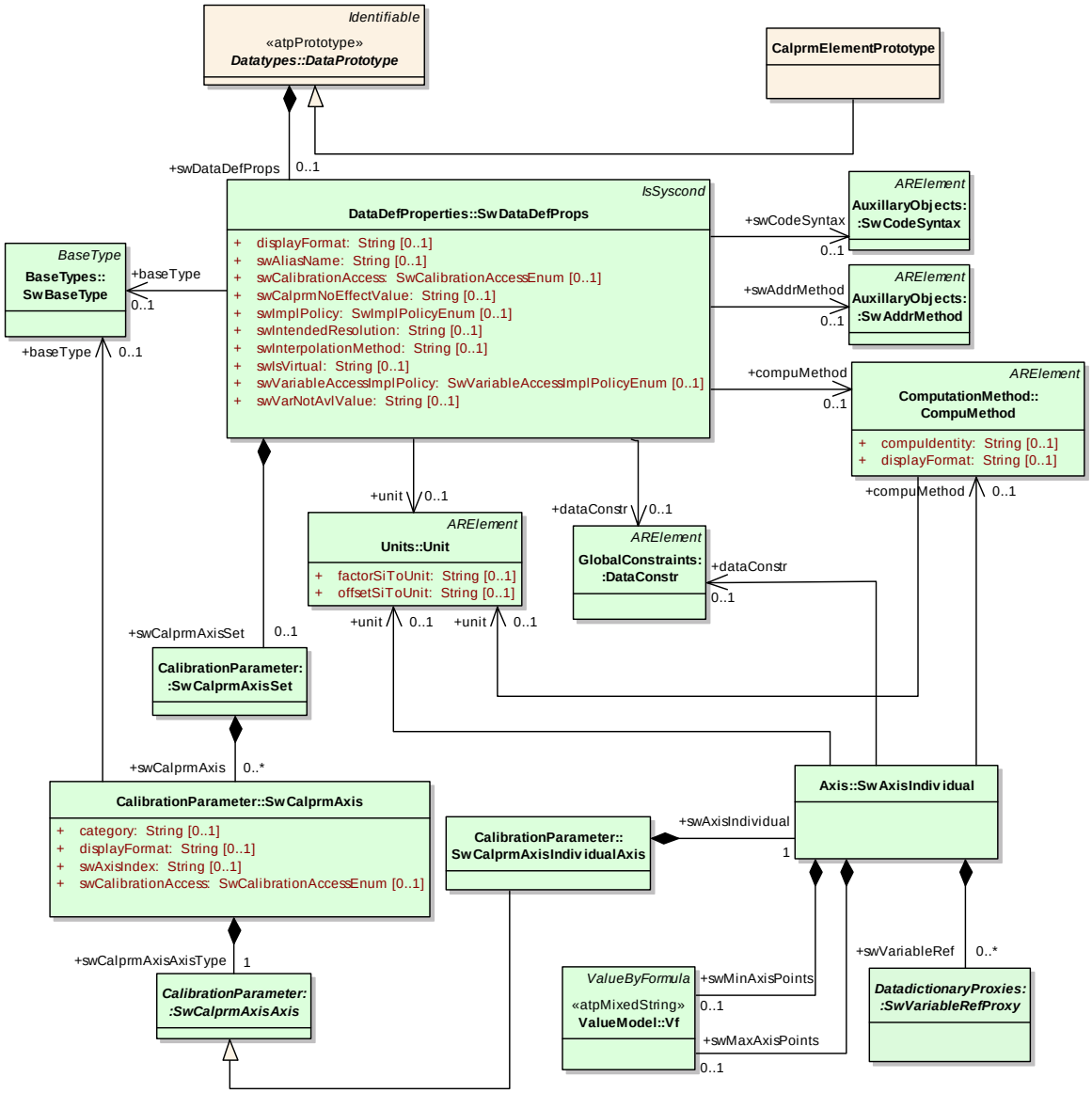

Figure 41: Model of a Curve

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 Figure 41.

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 Figure 16.

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 Figure 42 showing that the baseType can be specified on type and on prototype level.

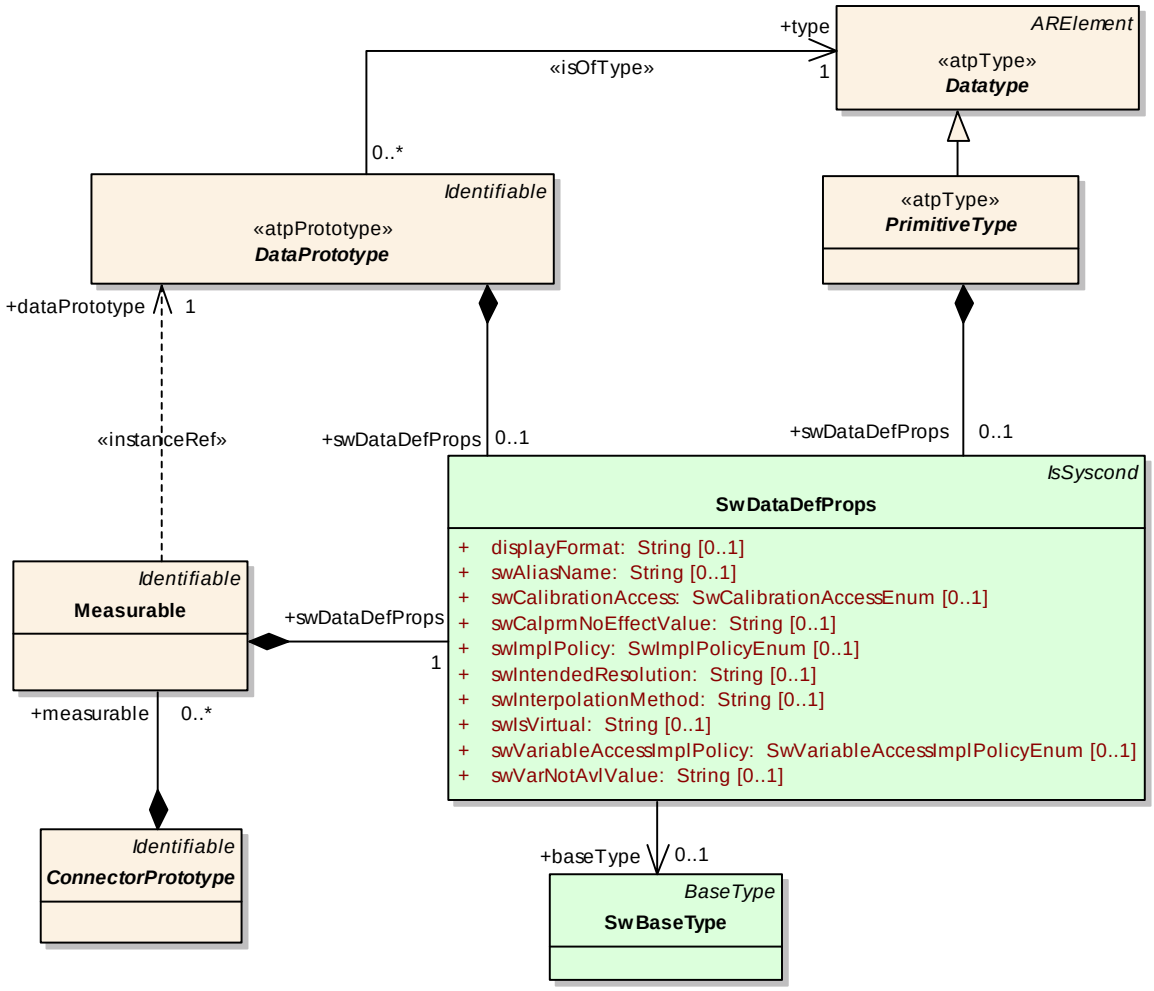

Figure 42: Type Determination of Calibration Data Value axis

For more details see chapter 6.9.

6.6 Using Calibration Parameters

As mentioned above, a CalprmElementPrototype can be used in the context of internal behavior as well as in the context of port prototypes.

6.6.1 Sharing Calibration Parameters within Compositions

This case is based on components, ports, and interfaces. As provider, a dedicated software component called “CalprmPrototypeComponentType”, which is derived from

"ComponentType", has to be used as prototype. This dedicated software component type has no internal behavior and employs exclusively "PPortPrototypes" of type "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 "SoftwareComponentProto-types" is explicitly visible. If a connector spans after the mapping of software "Com-ponentPrototypes" over two different ECUs, the system generation process has to ensure the proper allocation of the "CalprmElementPrototype" while the calibration system has to cope with setting the parameter synchronously.

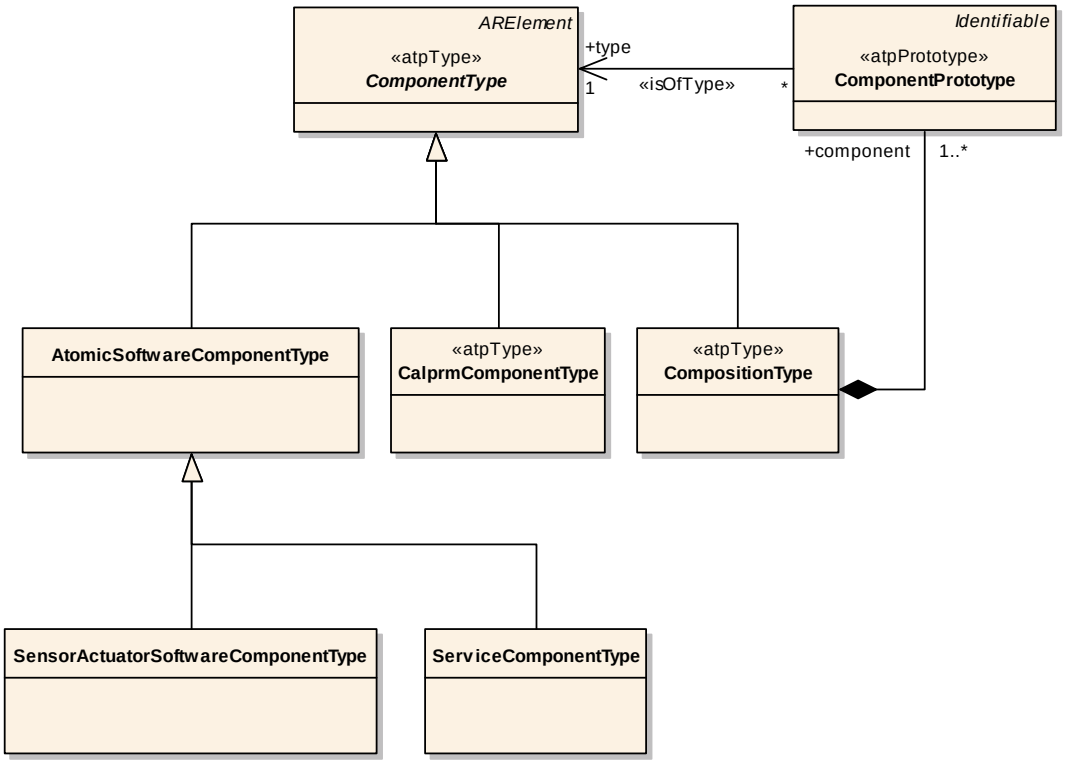

Figure 43: CalprmComponentType

Class	CalprmComponentType
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components
Class Description	
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, ComponentType

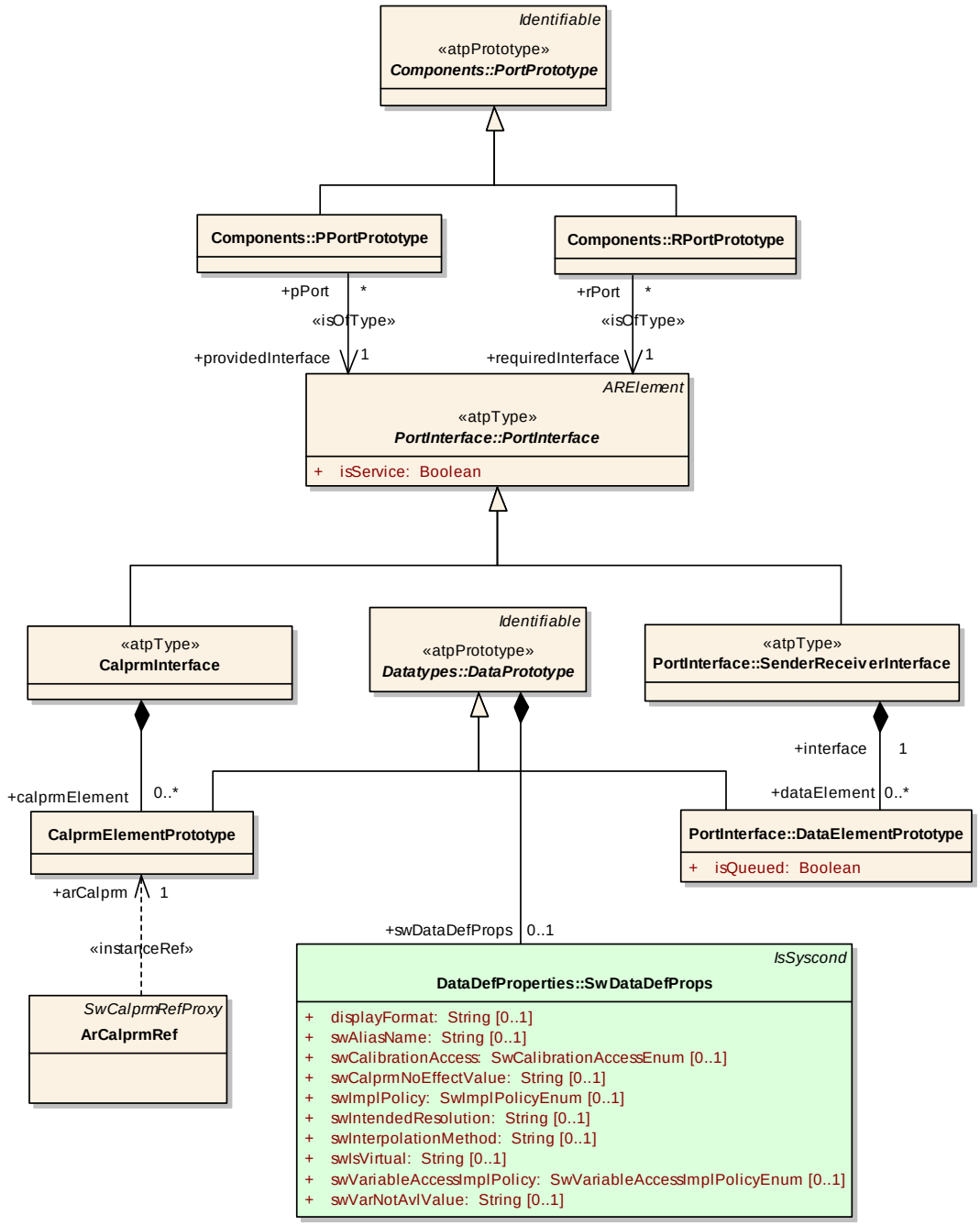

Figure 44: CalprmPrototype

Class	CalprmElementPrototype
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-Characteristic
Class Description	In the context of a component there is no typed re-use of CalprmElementPrototype entities

Base Class(es)	Identifiable, DataPrototype

Class	CalprmInterface			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-Characteristic			
Class Description				
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement, PortInterface			
Attribute	Datatype	Mul.	Link Type	Attribute Description
calprmElement	CalprmElementPrototype	0..*	aggregation	

6.6.2 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 a "SoftwareComponentType", the actual calibration values of the "CalprmElementPortotype" is the same for all prototypes. 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 "SoftwareComponentPrototypes" for each lambda sensor will use the very same Calibration Parameters.

6.6.3 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 "SoftwareComponentPrototype".

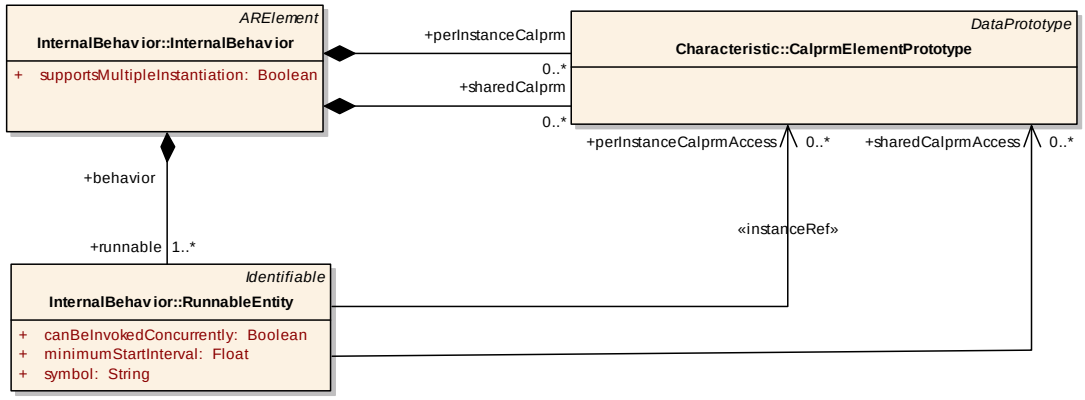

Figure 45: CalprmElementPrototypes in internal behavior

6.7 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”,
- a “PerInstanceMemoryVariable” within the “InternalBehavior”.

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 Figure 46. This “DataPrototypeRefProxy” has an InstanceRef to a “DataPrototype” in the appropriate context.

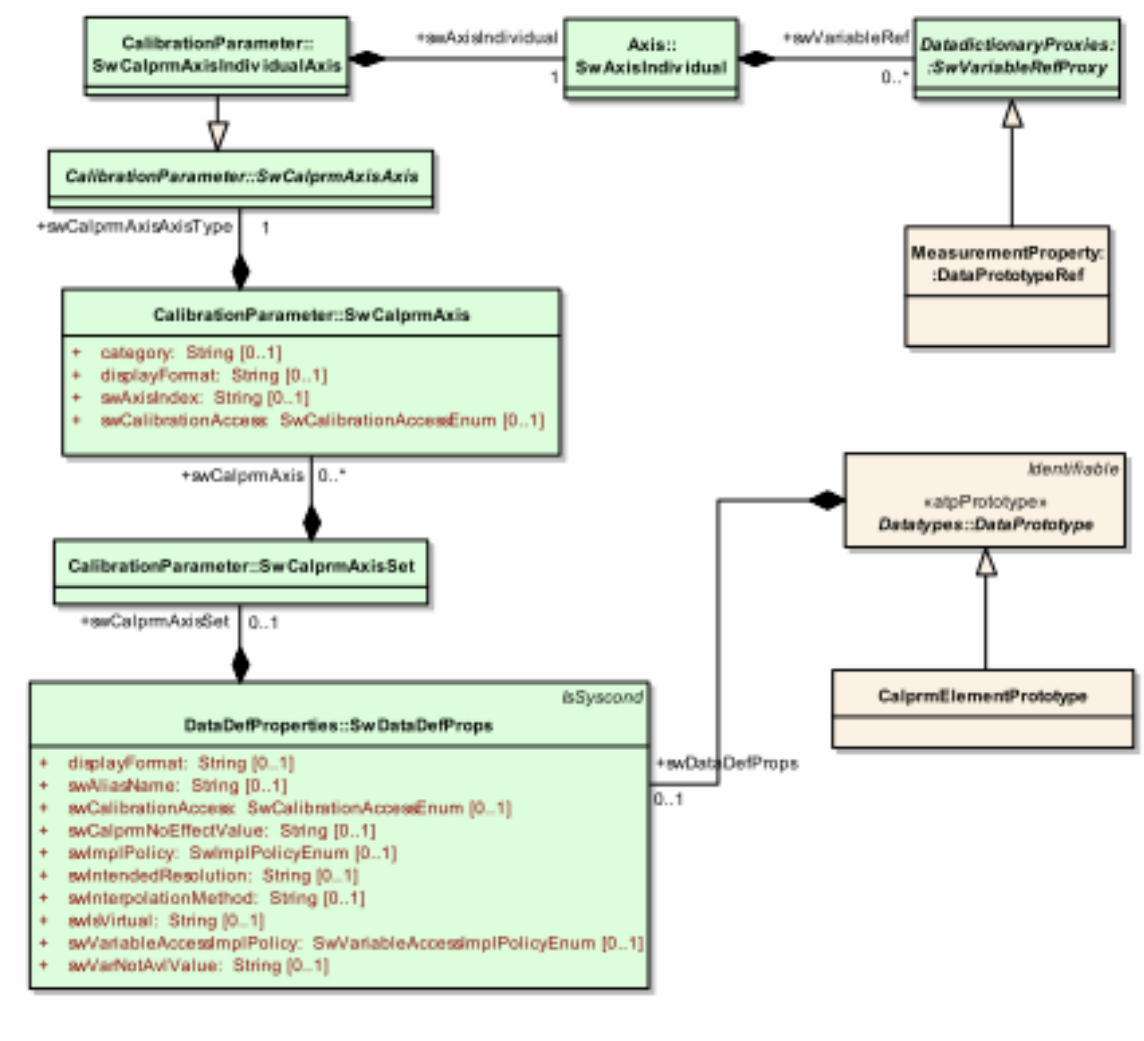

Figure 46: Extended Axis Elements and Input Variable Reference

Class	SwCalprmAxisSet			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-CalibrationParameter			
Class Description	This element specifies the input parameter axes (abscissas) of parameters (and variables, if these are used adaptively).			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
swCalprmAxis	SwCalprmAxis	0..*	aggregation	This element specifies an individual input parameter axis (abscissa). The following belongs to this: * Which axis is involved (<swAxisIndex>) * Whether the axis is integrated (<swAxisIndividual>) or whether it is adopted from another parameter (<swAxisGrouped>). * Whether the axis can be applicated (

				<swCalibrationAccess>). * Which display format should be used for the axis (<SW-DISPLAY-FORMAT>). * Which base type should be used for the axis (<SW-BASE-TYPE-REF>).

Class	SwCalprmAxis			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAnd-Calibration::CalibrationParameter			
Class Description	This element specifies an individual input parameter axis (abscissa). The following belongs to this: * Which axis is involved (<swAxisIndex>) * Whether the axis is integrated (<swAxisIndividual>) or whether it is adopted from another parameter (<swAxisGrouped>). * Whether the axis can be applicated (<swCalibrationAccess>). * Which display format should be used for the axis (<SW-DISPLAY-FORMAT>). * Which base type should be used for the axis (<SW-BASE-TYPE-REF>).			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
swAxisIndex	String	0..1	aggregation	<swAxisIndex> describes the index referring to the axis currently described, for which the contents is specified. The index satisfies the following convention: : : : : :
swCalprmAxisAxisType	SwCalprmAxisAxis	1	aggregation	
swCalibrationAccess	SwCalibrationAccessEnum	0..1	aggregation	Describes the applicability of parameters and variables.
baseType	SwBaseType	0..1	reference	

Class	SwCalprmAxisAxis {abstract}
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-CalibrationParameter
Class Description	
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject

Class	SwCalprmAxisIndividualAxis			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-CalibrationParameter			
Class Description				
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, SwCalprmAxisAxis			
Attribute	Datatype	Mul.	Link Type	Attribute Description
swAxisIndividual	SwAxisIndividual	1	aggregation	This element describes an axis integrated into a parameter (field etc.). The integration makes this individual to each parameter. The so-called group axis represents the counterpart to this. It is conceived as an independent parameter and is referenced by the respective parameters used (<swCalprmRef> within <swAxisGrouped>). The specification of this axis incorporates: * Possible input variables for the axis (<swVariableRefs>). 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-UNIT> s). * 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. * a conversion formula (<SW-COMPU-METHOD-REF>) * a measuring unit (<SW-UNIT-REF>) * a bit representation (<swBitRepresentation>) * Maximum and minimum number of axis base points (<swMaxAxisPoints>, <swMinAxisPoints>) * Plausibility checks (<SW-DATA-CONSTR-REF>) * Generic axis characteristics (<swAxisGeneric>

				), in the event that such an axis is involved. The specifications <swVariableRefs>, <SW-COMPU-METHOD-REF>, <SW-UNIT-REF> 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> * <SW-COMPU-METHOD-REF> * <SW-UNIT-REF>

Class	SwAxisIndividual
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-Axis
Class Description	This element describes an axis integrated into a parameter (field etc.). The integration makes this individual to each parameter. The so-called group axis represents the counterpart to this. It is conceived as an independent parameter and is referenced by the respective parameters used (<swCalprmRef> within <swAxisGrouped>). The specification of this axis incorporates: * Possible input variables for the axis (<swVariableRefs>). 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-UNIT> s). * 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. * a conversion formula (<SW-COMPU-METHOD-REF>) * a measuring unit (<SW-UNIT-REF>) * a bit representation (<swBitRepresentation>) * Maximum and minimum number of axis base points (<swMaxAxisPoints>, <swMinAxisPoints>) * Plausibility checks (<SW-DATA-CONSTR-REF>)

	* Generic axis characteristics (<swAxisGeneric>), in the event that such an axis is involved. The specifications <swVariableRefs>, <SW-COMPU-METHOD-REF>, <SW-UNIT-REF> 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> * <SW-COMPU-METHOD-REF> * <SW-UNIT-REF>			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
swVariableRef	SwVariableRefProxy	0..*	aggregation	This element references <swVariable> .
compuMethod	CompuMethod	0..1	reference	
unit	Unit	0..1	reference	Use <unit> to enter the unit of a parameter.
swMaxAxisPoints	Vf	0..1	aggregation	Maximum number of base points contained in the axis of a map or curve.
swMinAxisPoints	Vf	0..1	aggregation	This element specifies the minimum number of base points on the current axis of a map or curve.
dataConstr	DataConstr	0..1	reference	
swAxisGeneric	SwAxisGeneric	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.

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 Figure 47 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”.

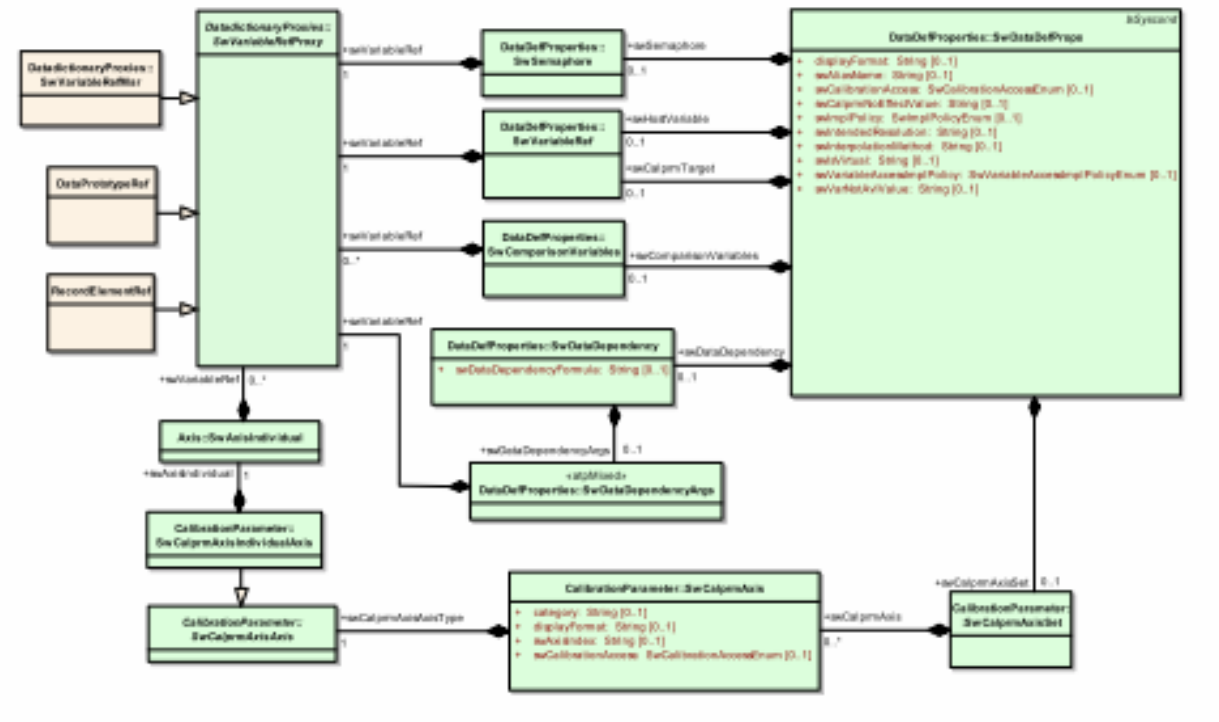

Figure 47: Extended Variable Reference Mechanism

Class	SwVariableRefProxy {abstract}
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-DatadictionaryProxies
Class Description	
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject

Class	DataPrototypeRef
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-MeasurementProperty

Class Description				
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, SwVariableRefProxy			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dataPrototype	DataPrototypeRef_DataPrototype	1	reference to instance	

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 Figure 48, while in the upper middle the MSRSW style is shown, referencing the “SwCalprm”.

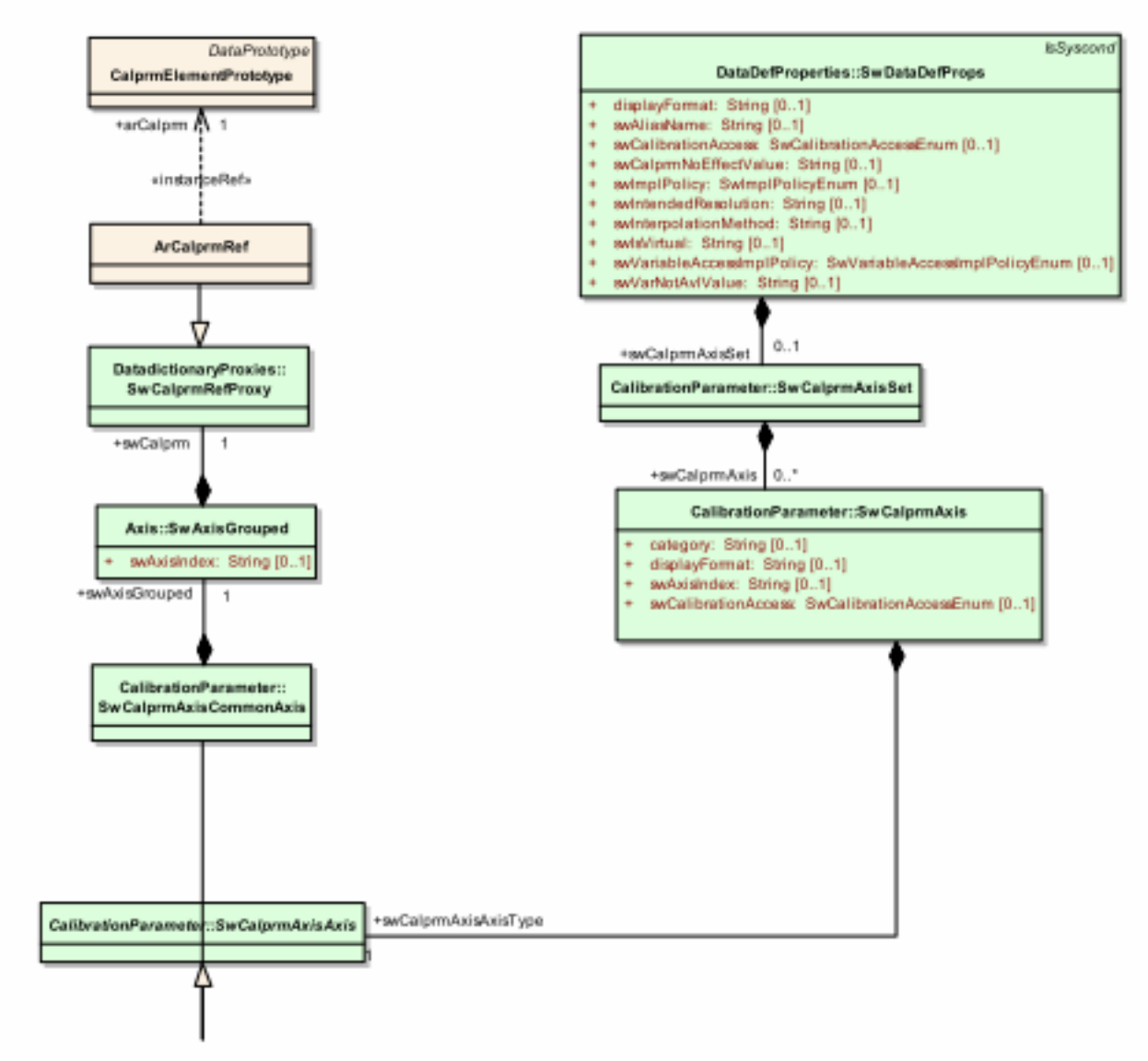

Figure 48: Grouped Curves sharing input values of another CalprmElementPrototype

Class	ArCalprmRef			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::-Characteristic			
Class Description				
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, SwCalprmRefProxy			
Attribute	Datatype	Mul.	Link Type	Attribute Description
arCalprm	ArCalprmRef_CalprmElementPrototype	1	reference to instance	

6.8 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 components a dedicated interface in a port prototype will be used. The designer of a software component can use this access mechanism when designing a runnable using a dataprototype

- from an arbitrary “RPort” associated either with a “ClientServer-“ or a “SenderReceiverInterface”,
- a “InterrunnableVariable”, or
- a “PerInstanceMemoryVariable”

as input value.
This input value will be fed to an interpolation routine whose result can be used internally or transferred to a neighbored component prototype via dedicated port-prototypes. A typical use-case might be that there is a client-server interface where some data will be provided at a call. Typically, there will be a dedicated runnable (with “ReceiveMode” “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 as an “ArgumentPrototype” with “Direction” being either set to “out” or “inout” in a “ClientServerInterface”.

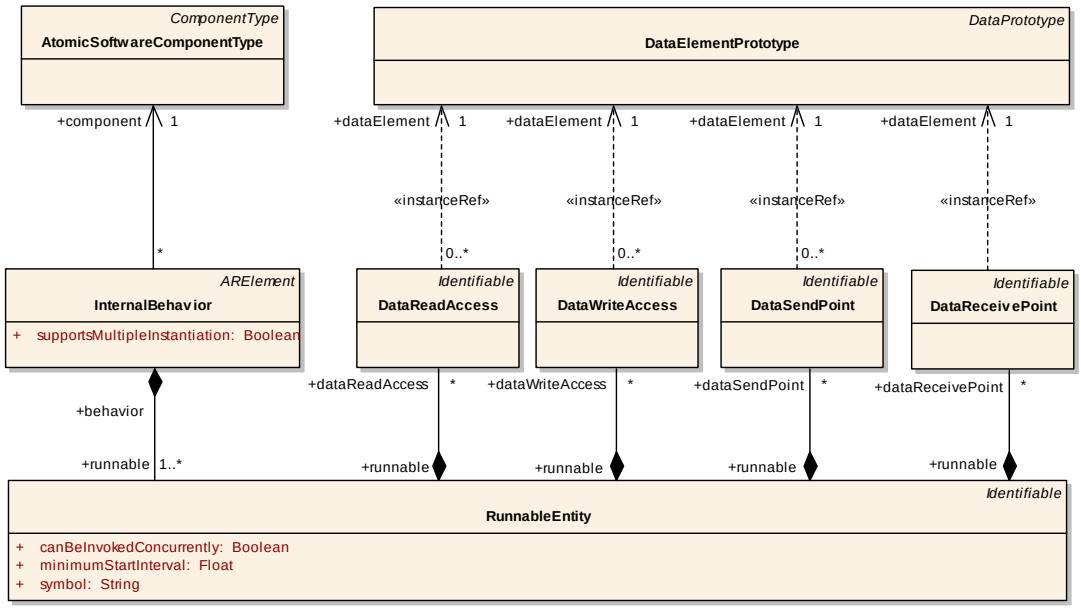

Figure 49: Runnable Access to a Calibration Port

Class	CalprmAccess			
Package	M2::AUTOSARTemplates::SWComponentTemplate::MeasurementAndCalibration::Characteristic			
Class Description				
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
calprmElementPrototype	CalprmAccess_calprmElementPrototype	1	reference to instance	

The access to a “CalprmElementPrototype” will be indicated

- by the “CalprmAccess” entity if the runnable wants to access it from a “RPort-Prototype”. This is shown in Figure 49
- by defining the “sharedCalprmAccess” association from a “Runnable” to the “CalprmElementPrototype”. This is shown in Figure 45 in the lower association from “Runnable” to “CalprmElementPrototype
- by defining the “perInstanceCalprmAccess” association from a “Runnable” to every instance of the “CalprmPrototype”. This is shown in Figure 45 in the upper association from “Runnable” to “CalprmElementPrototype”.

6.9 Addressing Methods

In an ECU there might be various methods to access a particular object (e.g measurment or calibration parameter) according to a given adress. 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 can be 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 SW-C shall be placed in another memory location than the other parameters of the CalprmComponentPrototype respectively the AUTOSAR SW-C. In Autosar 2.1 only the name and an informal description can be specified. Subsequently, memory segments can be associated with SwAdrMethod as it is done in MSRSW.

6.10 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.

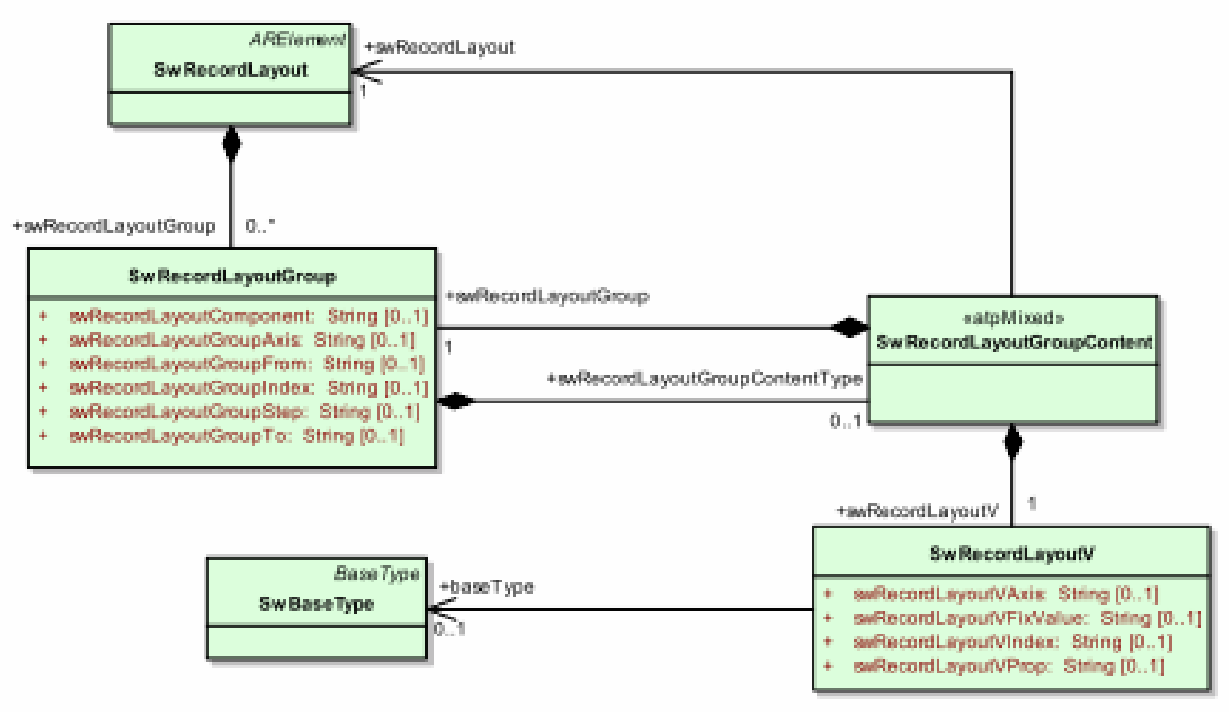

Figure 50: Specification of a record layout

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.
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 swRecordLayoutV elements. Note that this name can also be used to construct names for appropriate datatypes.
- swRecordLayoutGroupFrom specifies the starting point for the interation. 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, be-

cause 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 intended. 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.
- swRercordVIndex 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 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. 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:

VALUE	The value of the axis
COUNT	The amount of values of the axis
LEFTDIFF	The difference to the previous value
RIGHTDIFF	The difference to the next value
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
RESULT-ADR	The address of the result for this axis (if introduced)
ADDRESS	The address
FILL	Fill with the hex value specified as contents of swRecordLayoutFixValue
FIXLEFTDIFF	Difference between this and a fixed left-hand value (normally 0)
FIXRIGHTDIFF	Difference between this and a fixed right-hand value specified in swRecordLayoutFixValue

- 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.

Here you see an example for a SwRecordLayout noted in XML

```
<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>
```

6.11 Record Layouts and Data Types

As DataPrototypes have an isOfType Relation to DataTypes, the related datatypes must properly match to the details as specified in swDataDefProps as shown in the diagram.

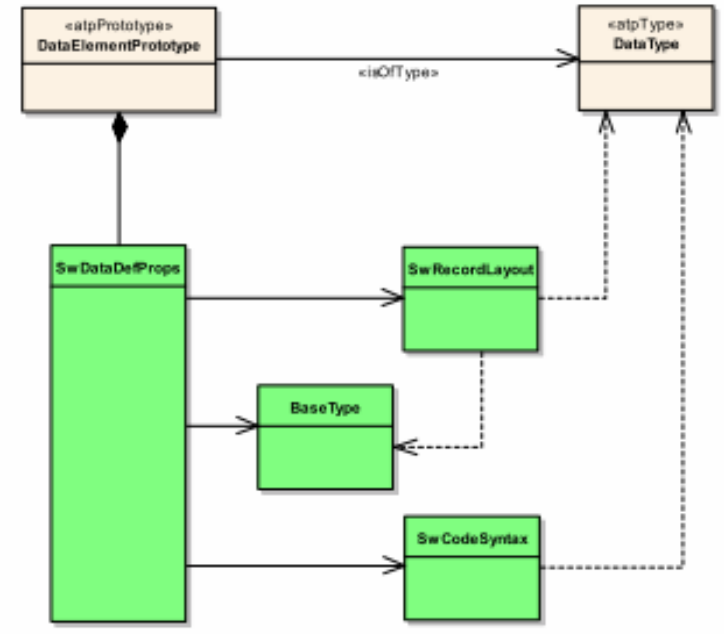

Figure 51: Dependency of DataTypes and RecordLayouts

In order to maintain this compliance there are three approaches

- Manually create DataTypes for the calibration parameters and compatible 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.

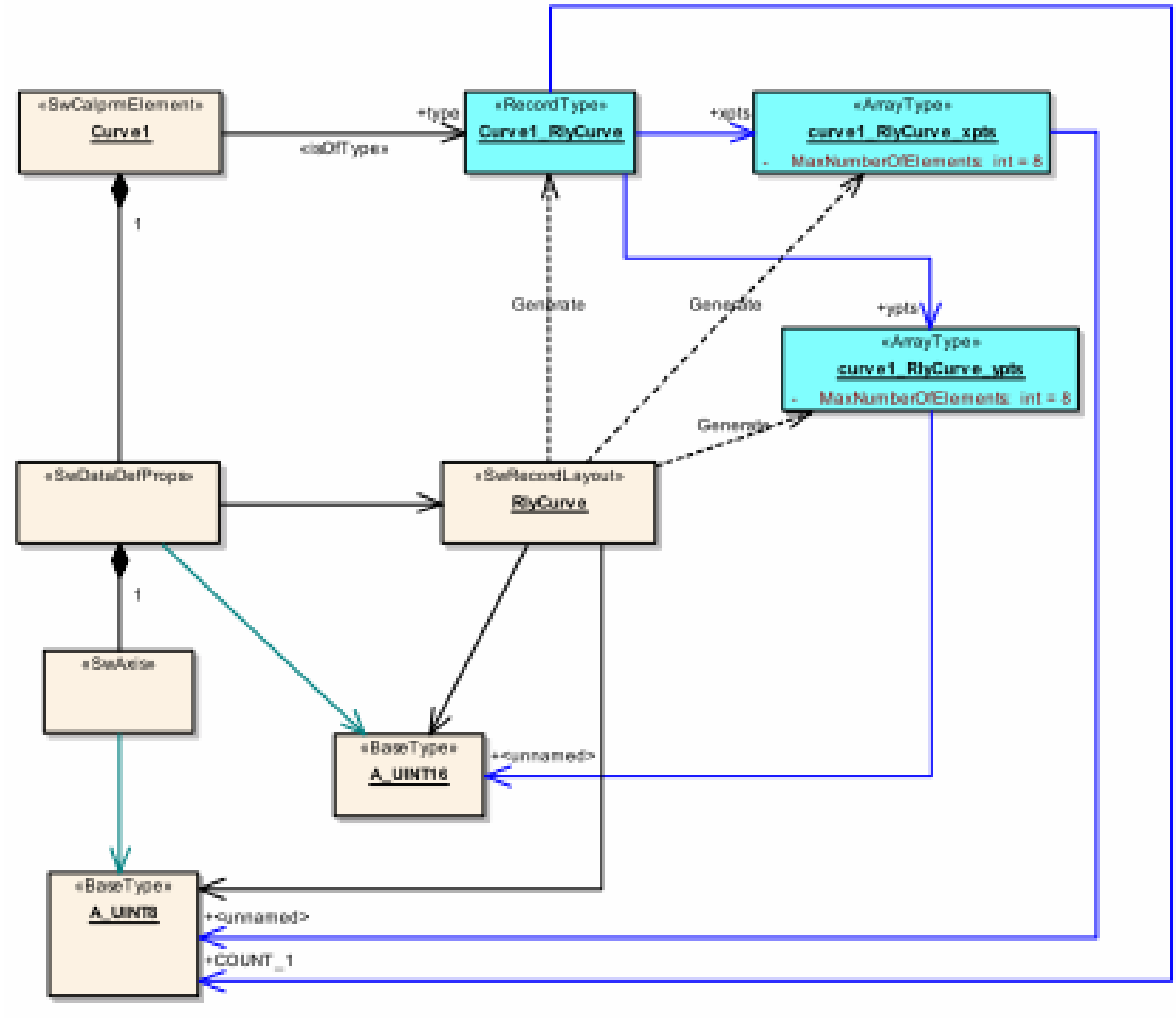

Figure 52: Curve implemented as two consecutive arrays

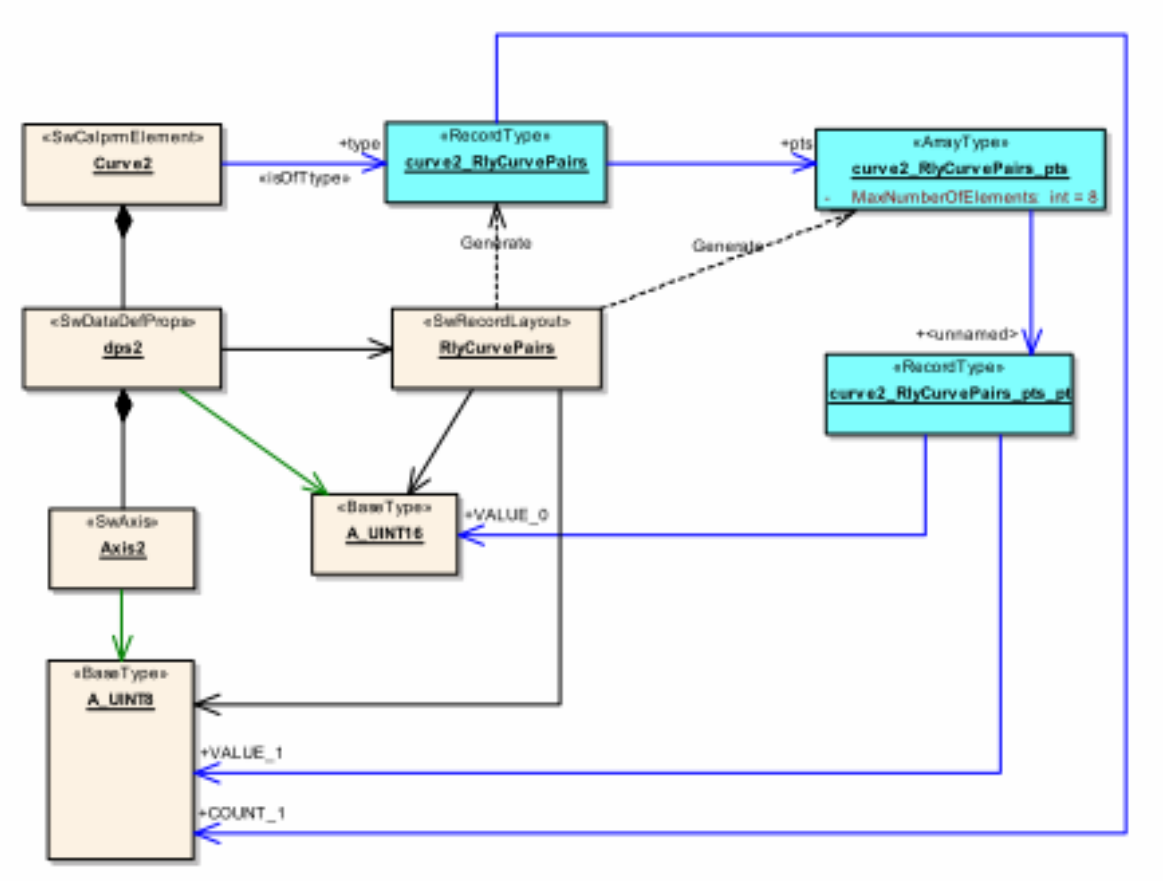

Figure 53: Curve implemented as arry of record

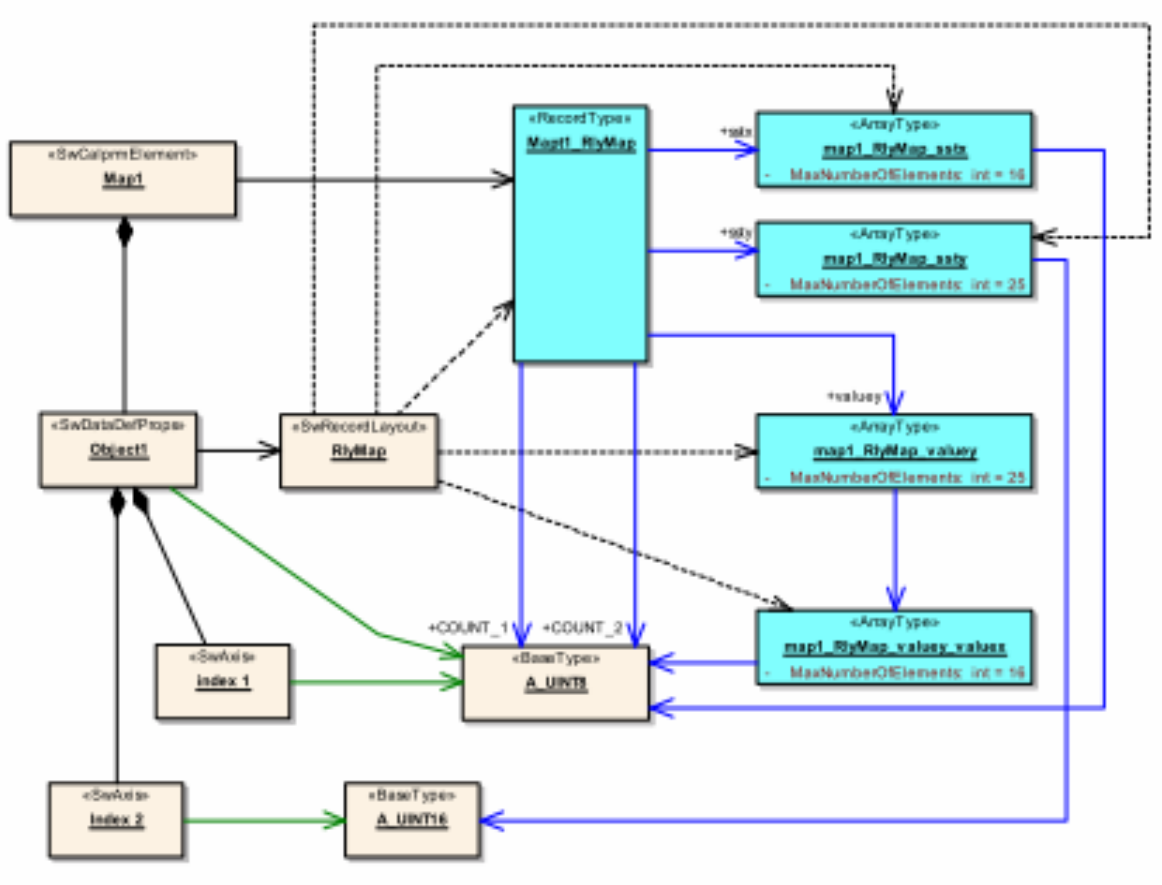

Figure 54: Record layout and data type for a map

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.

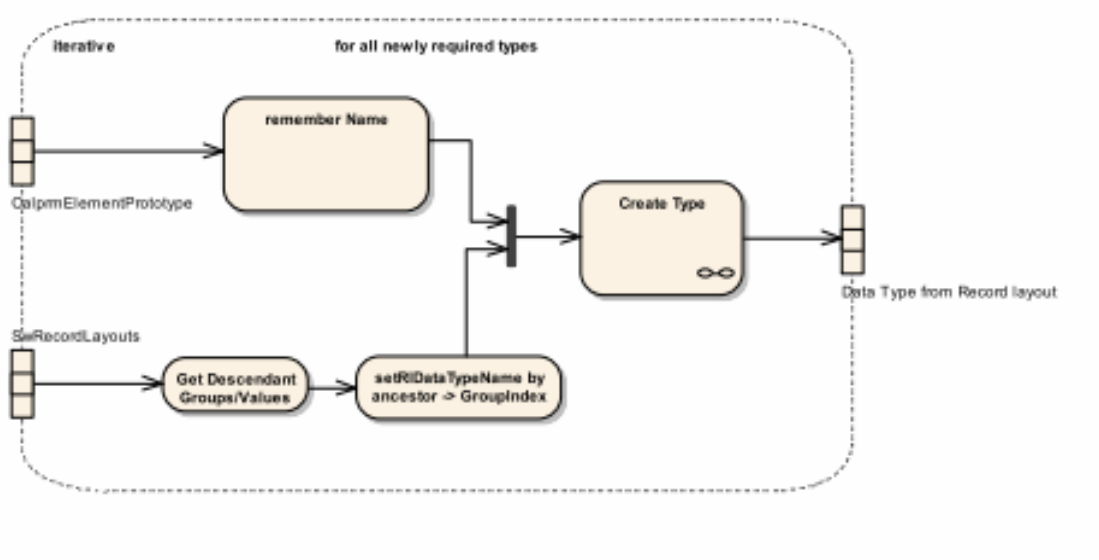

Figure 55: algorithm to map record layouts to types

For each data type, several subtypes must be created. The details of the algorithm are spefied in the following diagram.

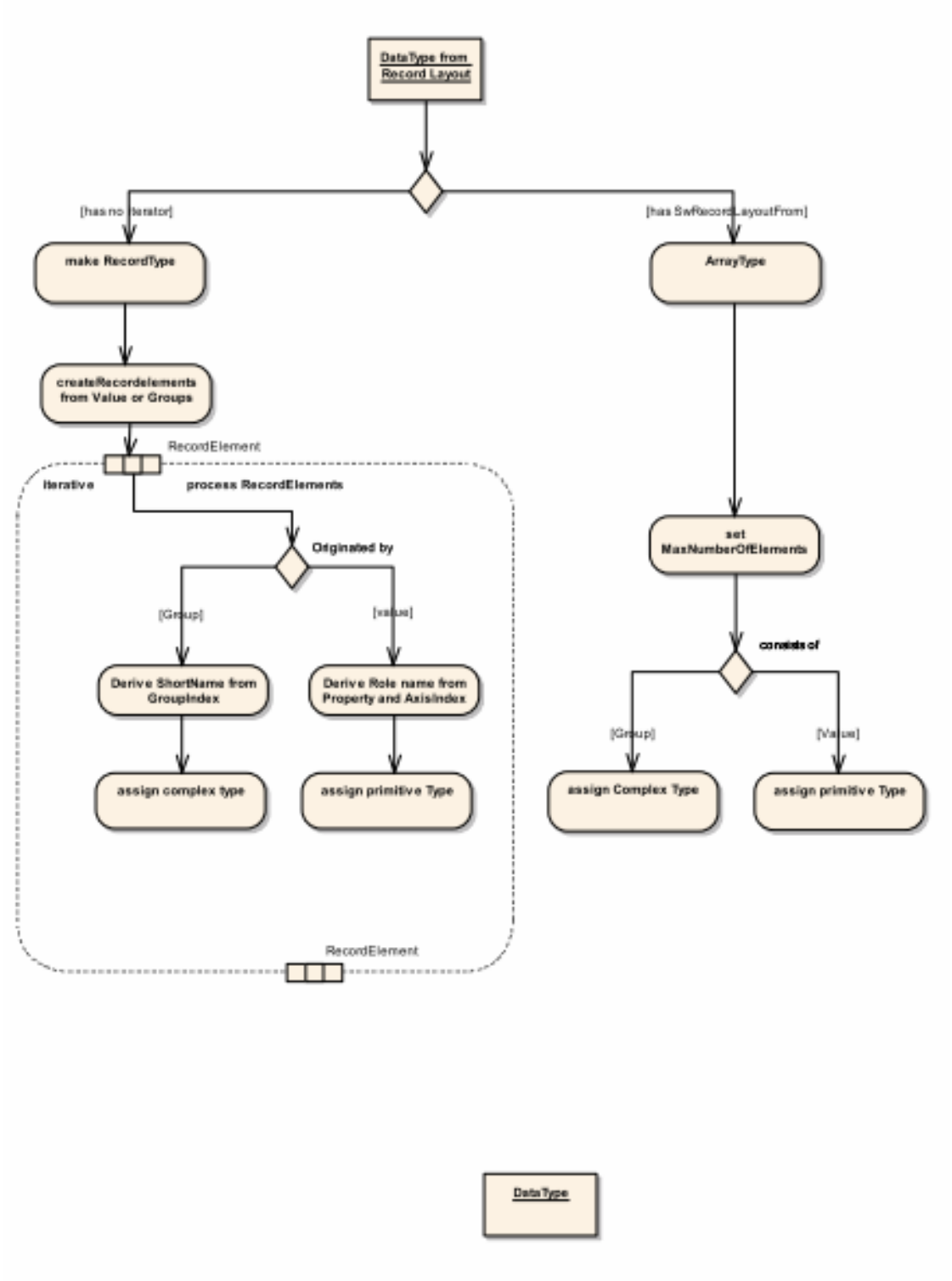

Figure 56: Creating types from record layouts

7 Interaction with the Run Time Environment

7.1 Scope of this Chapter

This chapter describes aspects of atomic software-components[20] that have not actually the character of an interface but are nevertheless essential for responding to the following challenges:

1. support the proper configuration of the RTE and the BSW: the software-component description needs to provide detailed information on how the underlying software should behave with respect to the software-component (for example: what runnables of the software-component should be started when by the RTE)[21],
2. describe the communication properties of a software-component,
3. serve as a basis for the description of the detailed resource requirements of software-components, and
4. provide a more detailed description of the timing behavior of atomic software-components.

[20] An atomic software-component is atomic in the sense that it cannot be distributed over several ECUs any more.
[21] Several section of this chapter have originally been written from an operating system (OS) perspective. Please note, however, that from the view of a software component the activation of runnable entities is a feature provided by the RTE not the OS. Which features of the underlying OS are used depends on the RTE implementation.

7.2 Overview of this Chapter

7.2.1 Meta-Model Overview

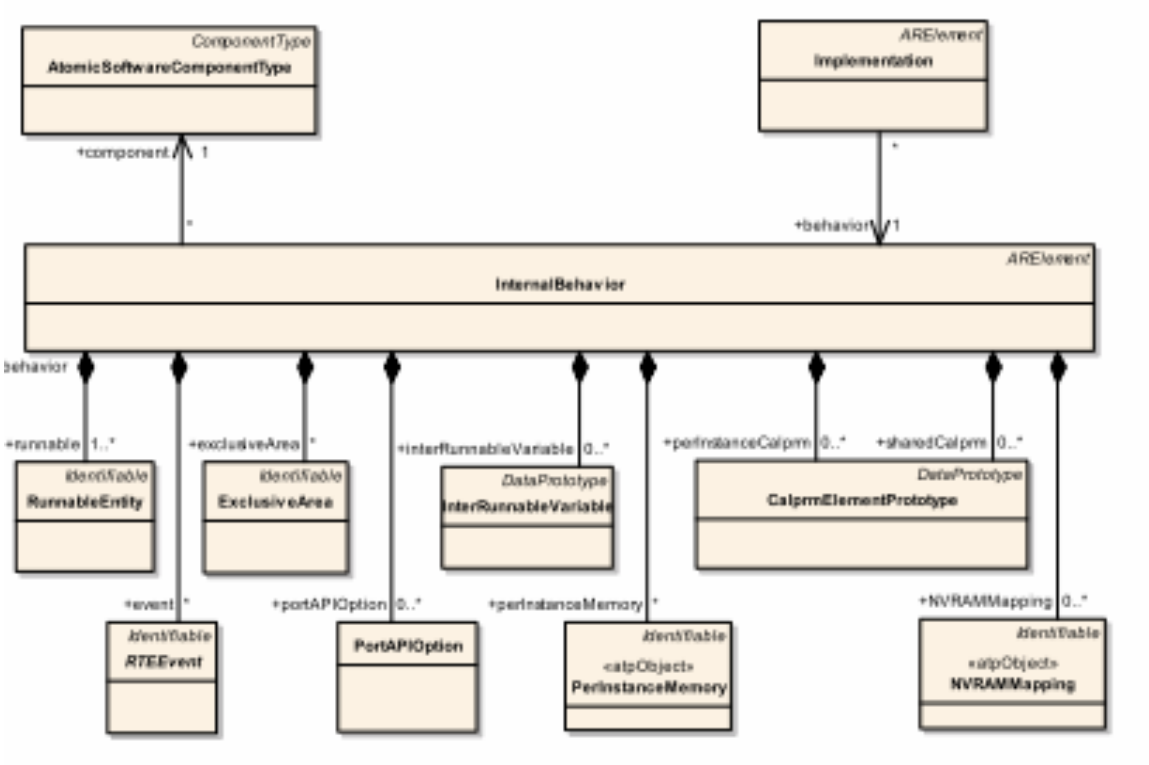

Figure 57: Overview of the meta-model related to Internal Behavior

Chapter 2 outlines the external view of a component (i.e. its ports). In the meta-model, the concept of an atomic software-component is represented by the meta-class "AtomicSoftwareComponentType".
This chapter focuses on the meta-class "InternalBehavior" which contains a description of the internal behavior of the atomic software-component.
Please note that chapter 8 deals with the meta-class "Implementation" which describes a specific implementation of a specific internal behavior of an atomic software-component. This description includes, for example, the specification of the resource usage.

Class	InternalBehavior			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior			
Class Description	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)	Identifiable, PackageableElement, NonSplitableElement, ARElement			
Attribute	Datatype	Mul.	Link Type	Attribute Description
component	AtomicSoftwareComponentType	1	reference	The component this behavior is defined for.
event	RTEEvent	0..*	aggregation	
exclusiveArea	ExclusiveArea	0..*	aggregation	

interRunnableVariable	InterRunnableVariable	0..*	aggregation	
NVRAMMapping	NVRAMMapping	0..*	aggregation	Mapping of NVRAM related resources to the respective port, see class description.
perInstanceCalprm	CalprmElementPrototype	0..*	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
perInstanceMemory	PerInstanceMemory	0..*	aggregation	The SW-C needs per-instance memory
portAPIOption	PortAPIOption	0..*	aggregation	Options for generating the signature of port-related calls from a runnable to the RTE and vice versa.
runnable	RunnableEntity	1..*	aggregation	
sharedCalprm	CalprmElementPrototype	0..*	aggregation	
supportsMultipleInstantiation	Boolean	1	aggregation	Flag, whether the component can be multiply instantiated. which will result in an appropriate component API on programming language level (with or without instance handle).

7.2.2 The Runnable Entities

The concept of *Runnable Entities* (also abbreviated simply as Runnable) is defined in the VFB spec. Runnable entities (represented by the meta-class "RunnableEntity") 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.

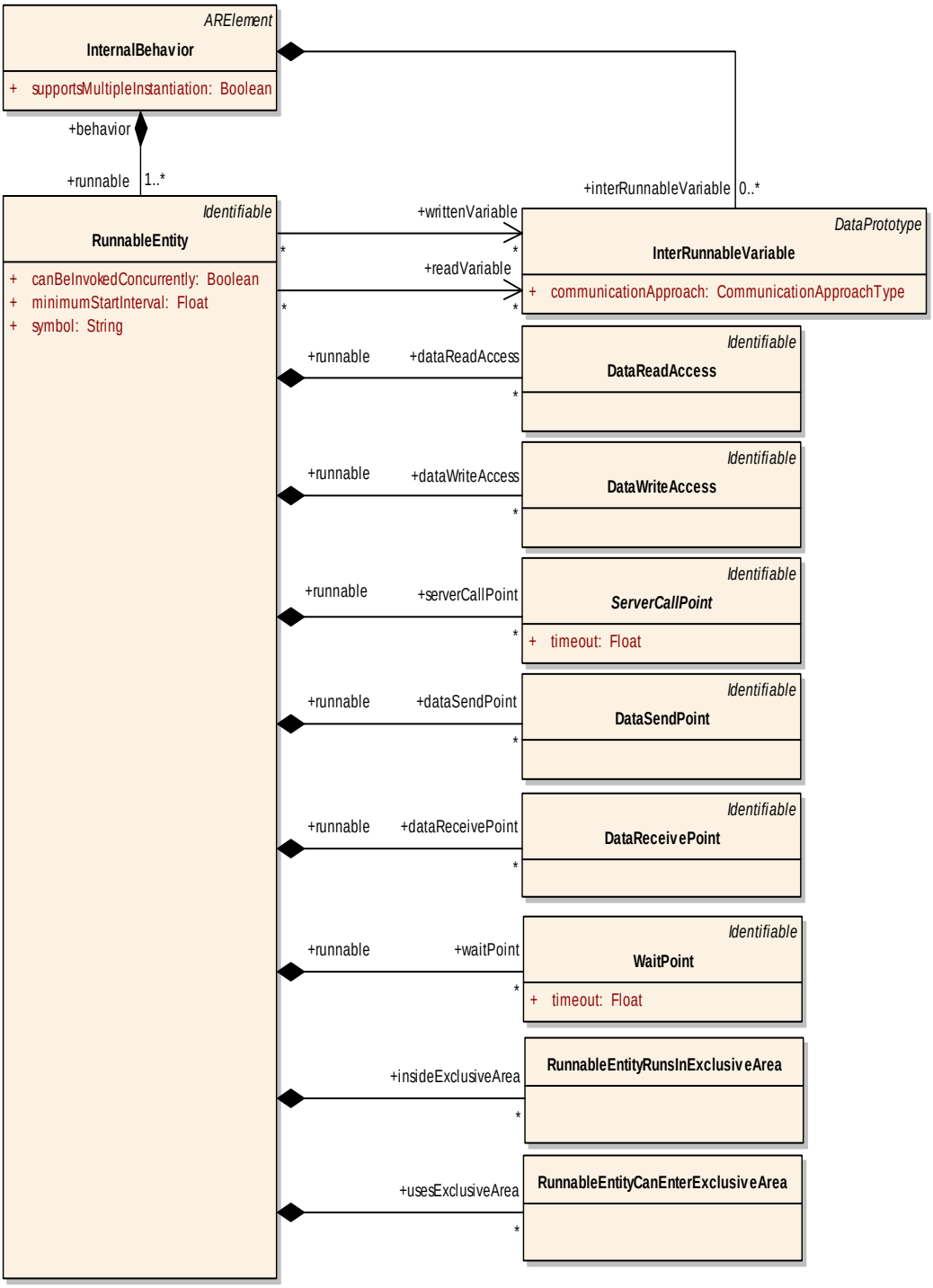

Figure 58: Overview of the meta-model for RunnableEntity

An implementation of an atomic software-component has to provide an entry-point to code for each Runnable in its "InternalBehavior" (please find a more detailed description of this subject in chapter 8).
Please note that it is intentionally not possible for "CompositionType" to be referenced by "InternalBehavior". Consequently, Compositions of software-components don't have Runnables by themselves. Only the atomic software-components that a

composition consists of may have Runnables. This correlation is depicted in Figure 59.

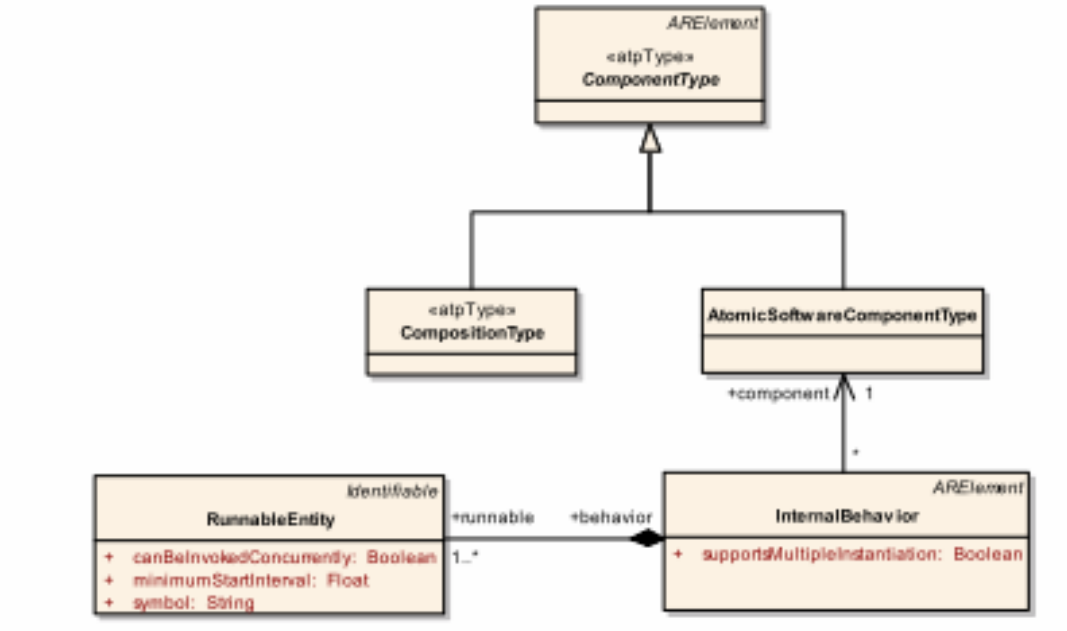

Figure 59: Only atomic software-components may have runnables

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. Note further that this document emphasizes on "RunnableEntities" of category 1A, 1B, and 2.

Class	RunnableEntity			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior			
Class Description	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)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
calprmAccess	CalprmAccess	0..*	aggregation	
canBeInvokedConcurrently	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.

dataReadAccess	DataReadAccess	0..*	aggregation	Runnable has read access to data element
dataReceivePoint	DataReceivePoint	0..*	aggregation	Data receive points of this runnable.
dataSendPoint	DataSendPoint	0..*	aggregation	The runnable has data send point.
dataWriteAccess	DataWriteAccess	0..*	aggregation	Runnable has write access to data element
insideExclusiveArea	RunnableEntityRunsInExclusive-Area	0..*	aggregation	The runnable entity runs inside the referenced exclusive area
minimumStartInterval	Float	1	aggregation	Specifies the time in seconds which two starts of a RunnableEntity are guaranteed to be separated.
modeSwitchPoint	ModeSwitchPoint	0..*	aggregation	The runnable has a mode switch point.
perInstanceCalprmAccess	RunnableEntity_perInstance-CalprmAccess	0..*	reference to instance	
readVariable	InterRunnableVariable	0..*	reference	Inter-runnable variables that are read by this Runnable.
serverCallPoint	ServerCallPoint	0..*	aggregation	The runnable has server call point.
sharedCalprmAccess	CalprmElementPrototype	0..*	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.
usesExclusiveArea	RunnableEntityCanEnterExclusive-Area	0..*	aggregation	This means that the runnable can enter/leave the referenced exclusive area through explicit API calls.
waitPoint	WaitPoint	0..*	aggregation	The runnable has wait point.
writtenVariable	InterRunnableVariable	0..*	reference	Inter-runnable variables that are written by Runnable.

The attribute “canBeInvokedConcurrently” defines whether the runnable is a single-threaded entity or whether a runnable of an instance of the component can be invoked concurrently (in multiple tasks of the OS). The following two paragraphs define both cases.
The attribute “minimumStartInterval” defines the time which the RTE will guarantee between two starts of this RunnableEntity.

7.2.2.1 Concurrency and Reentrancy of a Runnable that cannot be Invoked Concurrently

This section applies to the case that the attribute “canBeInvokedConcurrently” is FALSE.
During run-time, each Runnable of each instance of an atomic software-component is (by being a member of an OS task) in one of three states:

- "Suspended": the initial state, when the Runnable is passive and can be started
- "Enabled": the Runnable should run (because for example a message has been received on a port of a component or a timing event occurs)
- "Running": the Runnable is running within a running task. From this state, the Runnable can either perform a transition to "Enabled" (if it has been preempted because the task has been preempted) or to "Suspended".

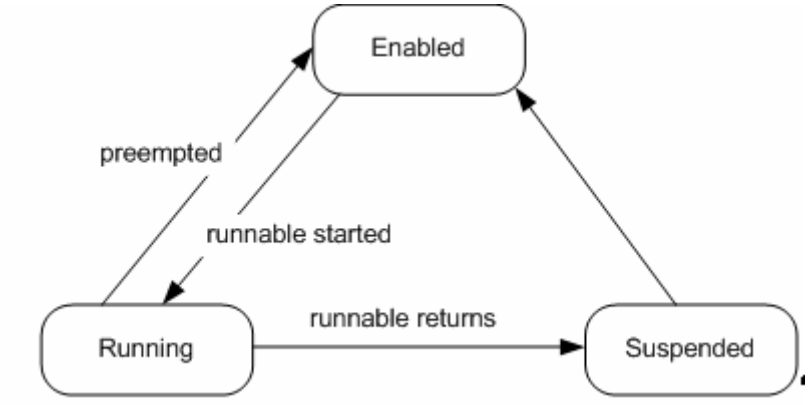

Figure 60: Task-derived run-time states of a runnable

The "InternalBehavior" describes for each Runnable, when a transition from "Suspended" to "Enabled" should occur. This is done using the concept of an "RTEEvent".
When a Runnable is in state "Enabled", the OS can decide to start running the Runnable. The delay between entering the state "Enabled" (e.g. a message has been received in response to which the runnable should run) and moving into the state "Running" (the first instruction of the runnable has been executed) depends on the scheduling strategy of the OS.
The transition from the state "Running" into the state "Suspended" is in the hands of the Runnable: the transition occurs when the Runnable returns (thereby handing over

control to the OS). Some Runnables (like cat. 2 Runnables) might never return from the "Running" state once they entered it. Cat. 1A and 1B Runnables will typically return after having executed a specific finite algorithm (the execution time of which might be provided).
In most cases Runnables will not be scheduled individually but as parts of OS tasks. Please note that the concept of run-time states as depicted in Figure 60 has been created along the example of the OSEK Operating System specification.
In case the internal behavior defines a runnable as one that cannot be invoked concurrently, it is the responsibility of the RTE and the BSW to make sure that the runnable is never started concurrently (in for example two OS tasks). This implies that the implementation of the SW-Component does not need to worry about concurrency issues.
For example: The internal behavior of a component-type MyComponentType describes a Runnable R1, which should be enabled when an operation on a client-server p-port of the component is invoked. The component specifies that the Runnable R1 cannot be invoked concurrently. The component MyComponentType is instantiated on an ECU. When a call of the operation is received, the corresponding instance of the Runnable R1 is enabled and the OS will start executing the Runnable (the Runnable is in state "running") in a task managed by the OS. If *another* call of the operation is received while the Runnable is in state "running", it is *not* allowed that the OS runs the Runnable again in a second task. Rather, the OS has to wait (and maybe queue the second incoming request) until the Runnable has returned and has moved to the "Suspended" state.

7.2.2.2 Concurrency and Reentrancy of a Runnables 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 runnable is running several times concurrently in different OS tasks. This implies that the state machine defined in Figure 62 is not the state of the runnable any more, but can be cloned an arbitrary number of times.
Note that the SW-Component description itself does not put any bounds on the number of concurrent invocations of the runnable that are allowed. The SW-Component description only specifies whether the runnable can be invoked concurrently or not.
Allowing concurrent invocation of a runnable implies that the implementation of the SW-component needs to take care of this additional form of concurrency.
For example: The internal behavior of a component-type MyComponentType describes a Runnable R1, which should be enabled when an operation on a client-server p-port of the component is invoked. The component specifies that the Runnable R1 can be invoked concurrently. The component MyComponentType is instantiated on an ECU. When a call of the operation is received, the corresponding instance of the Runnable R1 is enabled and the OS will start executing the Runnable (the Runnable is in state "running") in a task managed by the OS. If *another* call of the operation is received, it is allowed that the same runnable is started again in a different task.
A typical use-case of concurrent runnables 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 runnable implementing the AUTOSAR service runs directly in the context (in the task) of the software-component 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.

7.2.2.3 Additional Remarks and Clarifications

7.2.2.3.1 Reentrancy and Multiple Instantiation: supportMultipleInstantiation and canBeInvokedConcurrently

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

supportsMultipleInstantiation	canBeInvokedConcurrently	Implication for an implementation of a runnable
FALSE	FALSE	This implies that the implementation of the runnable will never be invoked concurrently from several tasks. The implementation does not need care about reentrancy issues[22] and can typically use “static variables” to store state.
TRUE	FALSE	In case there are several instances of the same component on the local ECU, the implementation of the runnable can still be invoked concurrently from several tasks. However, there will be not 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 runnable can be invoked concurrently from several tasks, even with the same instance handle.

7.2.2.3.2 Reentrancy and “Library Functions”

Note that all code that is called by different Runnables (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.

7.3 RTE Events

During execution, several run-time events will occur, such as the reception of a remote operation-invocation on a P-Port or a timeout on an R-Port that is not receiving the data-elements it expects. Describing an RTEEvent in the software-component template includes two aspects:

[22] Note that this statement is only correct in case each runnable is implemented by its own “C”-function. In case the SW-Component implementation decides to map several runnables to the same “symbol”, the implementation needs to take care of the reentrancy issues that this causes. This is however an unlikely scenario.

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

As described in the virtual functional bus specification, the runnable entities of a software-component can interact with the occurrence of such events in two ways:

- the RTE can be instructed to enable a specific runnable when the event occurs
- the RTE can provide "wait-points", that allow a runnable to block until an event in a set of events occurs

7.3.1 Defining an Event

The description of the internal behavior includes a description of all events that the internal behavior of the atomic software-component relies on. This "RTEEvent" shows up as an "abstract" base-class (see Figure 61) in the meta-model: the exact attributes of the "RTEEvent" depend on the exact event that is described.

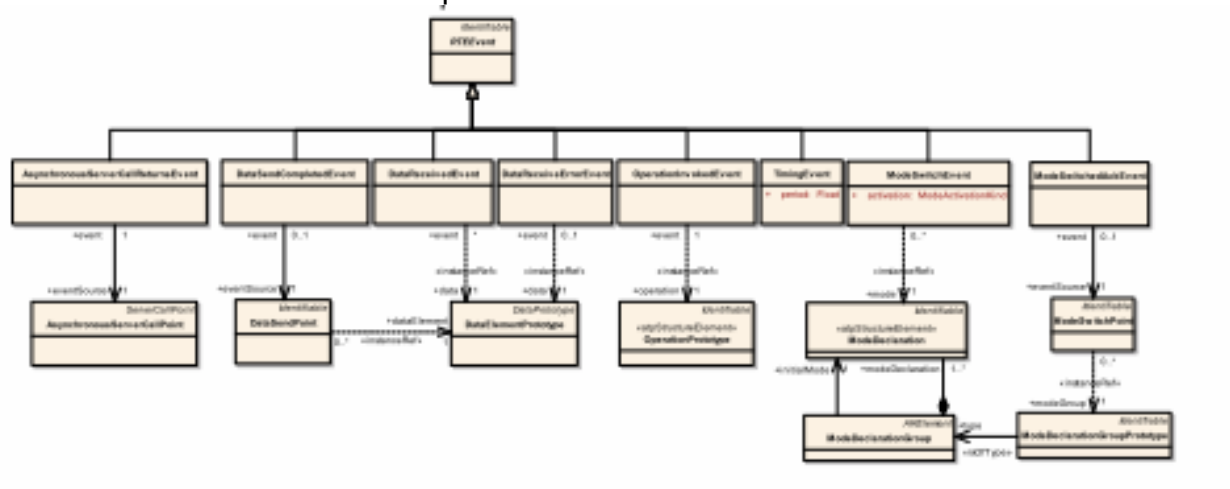

Figure 61: Kinds of RTE events

The details of the various kinds of concrete events (such as the "TimingEvent", "DataSendCompletedEvent", etc.), is described in chapters 7.5, 7.6, and 7.7

.

Class	RTEEvent {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents			
Class Description	Abstract base class for all RTE-related events			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
modeDependency	ModeDisablingDependency	0..1	aggregation	Provides the means to describe the Modes this RTEEvent can be disabled by.
startOnEvent	RunnableEntity	0..1	reference	Runnable starts when event occurs

.

Class	ModeDisablingDependency
Package	M2::AUTOSARTemplates::SWComponentTemplate::ModeDeclaration
Class Description	Collection of references to the Modes that disable the RTEEvent

Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dependentOnMode	ModeDisablingDependency_dependent-OnMode	1..*	reference to instance	Reference to the Modes that disable the Runnable Entity.

7.3.2 Defining how to Respond to an Event

In case the OS needs to start a Runnable when the corresponding event occurs, the "RTEEvent" can directly reference the Runnable that needs to be started. When the software-component description uses this feature, it is the responsibility of the AUTOSAR OS to start the Runnable when the event occurs.
In case the Runnable wants to block and wait for events (which makes the runnable into a cat. 2 runnable), the description of the runnable may include the definition of a "wait-point". Such a "WaitPoint" (see Figure 62) contains a reference to all events that are waited for. The wait-point will block until one of the referenced events occurs.

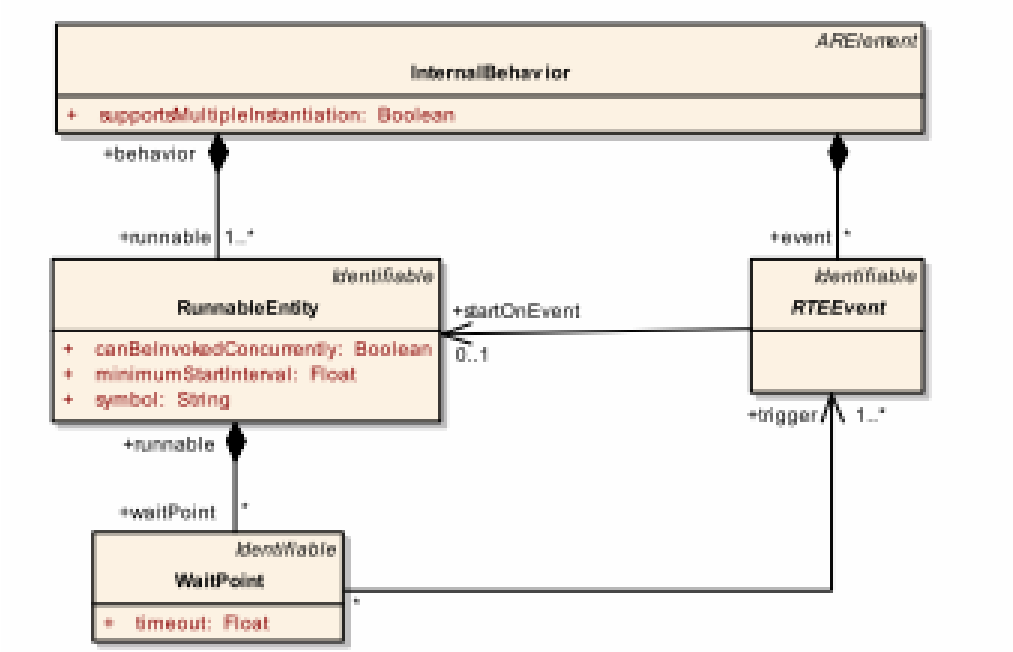

Figure 62: Description of the interaction between an RTEEvent and Runnables

A single "RunnableEntity" can **actually wait** only at a single "WaitPoint" provided that the runnable can only be scheduled a single time[23]. On the other hand, it is in general possible that a single event can be used to trigger "WaitPoints" in different "RunnableEntities".

Class	WaitPoint
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents

[23] 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.

Class Description	This defines a wait-point for which the runnable can wait.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute 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.

7.4 Overview of 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.
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 “communication specification” (short: 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 ComSpecs is restricted to the ports of “AtomicSoftwareComponentType”. It is forbidden to attach ComSpecs to a “PortPrototype” belonging to a “CompositionType”. Allowing this would mean that ComSpecs in delegated ports could be repeatedly overwritten in nested composition hierarchies. Such a behavior must be avoided as several attributes contained in the PortComSpec definitions are relevant for the communication API of the “AtomicSoftwareComponentType”; in order to maintain the reuse aspect of the component these attributes must not change when the “ComponentType” is integrated into different composition hierarchies.
Sections 7.5 and 7.6 then explain the sender-receiver and client-server communication patterns with respect to the RTE, the RTE events and the corresponding communication attributes.

7.4.1 Communication Specification of an R-Port

Figure 63 shows the model of the communication attributes relevant for an R-Port.

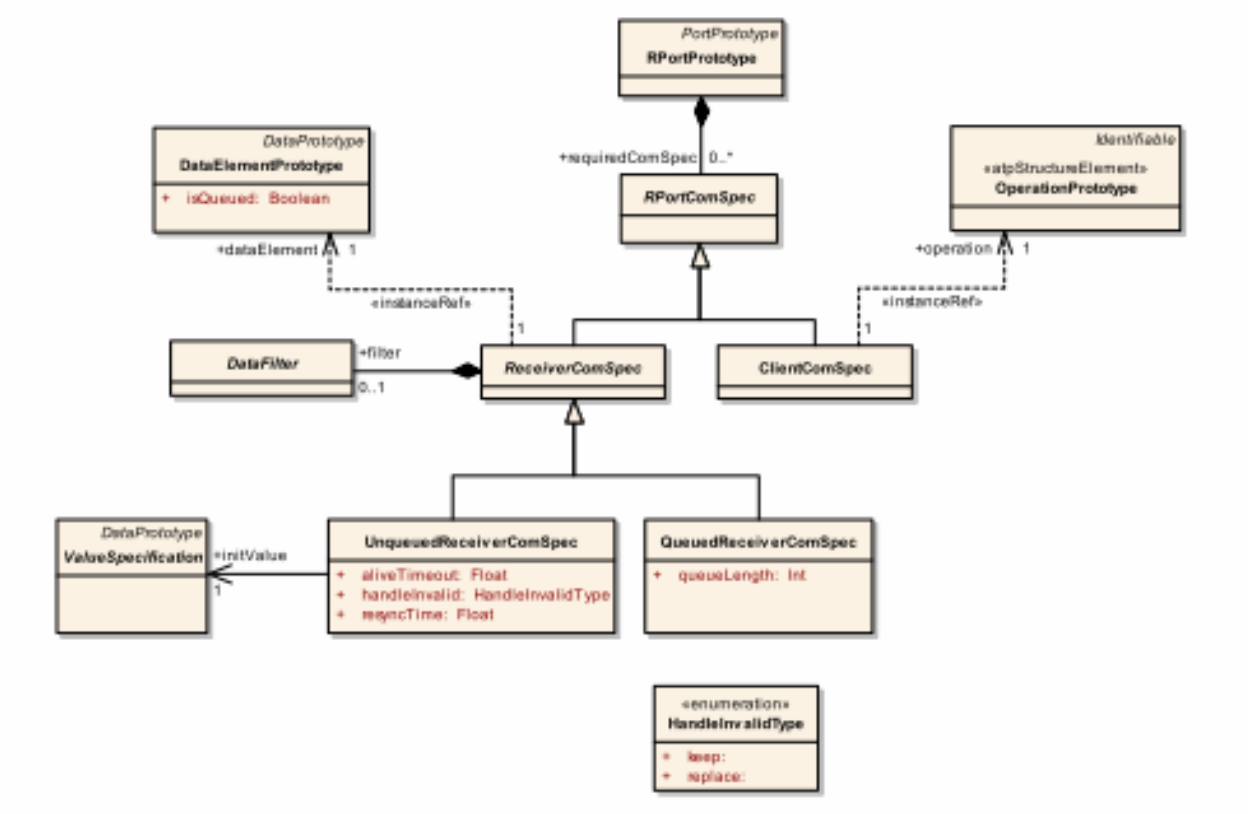

Figure 63: Communication attributes of "RPortPrototype".

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 depend on whether the data-elements represent queued or non-queued data [24]. This is expressed in the inheritance tree of ComSpec classes. Each of these classes may then carry the specific attributes.

An R-Port may aggregate many ComSpec classes, possibly one for each interface element (data element or operation) the associated interface contains. The meaning of the attributes shown above is explained in the following class tables. Classes that have no attributes are not listed here.

Class	ReceiverComSpec {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Receiver specific communication attributes (R-Port and sender-receiver interface).			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, RPortComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dataElement	ReceiverComSpec_dataElement	1	reference to instance	Data element these attributes belong to.
filter	DataFilter	0..1	aggregation	

[24] It is a frequent point of discussion whether those communication attributes are specified per data-element/operation or per port. At this point WP 2.1.1.1 follows the specification of the VFB, which identifies the *granularity* of those attributes to be on data-element/operation level.

Class	UnqueuedReceiverComSpec			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Communication attributes specific to reciving data (opposed to receiving events).			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, RPortComSpec, ReceiverComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute 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.
handleInvalid	HandleInvalidType	1	aggregation	Specifies strategy of handling the reception of invalidValue.
initValue	ValueSpecification	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.
resyncTime	Float	1	aggregation	Time allowed for resynchronization of data values after current data is lost, e.g. after an ECU reset.

Class	QueuedReceiverComSpec			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Communication attributes specific to reciving events.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, RPortComSpec, ReceiverComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
queueLength	Integer	1	aggregation	Length of queue for received events.

Class	ClientComSpec			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Client specific communication attributes (R-Port and client-server interface).			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, RPortComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
operation	ClientComSpec_operation	1	reference to instance	Operation these attributes belong to.

7.4.2 Communication Specification of Data Filters

Figure 64 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.

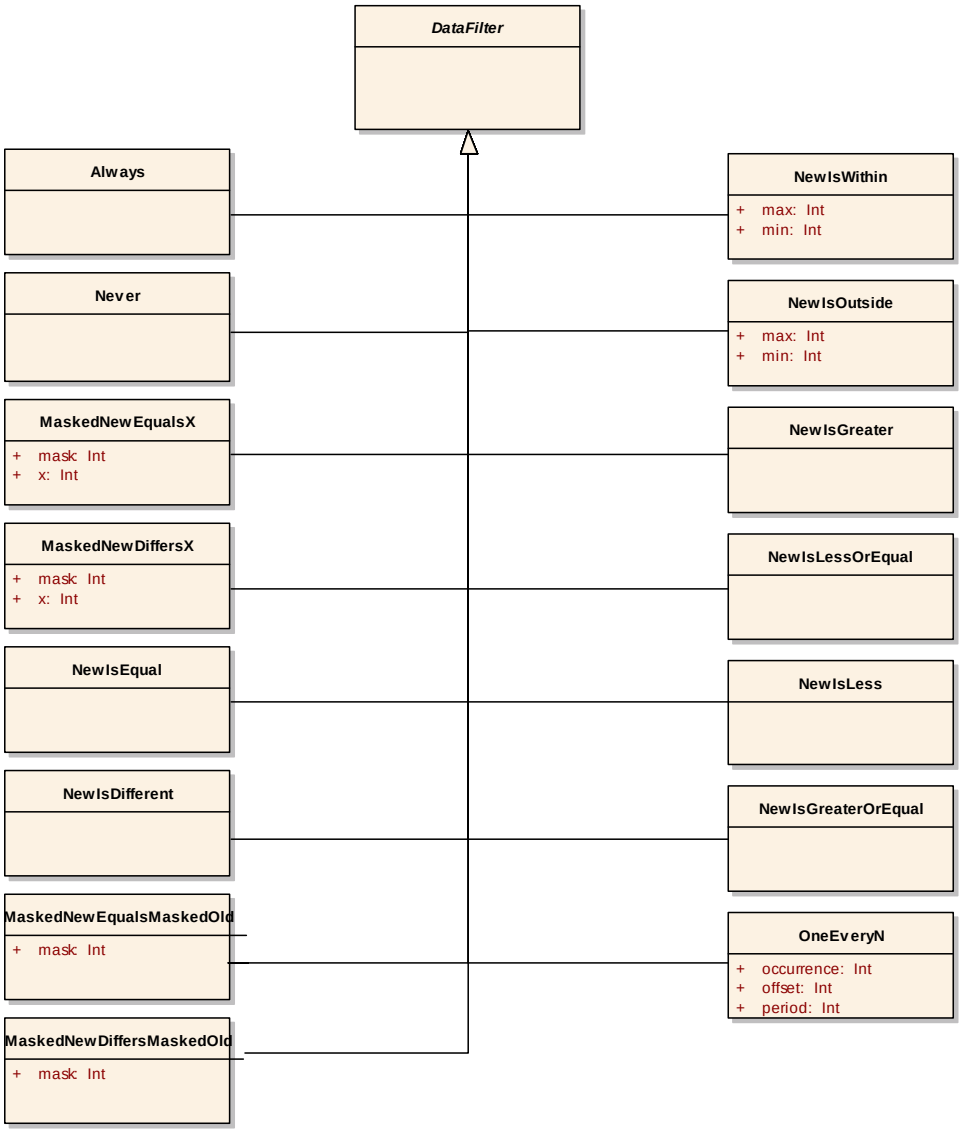

Figure 64: DataFilter and its communication attributes.

Class	DataFilter {abstract}
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	Base class for data filters.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject

Class	Always
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	No filtering is performed so that the message always passes.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Class	Never
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	The filter removes all messages.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Class	MaskedNewEqualsX			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter			
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter			
Attribute	Datatype	Mul.	Link Type	Attribute Description
mask	Integer	1	aggregation	mask for the new Value
x	Integer	1	aggregation	Value to compare with

Class	MaskedNewDiffersX			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter			
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter			
Attribute	Datatype	Mul.	Link Type	Attribute Description
mask	Integer	1	aggregation	mask for the new Value
x	Integer	1	aggregation	Value to compare with

Class	NewIsEqual
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Class	NewIsDifferent
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Class	MaskedNewEqualsMaskedOld			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter			
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter			
Attribute	Datatype	Mul.	Link Type	Attribute Description
mask	Integer	1	aggregation	mask for old and new value

Class	MaskedNewDiffersMaskedOld			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter			
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter			
Attribute	Datatype	Mul.	Link Type	Attribute Description
mask	Integer	1	aggregation	mask for old and new value

Class	NewIsWithin
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	Pass a message if its value is within a predefined boundary. min <= new_value <= max
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Attribute	Datatype	Mul.	Link Type	Attribute Description
max	Integer	1	aggregation	Value to specify the lower bounddary
min	Integer	1	aggregation	Value to specify the lower bounddary

Class	NewIsOutside			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter			
Class Description	Pass a message if its value is outside a predefined boundary. (min > new_value) OR (new_value > max)			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter			
Attribute	Datatype	Mul.	Link Type	Attribute Description
max	Integer	1	aggregation	Value to specify the lower bounddary
min	Integer	1	aggregation	Value to specify the lower bounddary

Class	NewIsGreater
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Class	NewIsLessOrEqual
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Class	NewIsLess
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Class	NewIsGreaterOrEqual
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter

Class	OneEveryN			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication::Filter			
Class Description	Pass a message once every N message occurrences. 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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, DataFilter			
Attribute	Datatype	Mul.	Link Type	Attribute Description
occurrence	Integer	1	aggregation	
offset	Integer	1	aggregation	
period	Integer	1	aggregation	

7.4.3 Communication Specification of a P-Port

In analogy to the previous section, Figure 65 shows the attribute classes relevant for a P-Port.

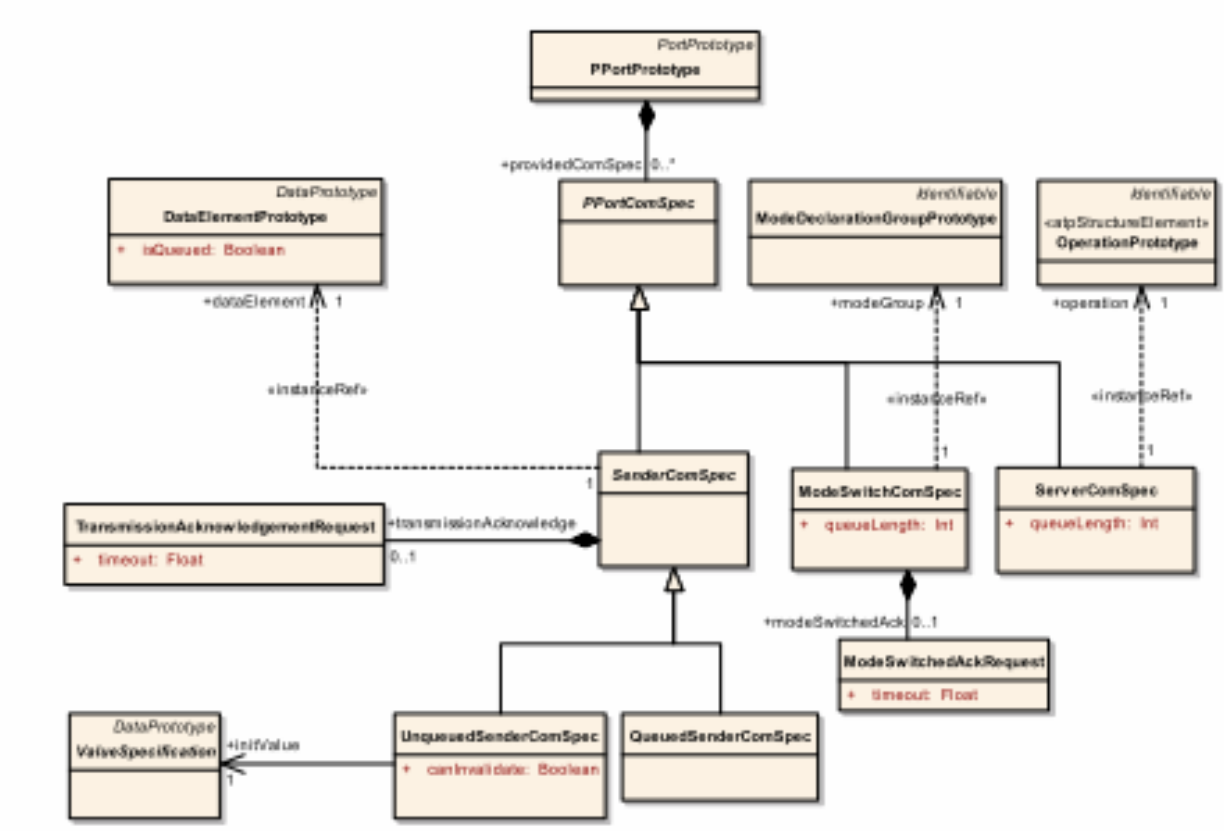

Figure 65: P-port and its communication attributes.

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	SenderComSpec {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Communication attributes for a sender port (P-Port and sender-receiver interface).			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PPortComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dataElement	SenderComSpec_dataElement	1	reference to instance	Data element these quality of service attributes apply to.
transmissionAcknowledge	TransmissionAcknowledgement-Request	0..1	aggregation	Requested transmission acknowledgement for data element.

Class	TransmissionAcknowledgementRequest
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication
Class Description	Requests transmission acknowledgement that data has been sent successfully. Success/failure is reported via a SendPoint of a Runnable.
Base	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject

Class(es)				
Attribute	Datatype	Mul.	Link Type	Attribute Description
timeout	Float	1	aggregation	Number of seconds before an error is reported or in case of allowed redundancy, the value is sent again.

Class	UnqueuedSenderComSpec			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PPortComSpec, SenderComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
canInvalidate	Boolean	1	aggregation	Flag whether the component can actively invalidate data.
initValue	ValueSpecification	1	reference	Init value to be sent if sender component is not yet fully initialized, but receiver needs data already.

Class	ServerComSpec			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Communication attributes for a server port (P-Port and client-server interface).			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, PPortComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
operation	ServerComSpec_operation	1	reference to instance	Operation these communication attributes apply to.
queueLength	Integer	1	aggregation	Length of call queue on the server side. The queue is implemented by the RTE.

Class	ModeSwitchComSpec			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Communication attributes for both sender /server port (P-Port and sender-receiver interface). (0=unqueued)			
Base Class(es)	ARObject, PPortComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
modeGroup	ModeSwitchComSpec_ModeDeclaration-GroupPrototype	1	reference to instance	--
modeSwitchedAck	ModeSwitchedAckRequest	0..1	aggregation	--
queueLength	Integer	1	aggregation	Length of call queue on the server side. The queue is implemented by the RTE.

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

7.4.4 Communication Specification of Connectors

Connectors also have some attributes associated with them as shown in Figure 66.

Figure 66: Quality of service attributes of a connector.

Please note that the current WP 2.1.1.1 model deviates from the VFB spec v1.02: WP 1.1.1 suggested to also defining an “exact transfer time” for a connector. However, a change request has been submitted and was accepted in Bugzilla[25] to remove it from the VFB specification.
The ComSpec classes that carry actual attributes are once again listed in the following tables.

Class	ConnectorComSpec {abstract}
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication
Class Description	Communication attributes for all connector classes.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject

[25] http://www.autosar.org/bugzilla/show_bug.cgi?id=669

Class	SenderReceiverConnectorComSpec			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Communication attributes for connectors between sender and receiver ports.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, ConnectorComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dataElement	SenderReceiverConnectorCom-Spec_dataElement	1	reference to instance	Data element these attributes apply to.
maxResponseTime	Float	1	aggregation	Maximum allowed time (seconds) for date transfer via this connector. The time has to be measured from the moment, at which the information to be sent has been made available by the sending component at its P-Port until it is ready to be used by the receiving component at its R-Port.
minResponseTime	Float	1	aggregation	Minimum allowed time (seconds) for date transfer via this connector. The time has to be measured from the moment, at which the information to be sent has been made available by the sending component at its P-Port until it is ready to be used by the receiving component at its R-Port. Note that in many cases the minimum required time will be zero.

Class	ClientServerConnectorComSpec			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Communication			
Class Description	Communication attributes for connectors between client and server ports.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, ConnectorComSpec			
Attribute	Datatype	Mul.	Link Type	Attribute Description
operation	ClientServerConnectorComSpec_-operation	1	reference to instance	Operation these attributes apply to.

7.5 Runnables and Sender-Receiver Communication

This section describes the sender-receiver communication relevant attributes of a 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 Runnable,
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.

7.5.1 Terminology: Explicit vs. Implicit Communication

The AUTOSAR meta-model foresees two different approaches for sender-receiver communication. These are described in detail in chapters 7.5.2 and 7.5.3. 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 7.5.2) and "explicit" for communication based on Data-Points (please refer to chapter 7.5.3).
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.

7.5.2 Data-Access

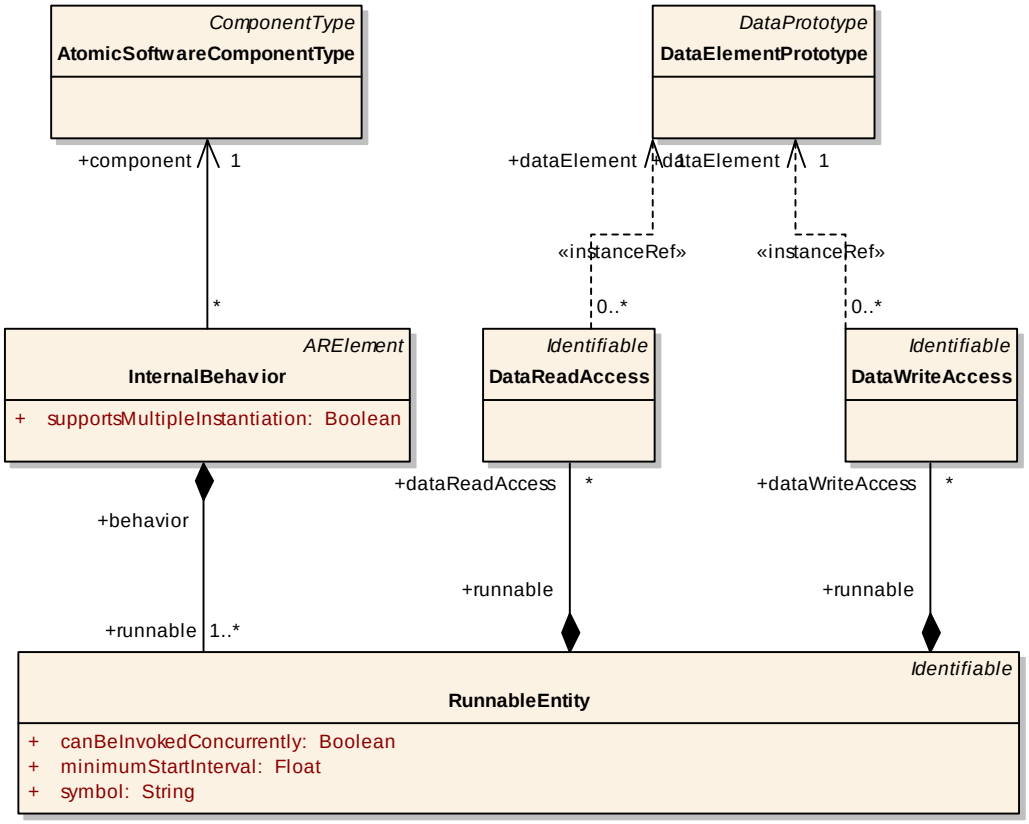

Figure 67: DataReadAccess and DataWriteAccess

The "InternalBehavior" can specify that a Runnable needs read-access (respectively write-access) to the data-elements of an RPort (respectively PPorts). The usage of this access mechanism to the data-elements is appropriate for cat. 1 Runnables only, which guarantees finite response time (opposed to waiting for data for instance).

Class	DataReadAccess			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::DataElements			
Class Description	The presences of a DataReadAccess means that the runnable needs access to the DataElement in the rPort. The runnable will not modify the contents of the data but only read the information. The runnable expects that the contents of this data does NOT change during its entire execution.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dataElement	DataReadAccess_dataElement	1	reference to instance	The data element that is going to be read by this runnable.

Class	DataWriteAccess
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::DataElements

Class Description	The presences of a DataWriteAccess means that the runnable will potentially modify the dataElement in the pPort. The runnable has free access to the data-element while it is running. The runnable has the responsibility to make sure that the data-element is in a consistent state when it returns. When using DataWriteAccess the new values of the data-element is not made available via the communication infrastrucure before the runnable returns (exits the "Running" state).			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dataElemement	DataWriteAccess_dataElement	1	reference to instance	The data element that is going to be written to by this runnable.

7.5.3 Explicit Sending and Receiving

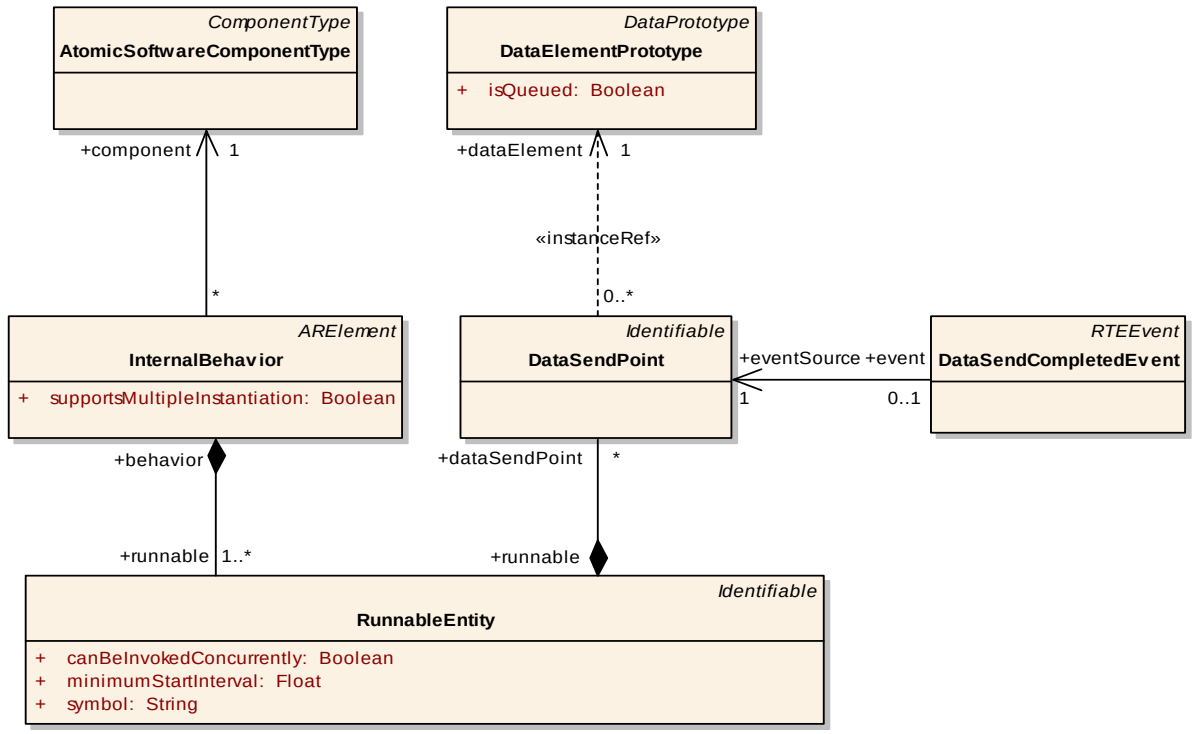

Figure 68: DataSendPoint

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 Runnable needs to explicitly "send" through an API; when using a “DataWriteAccess”, the Runnable 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 data-element is not made available before the Runnable returns (exits the "Running" state).
- The presence of a "DataSendPoint" per definition lets the corresponding "RunnableEntity" attain cat. 1B.

Class	DataSendPoint			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::DataElements			
Class Description	A data send point specifies that a runnable explicitly sends a certain data element.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
dataElement	DataSendPoint_dataElement	1	reference to instance	The data element that is sent by this runnable.

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 69.

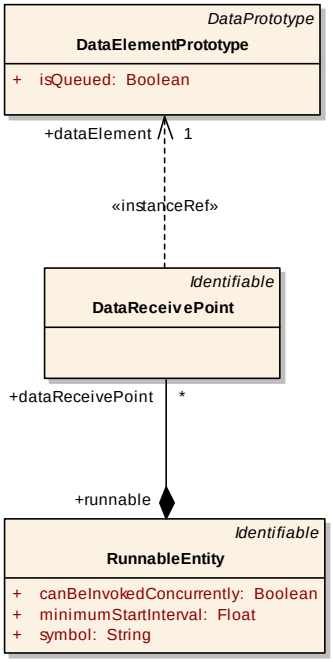

Figure 69: Definition of an explicit request to receive data.

By using a “DataReceivePoint” instead of “DataReadAccess” the constraining access to the referenced data element – other runnables must not change the data element during the read execution – is limited to a short, well-defined amount of time. Therefore, category 1 "RunnableEntities" may also have “DataReceicePoints” and consequently become "RunnableEntities" of cat. 1B.

Class	DataReceivePoint			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::DataElements			
Class Description	A data receive point allows a runnable 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	Attribute Description
dataElement	DataReceivePoint_dataElement	1	reference to instance	The data element to be explicitily read.

Please note that it would in general be possible to combine a "DataReceivePoint" with a "WaitPoint" (see Figure 62) 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 cat. 2.

7.5.4 DataSendCompletedEvent

The "DataSendPoint" also allows for the definition of a "DataSendCompletedEvent", as shown in Figure 68. 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 software-component describes what success or failure of the send operation means in particular.
Firstly, via a ComSpec class different acknowledgement requests (e.g. successful transmission, successful reception) can be attached to a "PPortPrototype", as is shown in the left part of Figure 65.
This will configure the RTE in such a way that when data is sent, it will try to obtain the specified acknowledgement, possibly by waiting a certain timeout period or re-sending data a number of times.
In the actual component implementation, the event may carry attributes about the kind of acknowledgement obtained and whether it was successful. On the description level, however, these event attributes are not visible, unless the need for configuration exists – e.g. which Boolean value indicates success, and which one indicates error.
This is assumed to be not the case; therefore the “data send completed event” does not carry a *success* flag or anything like it.

Class	DataSendCompletedEvent			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents			
Class Description	The event is raised when the referenced data elements have been sent or an error occurs.			
Base Class(es)	Identifiable, RTEEvent			
Attribute	Datatype	Mul.	Link Type	Attribute Description
eventSource	DataSendPoint	1	reference	Data send point that triggers the event.

7.5.5 DataReceivedEvent

Similarly, a receiver is notified through the same event mechanism when a data element is received. As shown in Figure 70, the “data received event” is directly associated with the corresponding data element.

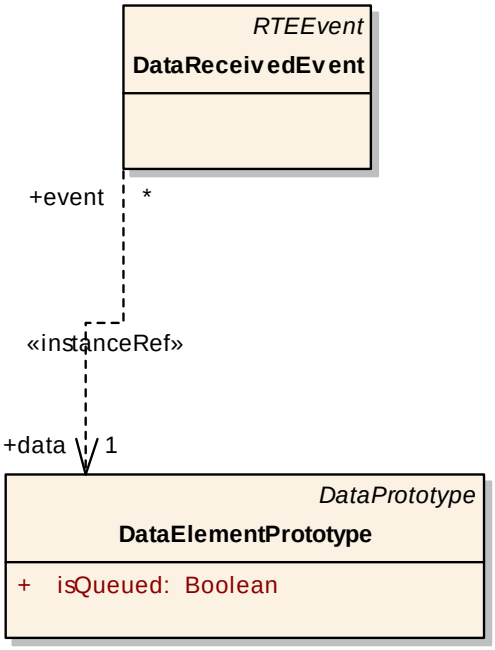

Figure 70: Receiver is notified by an event when new data has arrived.

The VFB specification stresses the point of the decision how to react to the event, e.g. through polling, by callback or some kind of OS supported suspend mechanism. Note that these concepts are already part of the overall event handling mechanisms described above.

Class	DataReceivedEvent			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents			
Class Description	The event is raised when the referenced data elements are received.			
Base Class(es)	Identifiable, RTEEvent			
Attribute	Datatype	Mul.	Link Type	Attribute Description
data	DataReceivedEvent_data	1	reference to instance	Data element referenced by event

According to the particular semantics of a "DataReceivedEvent", the latter is raised when the value of the associated "DataElementPrototype" is updated. However, this does not imply that a "RunnableEntity" automatically gets access to the "DataElementPrototype".
If the "RunnableEntity" requires access to the "DataElementPrototype" taken as the source for triggering the "RunnableEntity" it still needs to explicitly define the required kind of data access.

7.5.6 DataReceiveErrorEvent

A receiver is notified of "DataReceiveErrorEvent" through the activation of its Runnable which is referenced by this event. "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 layer. The following cases present some situations which will cause the RTE to raise a “data receive error event”:

- 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 which is 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.
- RTE receives a signal invalid notification from the COM layer when this latter detects that an incoming signal has the pre-defined 'invalid' value.

This event is used by the RTE to activate Runnables which handle the above-mentioned errors. The error code will be made available to the activated Runnable through the appropriate RTE API function.
This event cannot be associated with a wait point. It can only be used for the receiver component in a sender-receiver communication and in release 2.0 its data reference is restricted to DataElementPrototypes with their “isQueued” attribute set to false.
As shown in Figure 71, the “data receive error event” is directly associated with the corresponding data element and references the Runnable that is activated due to the occurrence of this event.

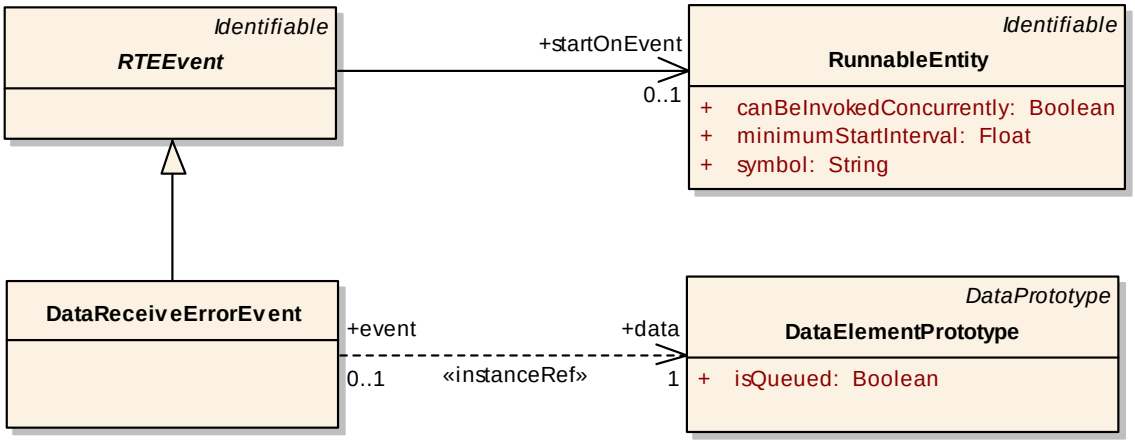

Figure 71: DataReceiveErrorEvent references a Runnable and a DataElementPrototype.

Class	DataReceiveErrorEvent			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents			
Class Description	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)	Identifiable, RTEEvent			
Attribute	Datatype	Mul.	Link Type	Attribute Description

data	DataReceiveErrorEvent_data	1	reference to instance	Data element referenced by event

7.6 Runnables and Client-Server Communication

7.6.1 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 72 for more details).
In case of a synchronous operation invocation the particular "RunnableEntity" merely needs a "SynchronousServerCallPoint" (see Figure 72). 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.

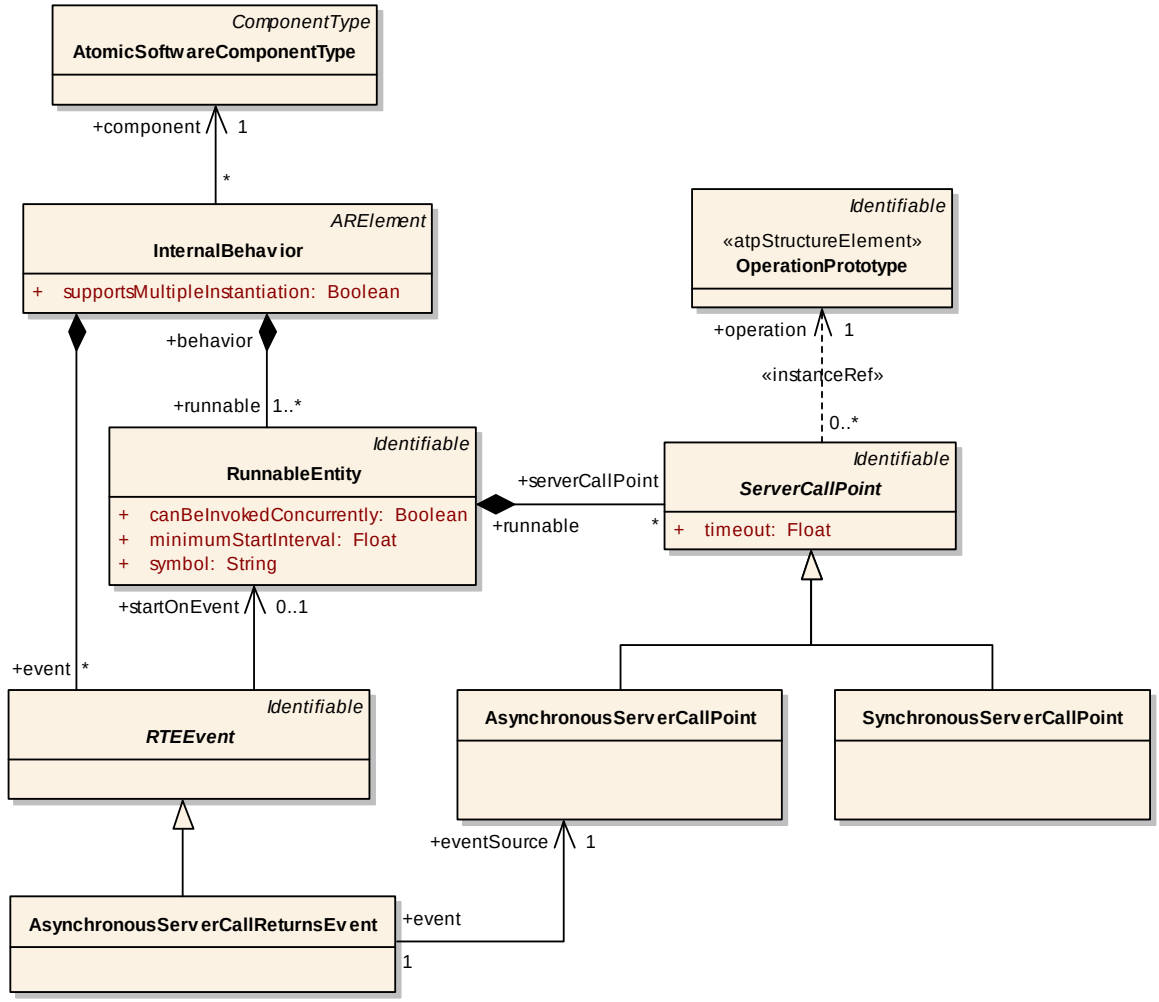

Figure 72: Model of a server call point.

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 "Operation-Prototype" '*remoteOperation*'.
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 “Asynchro-nousServerCallPoint” '*invokeMyOperation*' referencing the respective "Opera-tionPrototype" in the "PortInterface" used to type the "PortPrototype" '*myPort*'. This implies that the "RunnableEntity" is allowed to invoke this operation asyn-chronously.
4. The description of the "AtomicSoftwareComponentType" includes an “Asyn-chronousServerCallReturnsEvent” '*myOperationReturns*' which references the

previously defined "AsynchronousServerCallPoint" 'invokeMyOperation' out of "RunnableEntity" '*main*'.

5. The description of the “AsynchronousServerCallReturnsEvent” '*myOperation-Returns*' references the "RunnableEntity" '*callback*', indicating that the RTE should trigger the execution of this Runnable when '*myOperationReturns*' is raised.

Class	ServerCallPoint {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::ServerCall			
Class Description	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	Attribute Description
operation	ServerCallPoint_operation	1..*	reference to instance	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.

Class	AsynchronousServerCallPoint
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::ServerCall
Class Description	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)	Identifiable, ServerCallPoint

Class	SynchronousServerCallPoint
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::ServerCall
Class Description	This means that the runnable will block for a response from the server.
Base Class(es)	Identifiable, ServerCallPoint

Class	AsynchronousServerCallReturnsEvent			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents			
Class Description	This event is raised when an asynchronous server call is finished.			
Base Class(es)	Identifiable, RTEEvent			
Attribute	Datatype	Mul.	Link Type	Attribute Description

eventSource	AsynchronousServerCallPoint	1	reference	The referenced server call point

7.6.2 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.

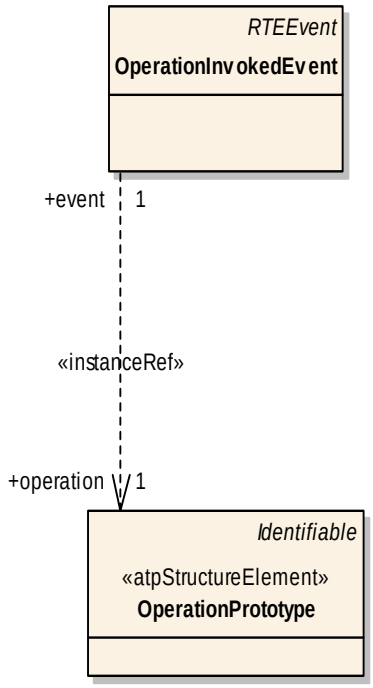

Figure 73: The OperationInvokedEvent references the operation that was called by a client.

Class	OperationInvokedEvent			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents			
Class Description	The OperationINvokedEvent references the operation invoked by the client.			
Base Class(es)	Identifiable, RTEEvent			
Attribute	Datatype	Mul.	Link Type	Attribute Description
operation	OperationInvokedEvent_operation	1	reference to instance	The operation to be executed as the consequence of the event.

7.7 Time Activation of Runnable Entities

In many cases, Runnables do not need to be started by the AUTOSAR OS in response to events related to communication (e.g. the reception of a response to an asynchronous operation invocation) but to timing events. Many Runnables will need to run cyclically with a fixed rate.

The approach taken in the software-component description is to define so-called "TimingEvents" as special kinds of RTEEvents. So far, only one kind of timing event has been defined: a simple "TimingEvent", which has a period as attribute.

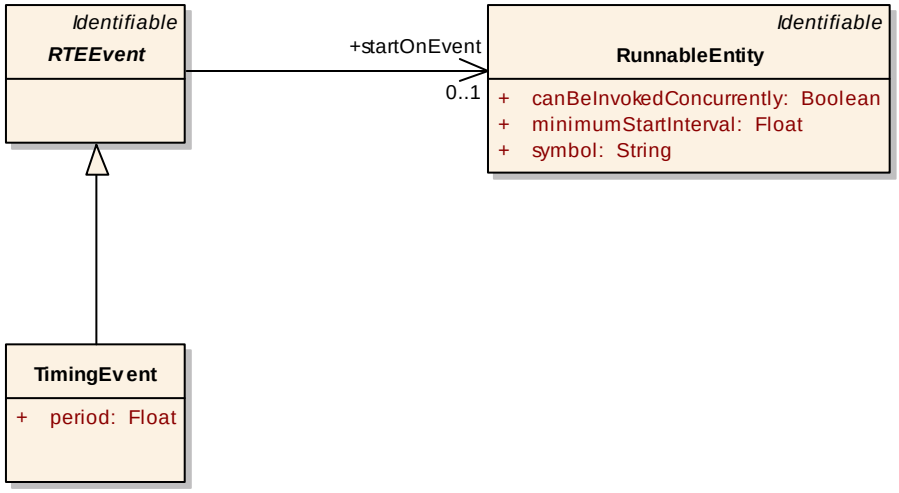

Figure 74: Time Activation of Runnables

When the internal behavior of an atomic software-component requires that the AUTOSAR OS executes certain Runnables periodically, the description will 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	TimingEvent			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::RTEEvents			
Class Description	TimingEvent references the runnable that need to be started in response to the TimingEvent			
Base Class(es)	Identifiable, RTEEvent			
Attribute	Datatype	Mul.	Link Type	Attribute Description
period	Float	1	aggregation	Period of timing event in seconds.

7.8 PerInstanceMemory

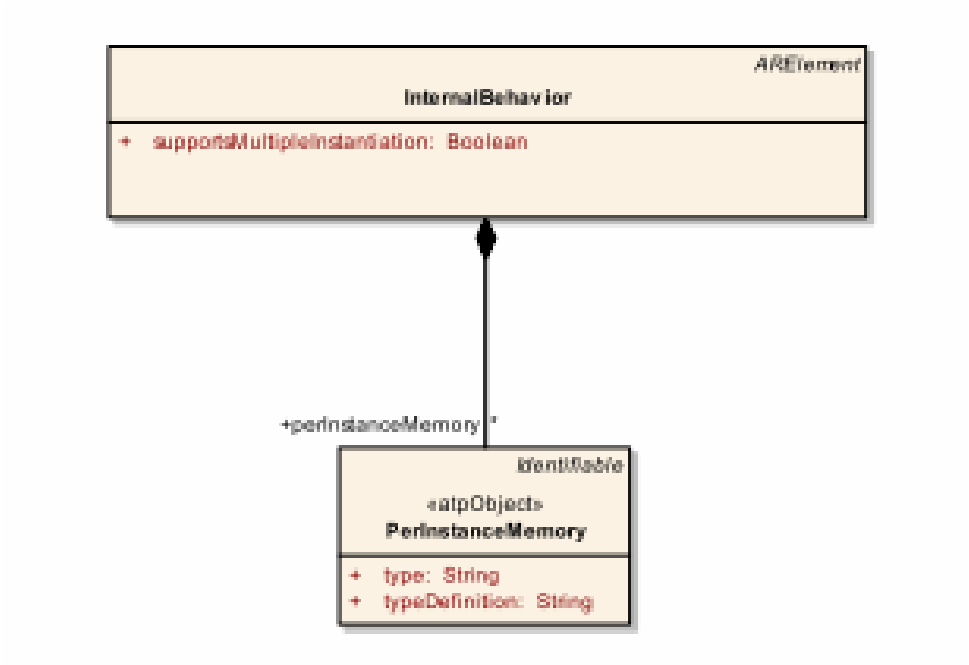

Figure 75: Specifying per-instance memory

Software components that support multiple instantiation (attribute “supportsMultipleInstantiation” == TRUE) will typically need memory per instance. It is the responsability of the RTE to provide a mechanisms with which each instance of a software-component can access its own instance-specific memory[26].
A Software component can define an arbitrary number of per-instance memory blocks.
For each such memory block, the software component must provide the name of the type (the “C”-type), that it wants to store in the memory block. This attribute allows the RTE to generate an API that provides a convenient and type-safe access.
In addition, the software-component must define the “type” in the attribute “typeDefinition”. This attribute must contain a “C” typedef of the “type” in valid C-syntax. This “typeDefinition” must be such that it can be included verbatim in a C header file.
Note that the per-instance memory is not explicitly initialized by the RTE. It is the responsibility of the SW-Component to initialize the per-instance memory.
More details on the use of these attributes in the generation of component header-files can be found in the RTE specification.
Software components that do NOT support multiple instantiation (attribute “supportsMultipleInstantiation” == FALSE) do not need to use the “PerInstanceMemory”: because there will only be a single instance of the software-component on an ECU, the software-component can use static variables to store the component’s state.

[26] Note that the current solution in the SW-C template is “C”-specific. In case programming languages like C++ or Java are used that support instantiation at the language-level, it might be possible to replace the PerInstanceMemory mechanism by built-in features of the programming language.

7.8.1 PerInstanceMemory used for the NvRam Manager

The NvRam Manager can be configured to use a RAM Mirror for the storage of the NvRam Block content. It is the responsibility of the NvRam Manager to provide the content of the NvRam Block inside this Ram Mirror during startup and write back the content to the NvRam during shut-down.
If the SW-Component is using the Ram Mirror feature, a Per Instance Memory section is used for each Ram Mirror. The Per Instance Memory section is allocated by the RTE during ECU Configuration. If the SW-Component is using some NvRam Blocks without the Ram Mirror feature it is the responsibility of the SW-Component to have a memory area available to provide in the API call to the NvRam Manager for storage of the NvRam Block data.
The SW-Component may provide several “NVRAMMapping” elements which define which “PerInstanceMemeory” section shall be used for which “NVRAMPort”.

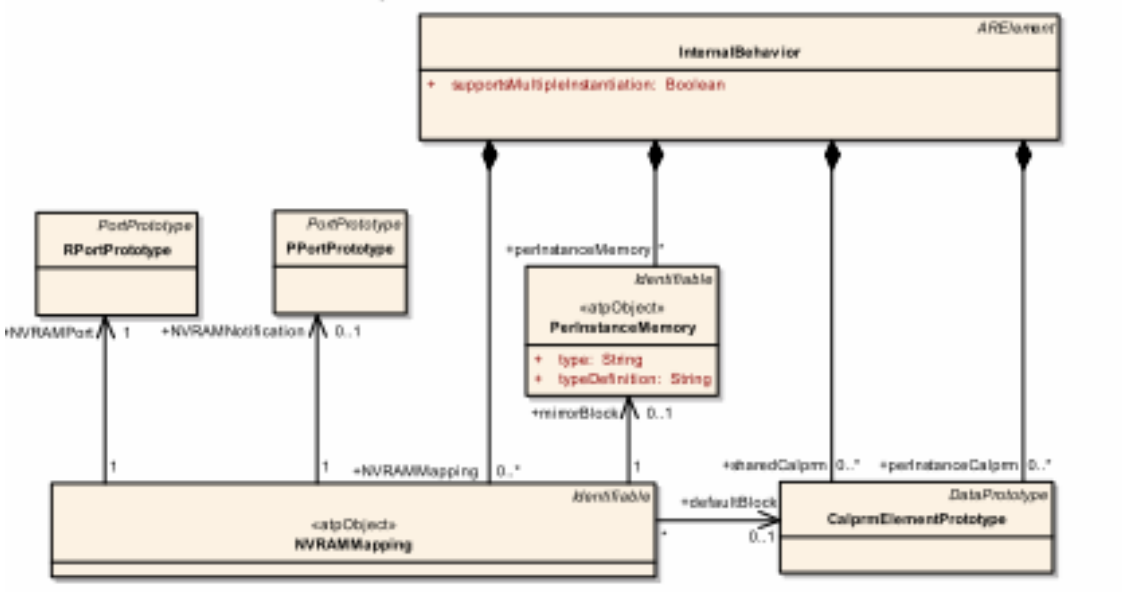

Figure 76: NvRam Manager usage of Per Instance Memory

Class	NVRAMMapping			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::NVRAM-Mapping			
Class Description	Specifies all mappings concerning a single NVRAM block owned by an SWC instance. Note that the mapping is the same for all instances of an SWC type. For details see documentation of the associations.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
defaultBlock	CalprmElementPrototype	0..1	reference	Defines the ROM default for an NVRAM block. This data can be also calibratable.
mirrorBlock	PerInstanceMemory	0..1	reference	Defines the RAM mirror for an NVRAM block.
NVRAMNotification	PPortPrototype	0..1	reference	This port provides the notification sink

				for an NVRAM block in case callbacks shall be configured for this blocks. This must be handled uniformly for all NVRAM blocks which play the same role for all component instances.
NVRAMPort	RPortPrototype	1	reference	This port is requiring operations for a single NVRAM block. It has to be connected to the NVRAM Manager Service at ECU integration time. The port should have annotations which help to configure the NVRAM Service.

7.9 Interaction between Runnables within one Component

It is taken for granted that particular "RunnableEntities" within a specific "AtomicSoftwareComponentType" will need to communicate among each other. This implies that the RTE and/or the AUTOSAR OS 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/AUTOSAR OS 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"

7.9.1 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"[27]. 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 inter-RunnableEntity communication they acquire the semantics of so-called *state-messages*.

[27] 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.

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 Runnable Entity**.
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.

7.9.1.1 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 "RunnableEntites" 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".

7.9.1.2 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.

7.9.1.3 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.

7.9.1.4 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 chapter 7.5.2).
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 77. 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.

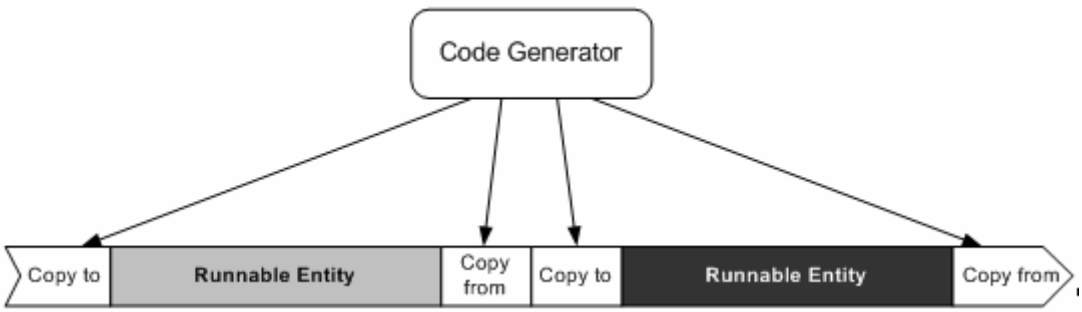

Figure 77: Generation of copy routines around runnable entities

The additional copy routines as sketched in Figure 77 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 78).
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.

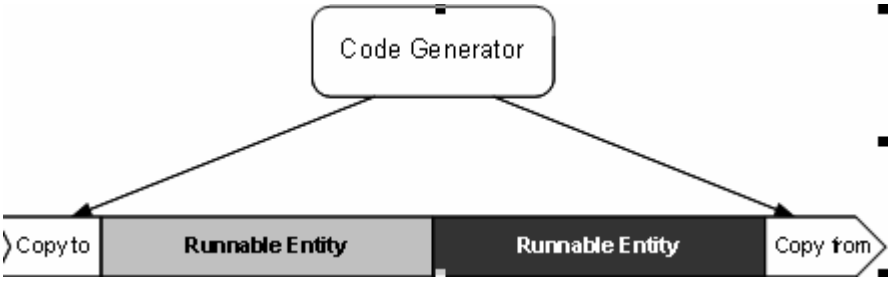

Figure 78: Optimized insertion of copy routines

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 a software-component has to expose all concurrently accessed variables to the outside world.

7.9.2 Description possibility 1: "ExclusiveArea"

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).
They just specify 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 (pre-empt each other) while executing inside the “ExclusiveArea”.

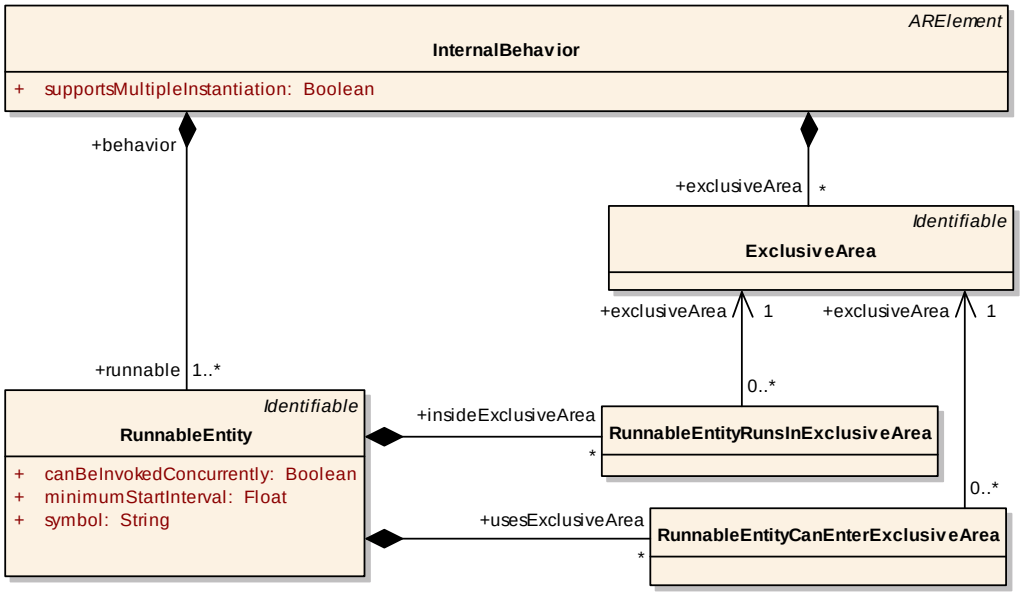

Figure 79: Description of logical exclusive areas

Class	ExclusiveArea
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::ExclusiveArea
Class Description	Prevent runnables from being preempted by other runnables.
Base Class(es)	Identifiable

Class	RunnableEntityRunsInExclusiveArea
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::ExclusiveArea
Class Description	The runnable entity runs inside the referenced exclusive area
Base	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject

Class(es)				
Attribute	Datatype	Mul.	Link Type	Attribute Description
exclusiveArea	ExclusiveArea	1	reference	The referenced ExclusiveArea

Class	RunnableEntityCanEnterExclusiveArea			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::ExclusiveArea			
Class Description	This means that the runnable can enter/leave the referenced exclusive area through explicit API calls.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
exclusiveArea	ExclusiveArea	1	reference	The referenced ExclusiveArea

Please note that there are in general two ways to use the "ExclusiveAreas".

7.9.2.1 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 (in close collaboration with the OS-scheduler) 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/AUTOSAR OS to implement this by using semaphores or mutexes that are taken before the Runnable starts and given when the Runnable returns. It only obliges the RTE to make sure that both Runnables 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 purpose it is necessary to make sure that the OS can only execute a single instance of the task into which the "RunnableEntities" are put. This is the case for the wide majority of embedded AUTOSAR OS on the market.
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".

7.9.2.2 Runnable would Dynamically Enter and Leave the Exclusive Area

In the second approach, the runnable 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 7.9.1.3.
Additionally it is possible to define the execution time the RunnbaleEntity will spend in this ExclusiveArea segment (see section 9.4.5.1).

7.9.3 Description possibility 2: Specifying the Data Exchanged between "RunnableEntities" within the same "AtomicSoftwareComponentType"

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.
This concept can be implemented even with the limited capabilities of an OSEK BCC operating system. This makes it even more attractive for power train and chassis applications.
Concerning the introduction in the AUTOSAR meta-model, data required for communication among runnables 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.

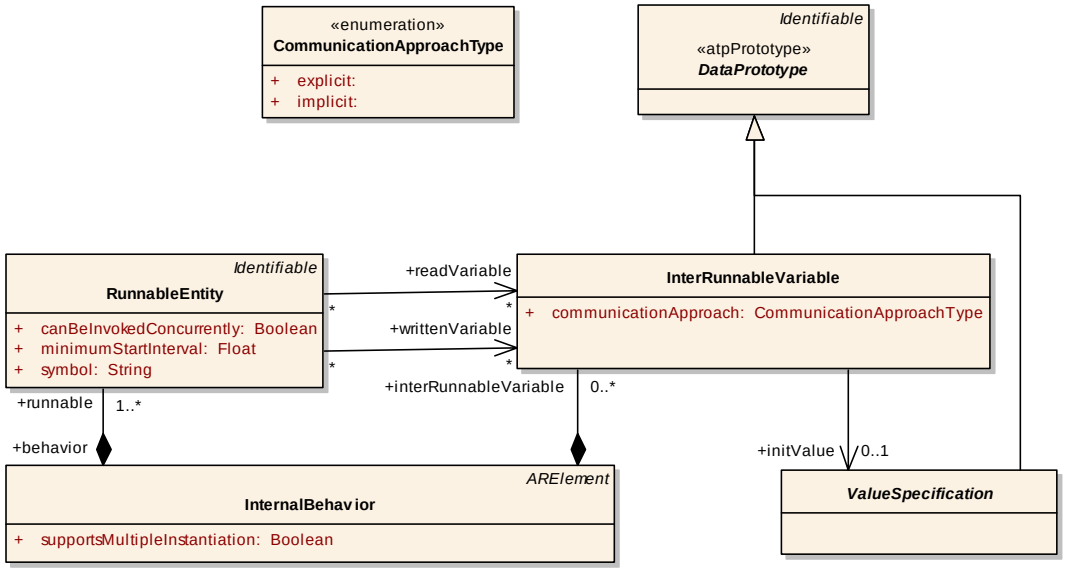

Figure 80: InterRunnableVariable

"InterRunnableVariables" must have a data type; therefore "InterRunnableVariable" is derived from "DataPrototype".
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 80).
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".

Class	InterRunnableVariable

Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::InterRunnableCommunication			
Class Description	Implement state message semantics for establishing communication among runnables of the same component.			
Base Class(es)	Identifiable, DataPrototype			
Attribute	Datatype	Mul.	Link Type	Attribute 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	
measurable	Measurable	1	aggregation	

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 7.9.1.4.

7.10 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 81.

7.10.1 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.

7.10.2 Port Defined Argument Value

In addition to the formal parameters of a client/server invocation that are defined as part of the server's port interface, 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 interface level in form of a formal operation argument, one or more port-defined arguments can be specified. This way, the implementation detail is hidden from the logical component designer.
Figure 81 shows the metamodel 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.

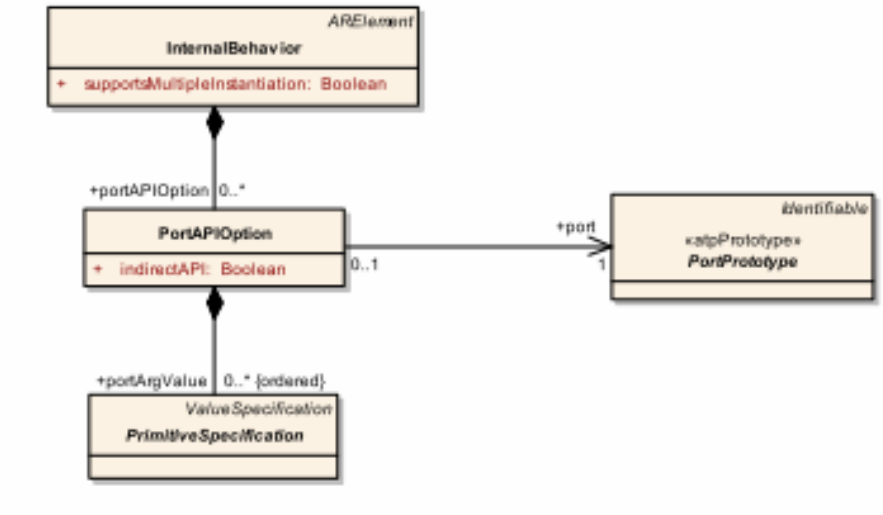

Figure 81: Port API Options.

Class	PortAPIOption			
Package	M2::AUTOSARTemplates::SWComponentTemplate::InternalBehavior::PortAPIOptions			
Class Description	Options how to generated the signatures of calls for an atomic SWC type in order to communicate over a port (for calls into a runable as well as for calls from a runnable to the port).			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
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.
portArgValue	PrimitiveSpecification	0..*	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.
port	PortPrototype	1	reference	the option is valid for generated functions related to communication over this port

8 Component Implementation

8.1 Introduction

This chapter explains, how the implementation details of AUTOSAR software components can be described. While AUTOSAR contains various component types, only atomic software components possess an implementation. In the meta model this means that implementations can be given for “AtomicSoftwareComponentType” or derived classes only.
On the other hand, compositions simply structure and encapsulate their contained components in a hierarchical manner, without adding any implementation relevant behavior or functionality. So they cannot be implemented directly. Instead, the leaf components in such a composition tree, which by definition are again atomic, are implemented.

8.2 Implementation Description Overview

The implementation class shown in Figure 82 serves the following main purposes:

- identify atomic software component that is implemented
- link to code (source code, object code, …)
- specify the build environment, if required (in case of source code for instance)
- specify the compatible runtime environment (software, hardware, resources)

As the diagram shows, “Implementation” is derived from “ARElement”, i.e. it may be shipped as a separate engineering artifact, e.g. independent of the description of interfaces, ports and the component type.

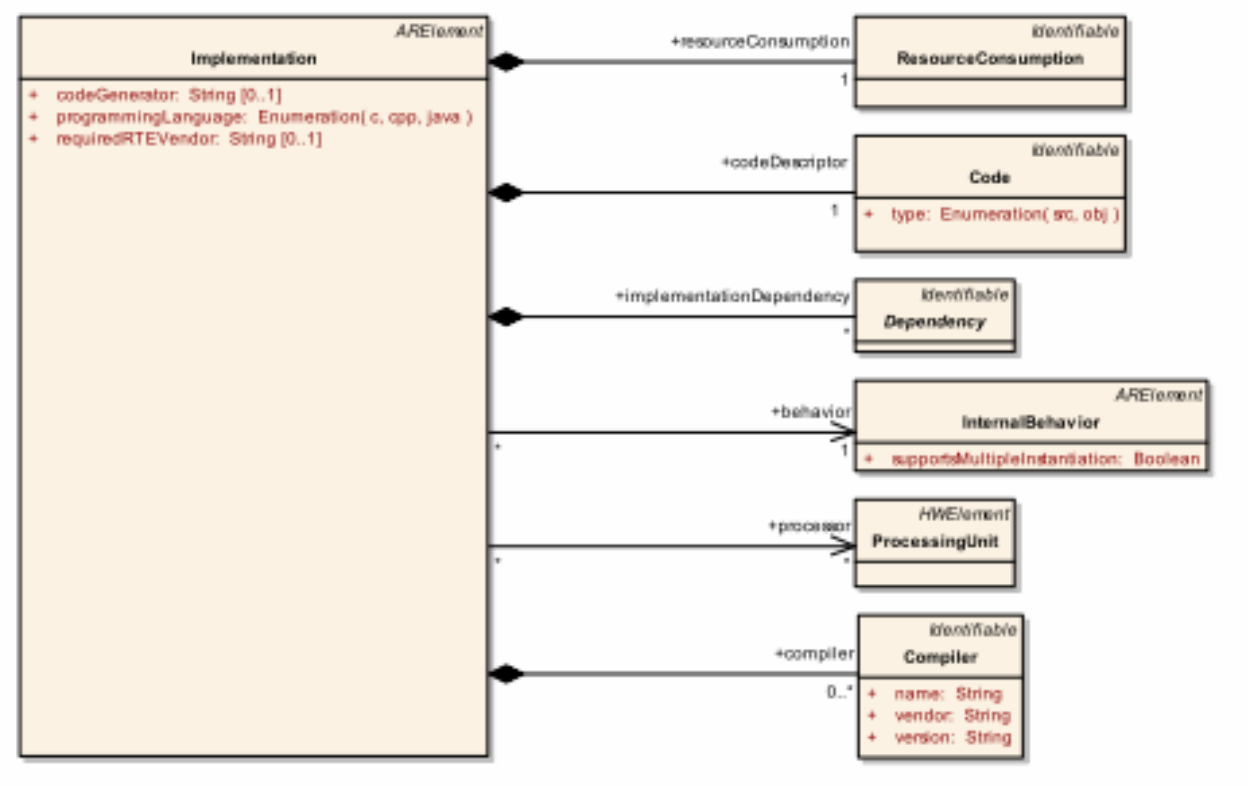

Figure 82: Overview of implementation description.

The following table lists all attributes shown in Figure 82, thereby explaining the meaning of the remaining simple assertions and requirements of class “Implementation”.

Class	Implementation			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Implementation			
Class Description	Description of an implementation for a single atomic software component.			
Base Class(es)	Identifiable, PackageableElement, NonSplitableElement, ARElement			
Attribute	Datatype	Mul.	Link Type	Attribute Description
behavior	InternalBehavior	1	reference	The internal behavior implemented by this Implementation.
codeDescriptor	Code	1	aggregation	
codeGenerator	String	0..1	aggregation	Optional: code generator used.
compiler	Compiler	0..*	aggregation	Specifies the compiler for which this implementation has been released
implementationDependency	Dependency	0..*	aggregation	
processor	ProcessingUnit	0..*	reference	The processor the implementation is compatible with.
programmingLanguage	Implementation-Programming-LanguageEnum	1	aggregation	Programming language the implementation was created in.
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.
resourceConsumption	ResourceConsumption	1	aggregation	All static and dynamic resources for each implementation are described within the ResourceConsumption class.

Class	Compiler			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Implementation			
Class Description				
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
name	String	1	aggregation	Compiler name (like gcc).
vendor	String	1	aggregation	Vendor of compiler.
version	String	1	aggregation	Exact version of compiler executable.

8.3 Assertions and Requirements

For some of the attributes mentioned below it is ambiguous whether they describe a requirement on the target environment or whether they are assertions made by the particular component implementation. The implementation description's "compiler" attribute is an example for this: does it describe a requirement for source code to be compiled with the named compiler, or is this simply information which compiler was used in the process of creating an object file?
The simple answer is: if possible, this is derived from the context. Otherwise the attribute needs to have proper documentation.
For the "compiler" example just mentioned, the situation is straightforward: for source code, the attribute describes a requirement, for object code it is historic information. The same needs to be applied to all attributes in this section.

8.4 Implementation of an Atomic Software Component

Probably the most important information in "Implementation" is which atomic software component is actually implemented. At first glance, this link seems to be missing in

the overview in Figure 82. However, implementations are actually given for a particular component behavior, specified through the class “InternalBehavior”. The contents of such a behavior are not of interest here, but as Figure 83 shows, it in turn is associated with a single atomic software component.
This way an implementation is in fact given for a particular atomic component.

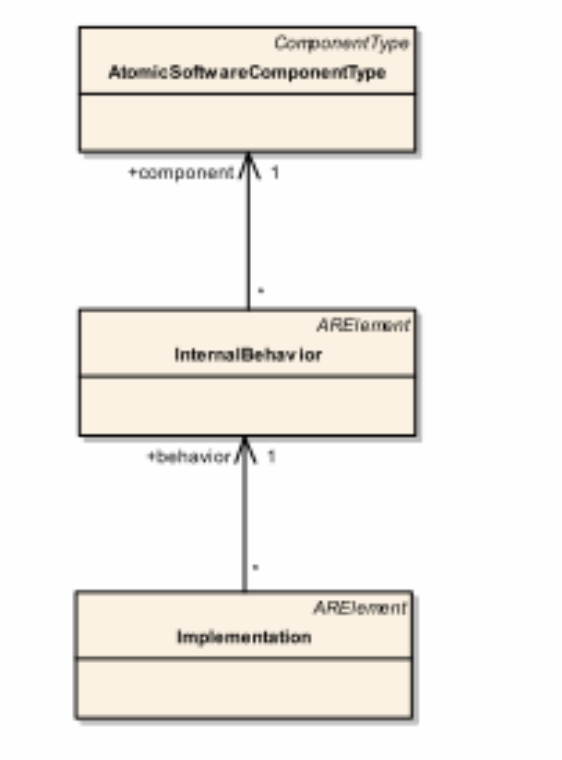

Figure 83: An implementation is associated with a single atomic software component.

8.5 Linking to Code

When a component is released by the supplier, the descriptions are accompanied by actual implementation code. This code can come in different ways: source code in C, C++ or Java, object code or even executable code[28].
Figure 84 shows how an implementation is linked to code files. For each available form of component code a “Code” is used. If for instance a component implementation is given as source code only, then the respective “Implementation” would contain exactly one “Code”, whose “type” attribute would have been set to “src”.
For each code description, all relevant files are then referenced in form of a standard URL. For relative URLs the path will start at the containing XML file.

[28] How such linked code would be embedded in the ECU is not in scope of the template specification. If required further attributes need to be added at a later time to support such a process.

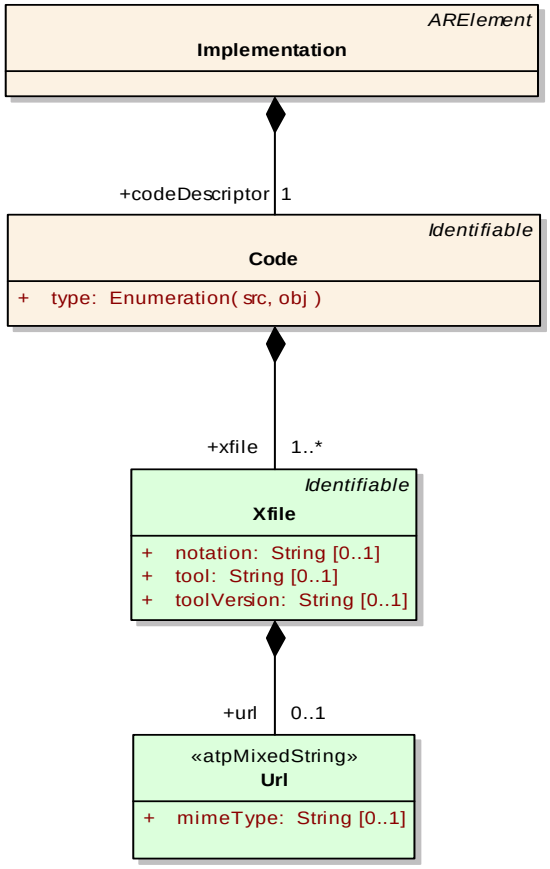

Figure 84: An implementation references the code files through the Code class.

Class	Code			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Implementation			
Class Description	A generic code descriptor.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
type	CodeTypeEnum	1	aggregation	The type of described code.
xfile	Xfile	1..*	aggregation	The files belonging to this code descriptor.

8.6 Resource Consumption

The contents of this class describe the resources consumed by a particular implementation, in fact within a very specific context. The details are described in chapter 9.

8.7 Dependencies

The one attribute left open for discussion is the “implementationDependencies”. By specifying dependencies an implementation can depend on certain other artifacts or model features. The real meaning of this is detailed by particular *kind* of dependency, as shown in Figure 85.

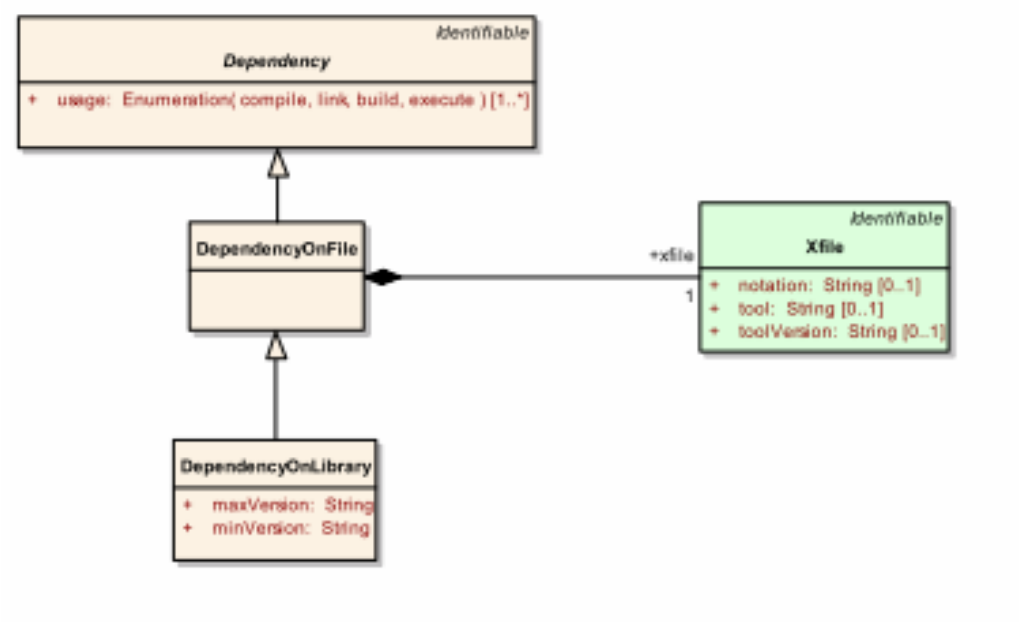

Figure 85: Dependencies of an implementation.

Depending on files (incl. libraries)
An implementation can generally depend on files. Such files could for example be required header files or configuration files. The URL points to the place where the files is expected, or simply contains the name of the file, in case the path is not relevant. For libraries, like e.g. a “math.lib”, a minimum and maximum version number can be specified, therefore trying to ensure compatibility.
Note, that the specification of version numbers is a meta-information about certain artifacts, for which a more general solution may be found in the future (e.g. as part of a catalog description). So the current solution has to be seen as a first and rough approach only.

Class	Dependency {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Implementation			
Class Description	General dependency, typically on the existence of another artifact.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
usage	DependencyUsageEnum	1..*	aggregation	Specification during for which process step(s) this dependency is required.

Class	DependencyOnFile			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Implementation			
Class Description	Dependency on the existence of a certain file.			
Base Class(es)	Identifiable, Dependency			
Attribute	Datatype	Mul.	Link Type	Attribute Description
xfile	Xfile	1	aggregation	The specified file needs to exist.

Class	DependencyOnLibrary			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Implementation			
Class Description	A specific file dependency: without the library that implementation cannot be used (compiled, linked, executed, ...).			
Base Class(es)	Identifiable, Dependency, DependencyOnFile			
Attribute	Datatype	Mul.	Link Type	Attribute Description
maxVersion	String	1	aggregation	Maximum version compatible with implementation. If not set, there is limitation on the upper version.
minVersion	String	1	aggregation	Minimum version compatible with implementation.

8.8 Multiple Instantiation of Components

8.8.1 Single Instantiation

The single instantiation of a component implementation represents the simplest case of instantiation and shall be described as a starting point for the discussion of multiple instantiation.
An instance of a component implementation consists of "code" and "data". The term "data" can be further subdivided into data relevant for the behavior of the software-component and data elements and arguments of the software-comonent's ports,

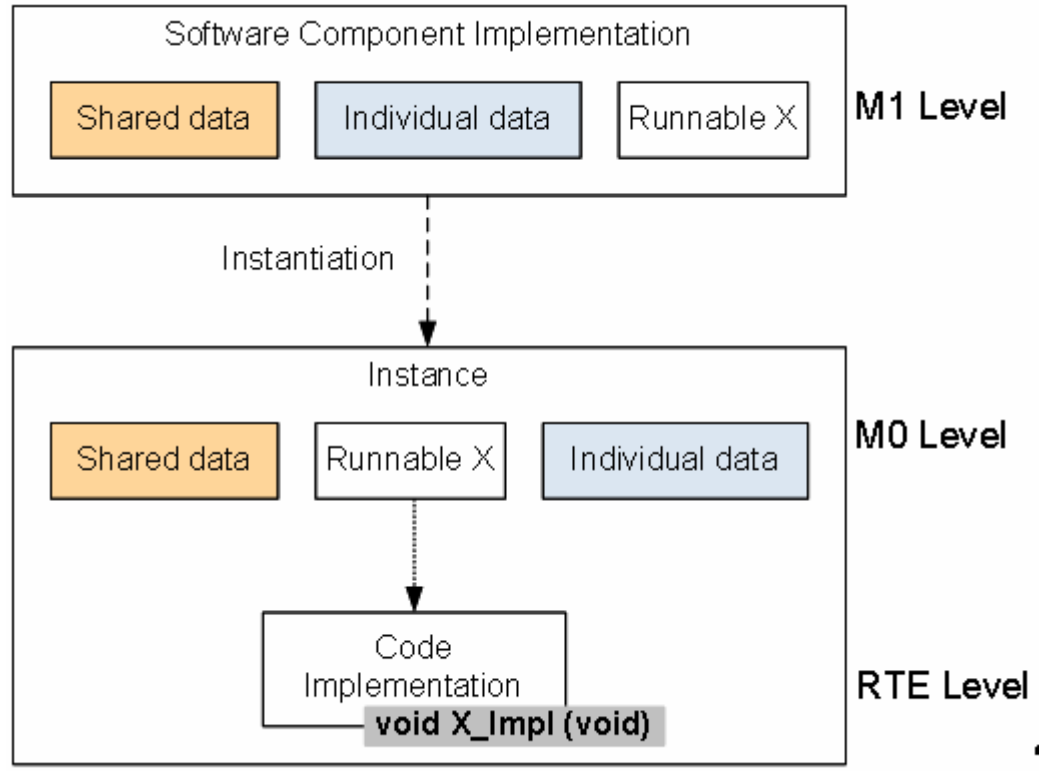

Figure 86: Single instantiation of a software component implementation

The single instantiation principle is sketched Figure 86. In this scenario merely a single instance of the component implementation is taken into account. It is therefore in general possible to closely couple the code to the individual data of the instance.

8.8.2 Multiple Instantiation

Multiple instantiation is a very powerful concept that nevertheless imposes some important consequences on the general concept of implementing AUTOSAR atomic software components.

8.8.2.1 Sharing of Code

The term instantiation resembles to the vocabulary of object-oriented modeling concepts. One of the major characteristics of the object-oriented approach (concerning the implementation on the code level) is to share the code representing a specific functionality among all instances of a particular class.
In the case of component implementation the code representing a runnable of a component implementation would be shared among all instances. Please note that code that can be shared among instances is different from code that is individually accessible for each instance.
The difference can be exemplified by a simple example: two instances are created. As a consequence it is no longer possible to couple the code tightly to the individual data of the instances (see Figure 87).

During run-time, the individual instance-specific information must be administrated properly and applied to the code such that an operation of the specific instance can be executed.
For this purpose additional "wrapper code" must be added such that it is possible to pass the individual instance data as an argument to the code (see Figure 87).
In this example implementation, the individual data of the component implementation are gathered in a single structure definition. Each instance defines a variable of this type for storing the individual information. A pointer to the instance-specific variable is passed to the code when the code is executed for the specific instance.

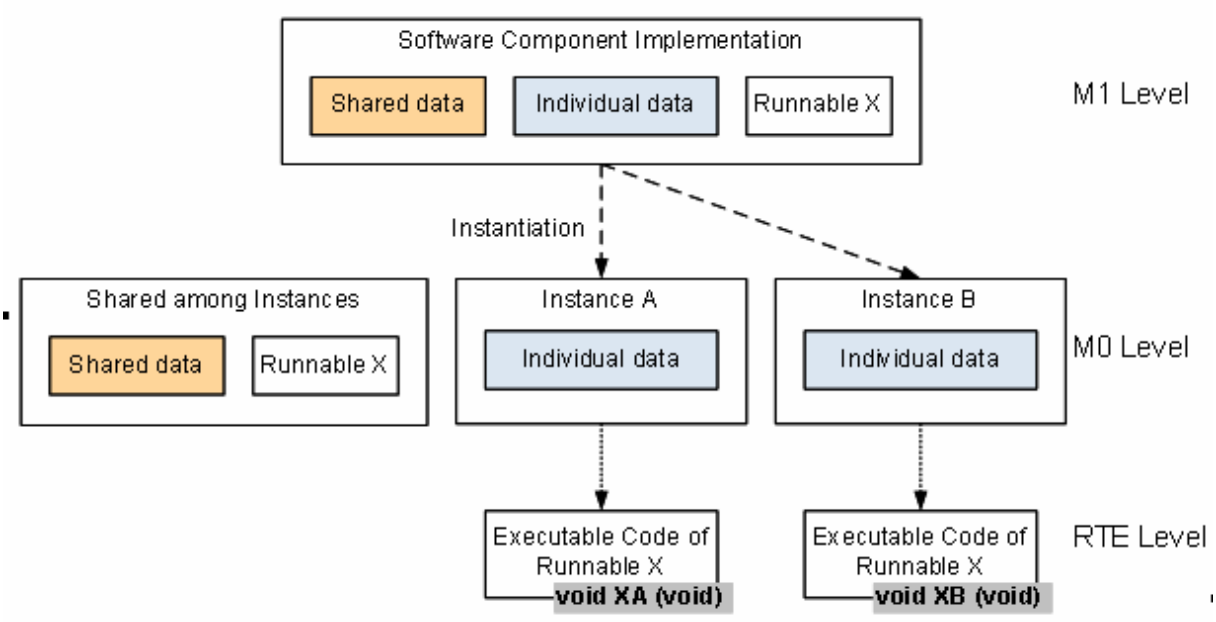

Figure 87: Scenario for code sharing

This model is a potential candidate for being adopted for implementing the AUTOSAR RTE. In this case the runnables of a SW-C are shared among all instances created from the SW-C. Of course, this applies only if the instances are located on the same ECU.

8.8.2.2 Duplicating of Code

On the other hand, it is certainly possible to introduce a different understanding of multiple instantiation: the existence of multiple instances is limited to the M0 model level. On the level of the RTE, all instances of the SWC would be separately handled as if the relationship based on the instantiation from a common SW-C does not exist.
This implies the fact that code representing the runnable entities of the corresponding component would typically **not** be shared by all instances. This approach is sketched in Figure 88.

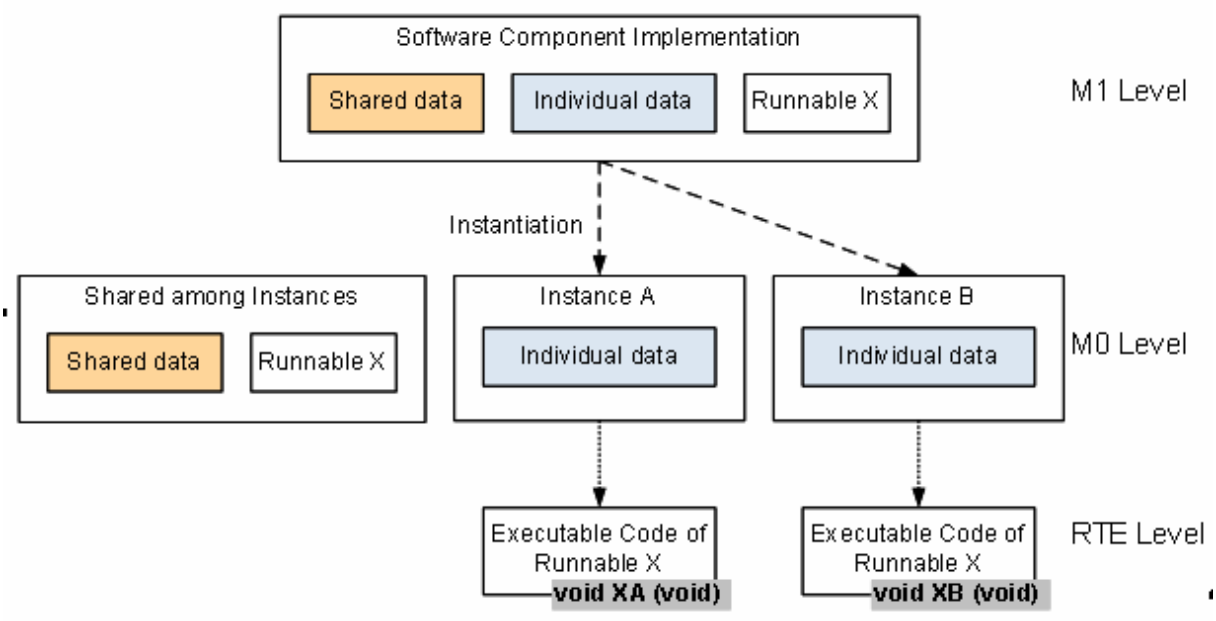

Figure 88: Duplication of code

Please note that all the functions (i.e. XA and XB) depicted Figure 88 necessarily have to be exact duplicates of each others. The names of the functions, however, obviously have to be different in order to avoid linking problems.
The identity of the code is a consequence of the meta-model. Different implementations have to be created if the code was not identical among instances. But please note that in this case different component implementation would be created and this would not be related to the scenario depicted in Figure 88.

8.8.2.3 Resource Consumption

Multiple instantiation certainly has an impact on the resource consumption, especially with respect to memory. Please refer to ch. 9.2 for a detailed discussion of memory resource description.

8.8.3 Decision on Approach for Multiple Instantiation

The decision of the approach to be applied for instantiation (as described in ch. 8.8.2) shall be taken by the developer of the SW-C rather than arbitrarily by any code generation or configuration tool.
Therefore, it is necessary to introduce a flag "supportsMultipleInstantiation" on the level of "InternalBehavior" that controls the process of multiple instantiation to take either one or the other approach for realizing component implementations on the level of the RTE.

8.8.4 Roles of Ports

In general, the roles of ports of a SW-C can be different depending on the instantiation scenario. For example, a port receiving a bus signal in a particular instance can be configured to receive a locally generated sensor value in another instance. This scenario is depicted in Figure 89.

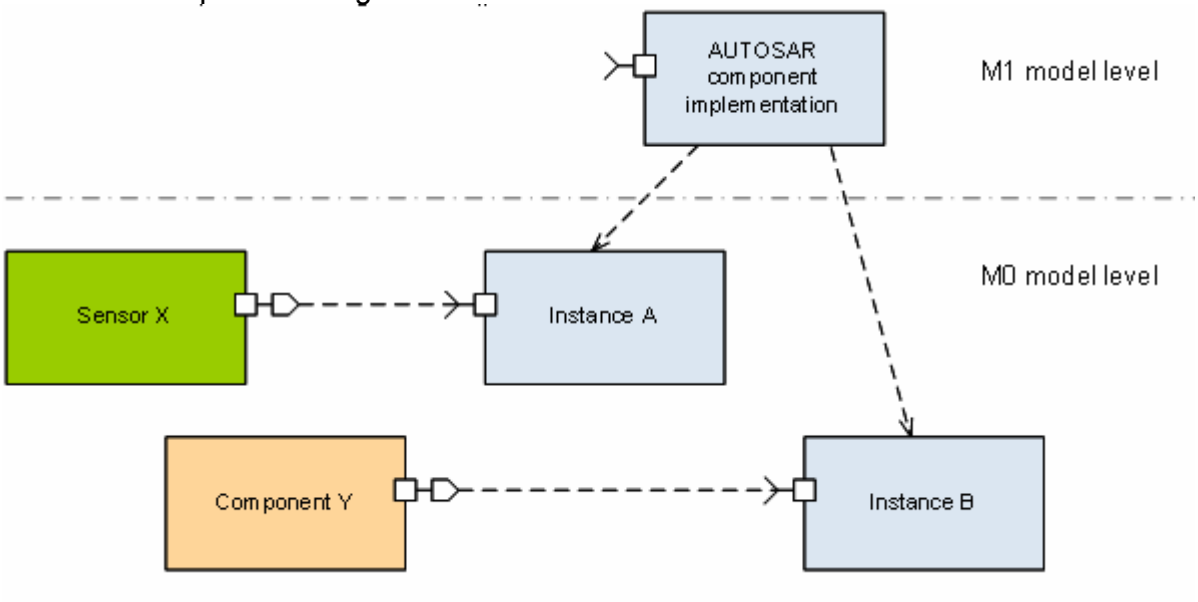

Figure 89: Role of ports

The roles of ports are determined from outside the SW-C and thus are not known at the time the SW-C is designed. In case the code of runnables is shared among instances it is necessary to provide additional inter-communicable code for accessing a port according to the role of the port in a particular instance.
In other words: the component implementation is supposed to provide code for accessing particular ports in all roles that come into question.

9 Resource Consumption of AUTOSAR SW-Components

9.1 SW-Component Resources

AUTOSAR SW-Components need to be mapped on ECUs at some point during the development. Atomic AUTOSAR SW-Components can be basically mapped to any ECU available within the car. The mapping freedom is limited by the "System Constraints" and the available resources on each ECU.
The SW-Component description provides information about the needed resources concerning memory and execution time for each "AtomicSoftwareComponentType".
Each ECU provides only a limited amount of memory and calculation power. These are called hardware resources. The hardware resources are going to be used by all software on that ECU, including OS, Basic SW, RTE, ECU abstraction, CCD, Services and the mapped "ComponentPrototypes". Therefore, only a limited amount of the hardware resources is actually available for the mapped "ComponentPrototypes". These resources are called available resources.
The resource consumption of the other software on an ECU (OS, RTE, Basic SW,...) is not covered by the AUTOSAR SW-Component template explicitly although the template might be used to capture the memory and execution time consumption of a specific configuration of the Basic SW.
And even some of these resources are highly dependent on the configuration actually mapped on the ECU. So an iterative resource description and estimation is needed to cover the RTE and Basic SW resource needs.

Types of resources
Resources can be divided into static and dynamic resources.
Static resources can only be allocated by one entity and stay with this entity. If the required amount of resources is bigger than the available resources the mapping does not fit physically. ROM is an example of a spare resource where obviously only the amount of data can be stored that is provided by the storage capacity.
Dynamic resources are shared and therefore can be allocated dynamically to different control threads over time. Processing time is a good example, where different "RunnableEntities" are given the processor for some time.
If some runnable entity uses more processing time than originally planned, it can lead to functional failure. Also some sections of RAM can be seen as dynamic resources (e.g. stack, heap which grow and shrink dynamically).

Resource consumption overview
In Figure 90, the meta-model of the resource consumption descriptions is depicted. The resource consumption is attached to an "Implementation" of an "AtomicSoftwareComponentType". For each "Implementation", there can be one "ResourceConsumption" description.

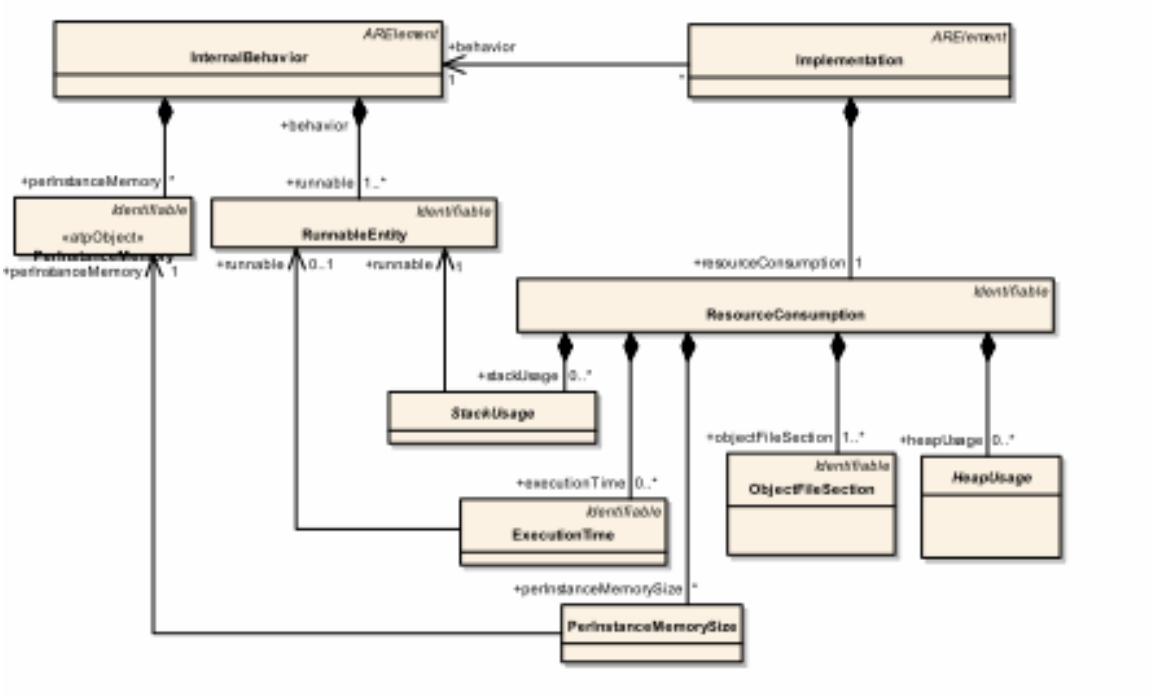

Figure 90: Resource consumption overview

As depicted by Figure 90, all resources are described within the "ResourceConsumption" meta-class.
”ExecutionTime” and “StackUsage” are used to provide information on the implementation-specific resource usage of the runnables defined in the “InternalBehavior”.
“PerInstanceMemorySize” provides information regarding the actual size (in bytes) of the “PerInstanceMemory” defined in the “InternalBehavior”.
“ObjectFileSection” documents the resources needed to load the object-file containing the implementation on the ECU.
“HeapUsage” describes the dynamic memory usage of the component.

Class	ResourceConsumption			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources			
Class Description	Description of consumed resources by one implementation of a software component.			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
executionTime	ExecutionTime	0..*	aggregation	Collection of the execution time descriptions for the runnable entities of this implementation.
heapUsage	HeapUsage	0..*	aggregation	Collection of the heap memory allocated by this implementation.
objectFileSection	ObjectFileSection	1..*	aggregation	Provides additional information to the sections of the object-file containing the implementation of the SW-Component
perInstanceMemorySize	PerInstanceMemorySize	0..*	aggregation	Allows a definition of the size of the per-instance memory for this implementation

stackUsage	StackUsage	0..*	aggregation	Collection of the stack memory usage for each runnable entity of this implementation.

In the next sections a detailed description of the different kinds of resource consumption descriptions will be provided.

9.2 Static Memory Needs

9.2.1 Content

This subchapter describes how the static memory needs of the implementation of an atomic software-component are described.
This includes all memory needs of a component for code or data both at the class and at the instance level *except* for:

- stack space needed in the task that activates a runnable of the implementation (see chapter 9.3)
- dynamic heap-behavior of the component (in case the component uses „malloc“/“free“ to get/free buffers from the heap, see chapter 9.3)[29]

9.2.2 Resources Needed to Load the Object File on the ECU

Memory will be needed to load the object-file containing an implementation of the component on an ECU. The SW-Component Template makes it possible to list the different sections in the object-file that need to be stored on the ECU using the “ObjectFileSection”.

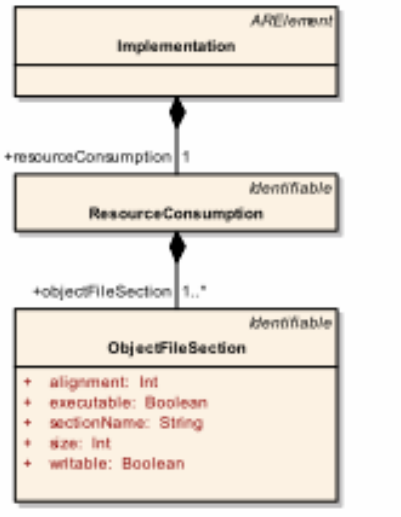

Figure 91: Meta-model related to the “ObjectFileSection”

[29] This is often problematic in embedded and real-time systems: many components will only need static memory blocks and stack-size but will not require dynamic memory allocation

The approach described above is consistent with the approach used in common link-formats like COFF or ELF. What is similar to COFF/ELF is the concept of named memory-sections that have a size and alignment and the presence of the attributes „executable“ and „writable“
Note that the definition of an "ObjectFileSection" does **not** imply the creation of a dedicated container data structure along with a specific API for accessing the container in the component's source-code.
Typical object file sections will be:

- “code”: this corresponds to the memory needed to store the compiled code of the component
- “read only memory”: this typically corresponds to the constants used in the implementation of the component
- “writable memory”: this will typically only occur when static variables are used. This is normally only allowed for implementations of component that cannot be multiply instantiated.

The attributes of the meta-class "ObjectFileSection" are shown below:

Class	ObjectFileSection			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::ObjectFileSection			
Class Description	The objectFileSection provides additional information to the different sections in the object-file containing the implementation of the SW-C			
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
alignment	Integer	1	aggregation	The alignment (typically 1, 2, 4,...)
executable	Boolean	1	aggregation	Is code stored in this section that needs to be executed. In some processor architectures code execution is only allowed from specific memory sections.
sectionName	String	1	aggregation	This is the name of the section in the object-file
size	Integer	1	aggregation	The size in bytes of the section.
writable	Boolean	1	aggregation	Is it possible to write this section.

9.2.3 Information on the Size of Per-Instance Memory

In addition, it is possible to define the actual size (in bytes) and alignment needed for the “PerInstanceMemory” defined in the “InternalBehavior”.

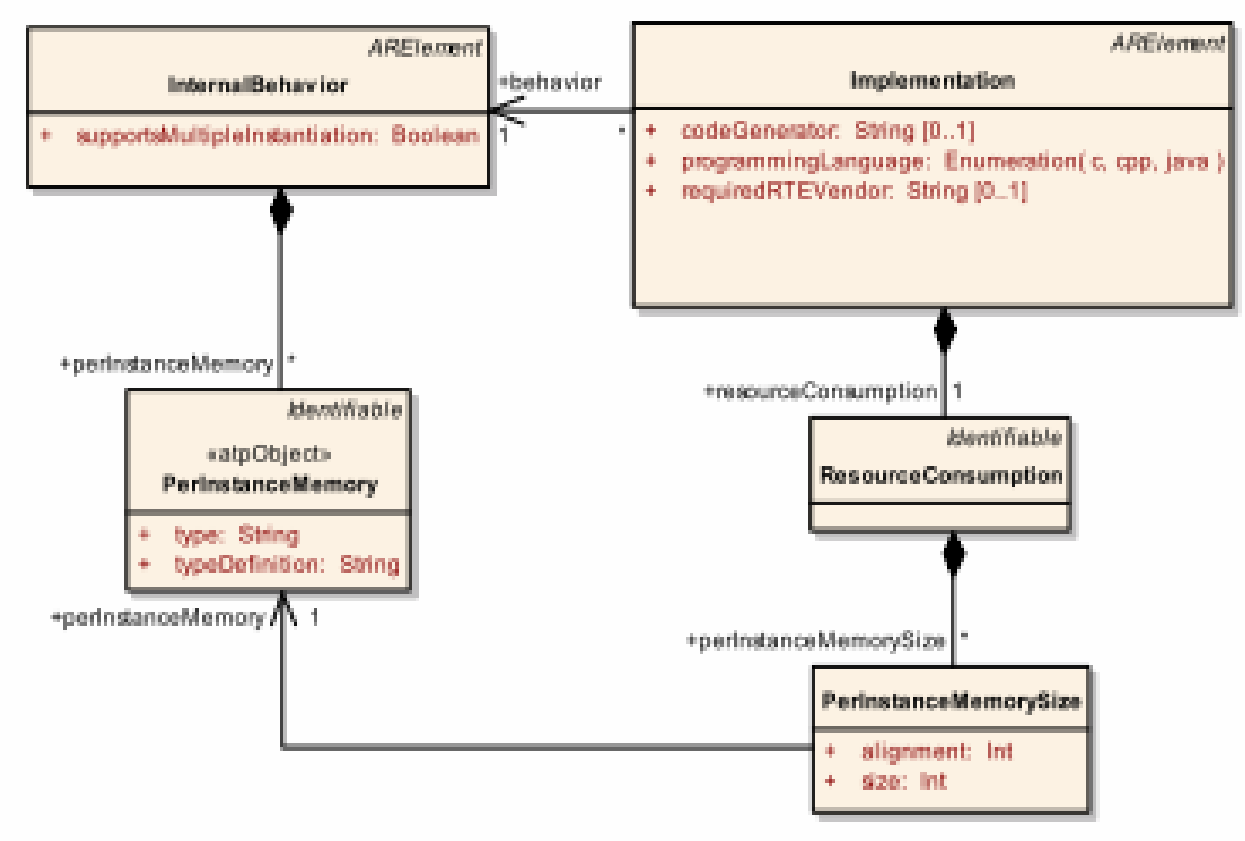

Figure 92: Meta-model related to the ObjectFileSection

Class	PerInstanceMemorySize			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::PerInstance-MemorySize			
Class Description	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)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute 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

9.3 Dynamic Memory Needs

9.3.1 General

The dynamic memory is mainly divided into two categories, the stack and the heap. While the stack is almost always used in embedded software, the heap is avoided as

much as possible due to the complexity of its implementation and segmentation issues.
The dynamic memory consumption of a SW-Component has a much different quality than the static memory consumption. The amount of the static memory consumption can be retrieved from the compiler and is only dependent on the compiler and processor used as well as on the number of instances.
Dynamic memory consumption is heavily dependent on the actual code being executed, which is dependent on the state of the software and the parameters. With the introduction of recursive concepts the uncertainty is even higher. Therefore the approach for dynamic memory consumption is far more related to the description of the execution time introduced in chapter 9.4.
The current meta-model on resource consumption is shown in chapter 9.1.
Of course the memory needs are highly dependent on the software context, which represents the state and the input to the SW-Component and sometimes also on the hardware configuration.

9.3.2 Stack

The stack is an area in memory that is used to store temporary information like parameters and local variables of function calls. Therefore the stack usage is highly dependent on the calling hierarchy and the nesting level of function calls.
The stack is organized in a LIFO (last in first out) manner. So each time a function is called the necessary stack memory is occupied. After leaving the function also the associated memory area is freed again and can be used for the next function call. Therefore segmentation is not a problem for a stack. Only the available amount of stack memory is relevant from the SW-Component view point.
Different mechanisms can be used to describe the stack memory needs of a SW-Component. Needed stack size can either be calculated, measured or estimated. This is shown in Figure 93.

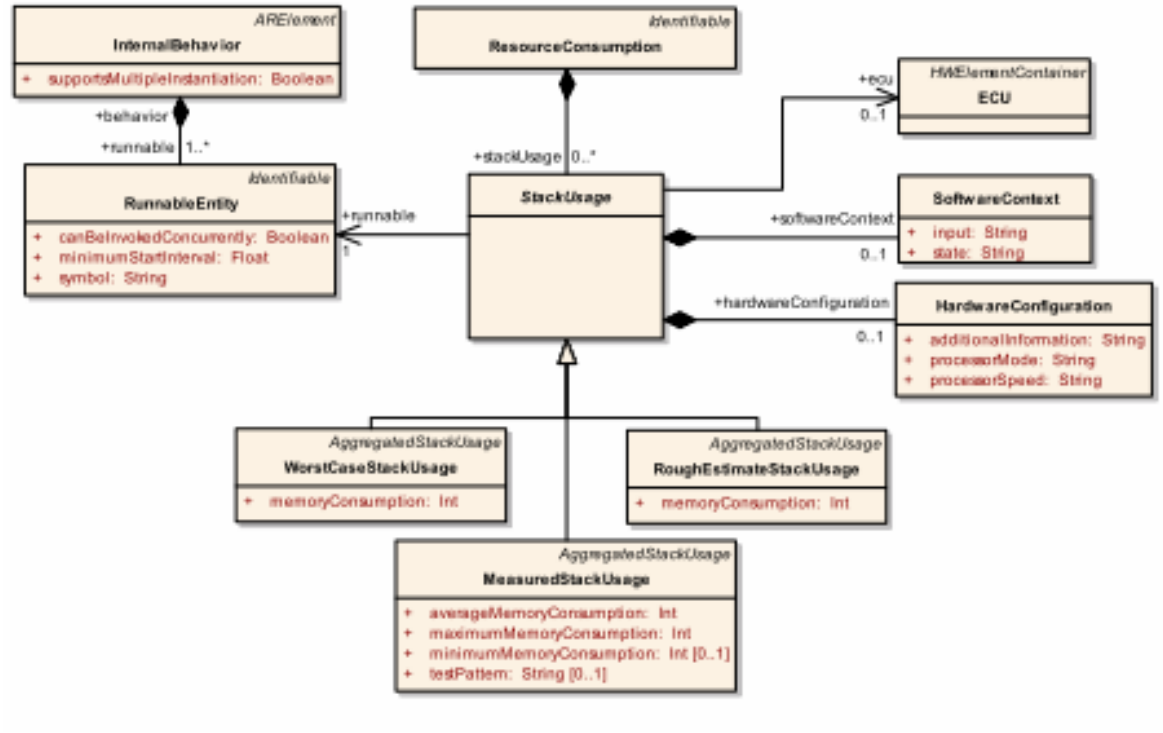

Figure 93: Stack Memory Consumption

The given stack memory consumption is dependent on the ECU, the software context and maybe also on the hardware configuration. The software context and the hardware configuration describe the state of the software and hardware under which the given stack usage was gathered. So for each given stack memory consumption these environmental descriptions have to be provided.

Class	StackUsage {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::StackUsage			
Class Description	Describes the stack memory usage of a software component			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
ecu	ECU	0..1	reference	Reference to the ECU description this implementation is provided for.
hardwareConfiguration	HardwareConfiguration	0..1	aggregation	Contains information about the hardware context this stack usage is describing.
runnable	RunnableEntity	1	reference	Reference to the runnable this stack usage is provided for.
softwareContext	SoftwareContext	0..1	aggregation	Contains details about the software context this stack usage is provided for.

Class	WorstCaseStackUsage
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::StackUsage
Class Description	Provides a formal worst case stack usage.
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, StackUsage,

	AggregatedStackUsage			
Attribute	Datatype	Mul.	Link Type	Attribute Description
memoryConsumption	Integer	1	aggregation	Worst case stack consumption.

Class	MeasuredStackUsage			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::-StackUsage			
Class Description	The stack usage has been measured.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, AggregatedStackUsage, StackUsage			
Attribute	Datatype	Mul.	Link Type	Attribute Description
averageMemoryConsumption	Integer	1	aggregation	The average stack usage measured.
maximumMemoryConsumption	Integer	1	aggregation	The maximum stack usage measured.
minimumMemoryConsumption	Integer	0..1	aggregation	The minimum stack usage measured.
testPattern	String	0..1	aggregation	Description of the test pattern used to aquire the measured values.

Class	RoughEstimateStackUsage			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::StackUsage			
Class Description	Rough estimation of the stack usage.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, AggregatedStackUsage, StackUsage			
Attribute	Datatype	Mul.	Link Type	Attribute Description
memoryConsumption	Integer	1	aggregation	Rough estimate of the stack usage.

9.3.3 Heap

Heap is the memory segment that is used to cover dynamic memory needs with explicit memory allocation and de-allocation. Since the allocation of the memory is controlled by the application program it also survives changes in the scope of invocation from entering a function nesting level and leaving it again. So a memory block allocated in the subroutine can be used in the calling routine after the subroutine has returned. Also the allocated memory can be freed again in different places.
Because of the independence of the heap consumption from processes and tasks only the SW-Component heap consumption is provided in the SW-Component template. Here the heap consumption is considered in the SW-Component that allocates the heap memory. So if one SW-Component allocates a heap memory and passes it onto another SW-Component the heap consumption will only be described in the allocating SW-Component's Resource Consumption description.
The meta-model is shown in Figure 94.

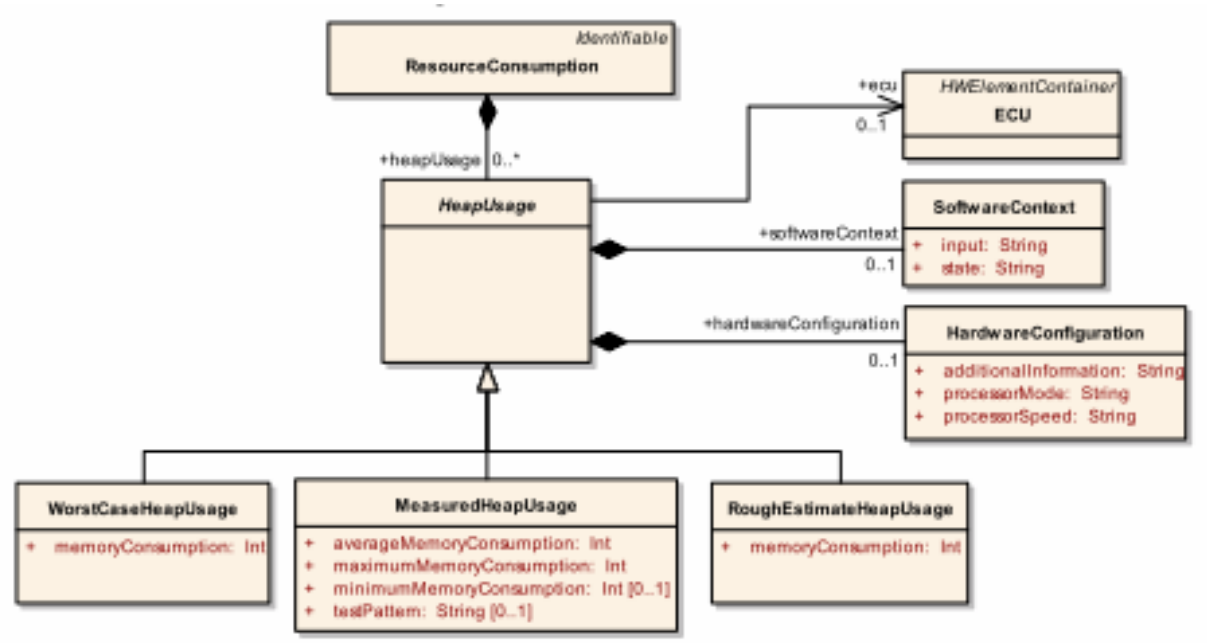

Figure 94: Heap memory consumption

The heap memory consumption also depends on the ECU, the software context and the hardware configuration.
Due to the highly dynamic nature of heap memory one problem is the segmentation of the available memory area. So in some cases there can be not enough memory allocated, even though the total amount of free heap memory is big enough, because the available memory space is not available continuously.

Class	HeapUsage {abstract}			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::HeapUsage			
Class Description	Describes the heap memory usage of a SW-Component.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
ecu	ECU	0..1	reference	Reference to the ECU description this implementation is provided for.
hardwareConfiguration	HardwareConfiguration	0..1	aggregation	Contains information about the hardware context this heap usage is describing.
softwareContext	SoftwareContext	0..1	aggregation	

Class	WorstCaseHeapUsage			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::HeapUsage			
Class Description	Provides a formal worst case heap usage.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, HeapUsage			
Attribute	Datatype	Mul.	Link Type	Attribute Description
memoryConsumption	Integer	1	aggregation	Worst case heap consumption.

Class	MeasuredHeapUsage

Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::-HeapUsage			
Class Description	The heap usage has been measured.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, HeapUsage			
Attribute	Datatype	Mul.	Link Type	Attribute Description
averageMemoryConsumption	Integer	1	aggregation	The average heap usage measured.
maximumMemoryConsumption	Integer	1	aggregation	The maximum heap usage measured.
minimumMemoryConsumption	Integer	0..1	aggregation	The minimum heap usage measured.
testPattern	String	0..1	aggregation	Description of the test pattern used to aquire the measured values.

Class	RoughEstimateHeapUsage			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::HeapUsage			
Class Description	Rough estimation of the heap usage.			
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject, HeapUsage			
Attribute	Datatype	Mul.	Link Type	Attribute Description
memoryConsumption	Integer	1	aggregation	Rough estimate of the heap usage.

9.4 Execution Time

9.4.1 Content

This subchapter defines a model to describe the “execution time” of a specific runnable entity of a specific implementation of an atomic software-component.
Section 9.4.3 describes the goals and scope of the "execution time" description proposed.
Section 9.4.4 lists all the thoughts and observations that lead to the actual model which is described in section 9.4.5.

9.4.2 Preliminaries

This subchapter assumes that the reader is familiar with the definition of the following terminology (please see the AUTOSAR glossary after V1.2 for details):

- task
- thread
- process
- runnable entity
- (worst case) execution time
- (worst case) response time

9.4.3 Scope

9.4.3.1 Assertions Versus Requirements

The execution time is an **ASSERTION**: a statement about the duration of the execution of a piece of code in a given situation. The execution time is **NOT A REQUIREMENT** on the software-component, on the hardware or on the scheduling policy.

9.4.3.2 In Scope

This section proposes a description of the execution time of a runnable entity of an implementation of an atomic software-component. Very roughly, this description includes:

- the nominal execution time (“0.000137 s”) or a range of times
- a description of the entire context in which the execution-time measurement or analysis has been made
- some indication of the quality of this measurement or estimation

The goal thereby is that the template finds a good compromise between flexibility and precision.
The description must be flexible enough so that the entire range between analytic results (“worst-case execution time”) and rough estimates can be described.
The description should be precise enough so that it is entirely clear what the relevance or meaning of the stated execution time is. This implies that a large amount of context information needs to be provided. The following sections analyze what this context is and provide an appropriate structure for this information.

9.4.3.3 Out of Scope

It is however **not** in the scope of this section to specify *how* the execution time of a runnable entity can be or should be measured or analyzed. We will not discuss what tools or techniques can be used to find the execution time or worst-case execution time of a runnable entity.
It also is **not** in the scope of this section to define *how* information about execution-times is used when integrating various software-components onto one ECU. Similarly this section does not deal with the response-time of the system to certain events. The response-time does not only depend on the execution-times of the runnables of the software-components but also on the infrastructure overhead and on the scheduling policies used.
The focus also is on the description of the execution time of assembly-instructions (typically generated out of compiled C or C++ code). The execution time of e.g. Java byte-code on a virtual machine has not been explicitly considered.

9.4.4 Background

This section provides some background to the proposed solution. Readers who want to skip to the result should go to section 9.4.5.

The execution time can be described for a specific sequence of assembly instructions. It does not make sense to describe the execution time of a runnable provided as source-code unless a precise compiler (and compiler options) are also provided so that a unique set of assembly instructions can be generated out of the source-code.
In addition, the execution-time of such a sequence of assembly instructions depends on:

1. the hardware-platform
2. the hardware state
3. the logical (software) context
4. execution-time of external pieces of code called from the runnable

These dependencies are discussed in detail in the following sections.

9.4.4.1 Dependency of the Execution Time on Hardware

The execution-time depends both on the CPU-hardware and on certain parts of the peripheral hardware:

- The execution time depends on a complete description of the processor, including:
  - kind of processor (e.g. „PPC603“)
  - the internal Processor frequency („100 MHz“)
  - amount of processor cache
  - configuration of CPU (e.g. power-mode)
- Aspects of the periphery that need to be described include:
  - external bus-speed
  - MMU (memory management unit)
  - configuration of the MMU (data-cache, code-cache, write-back,...)
  - external cache
  - memory (kind of RAM, RAM speed)

In addition, when other devices (I/O) are eventually accessed „as memory“ by the I/O hardware abstraction, the speed of those devices potentially has a large influence on the execution time of software components.

On top of this, the ECU might provide several ways to store the code and data that needs to be executed. This might also have a large influence on the execution time. For example:

- execution of assembly instructions stored in RAM versus execution out of ROM might have very different execution times
- when caching is present, the relative physical location of data accessed in memory might also influence the execution time

9.4.4.2 Dependency on Hardware State

In addition to the static configuration of the hardware and location of the code and data on this hardware, the dynamically changing state of the hardware might have a large influence on the execution time of a piece of code : some examples of this hardware state are:

- which parts of the code are available in the execution-cache and what parts will need to be read from external RAM
- what part of the data is stored in data-cache versus must be fetched from RAM
- potentially, the state of the processor pipeline

Although this influence is not relevant on simple or deterministic processors (without cache), the influence of the cache-state on modern processors can be enormous (an order of magnitude difference is not impossible).
Despite the potential importance of this initial hardware-state when caching is present, it is almost impossible and definitely impractical to describe this hardware state. **Therefore it is important and clear that we will not provide explicit attributes for this purpose.**

9.4.4.3 Dependency on Logical Context

This logical context includes:

1. the input parameters with which the runnable is called
2. also the logical "state" of the component to which the runnable belongs (or more precisely: the contents of all the memory that is used by the runnable)

While a description of the input-parameters is relatively straight-forward to specify, it might be very hard to describe the entire logical state that the runnable depends on.
In addition, in certain cases, one wants to provide a specific (e.g. measured or simulated) execution time for a very specific logical context; whereas in other cases, one wants to describe a "worst-case execution time" over all valid logical contexts or over a subset of logical contexts.

9.4.4.4 Dependency on External Code

Things get very complex when the piece of code whose execution time is described makes calls into ("jumps into") external libraries.
To deal with this problem, we could take one of the following approaches:

1. Do not support this case at all: only code that does not rely on external libraries can be given an execution time
2. Support a description of the execution time for a very specific version (again at object-code level) of the libraries. The exact versions of external libraries used would be described together with the execution time. In addition, the relative location in memory of the runnable and the library, the HW-state with respect to the library (e.g. whether this code is in cache or not) and the logical state of the library might have an influence.
3. Conceptually, it might be possible to support a description of the software-component, which explicitly describes the dependency on the execution-times of the library. This description would include:
    a. the execution time of the code provided by the component itself
    b. a specification of which external library-calls are made (with what parameters, how often, in what order,…)

Option 3 is deemed unrealistic and impractical and is not supported.
Option 2 however is important as many software-components might depend on very simple but very common external libraries (like a math-library that provides floating-

point capability in software). “Option 2” will therefore be supported for the case that the external library does not have an additional logical context which influences its execution time.

9.4.5 Description-Model for the Execution Time

9.4.5.1 Inclusion in the Overall Model

Figure 95 shows how the execution time is part of the overall description of the implementation of a component.
The description of the implementation of a component references the description of the internal behavior of the atomic software-component that is implemented.
The description of the internal behavior describes all the "runnable entities" of the component. A runnable entity essentially is a piece of code that can be executed by the run-time environment. (*Theoretically* it should be possible through optimization that jumps between runnable entities of different components on the same ECU are made without going via the run-time environment. This optimization could e.g. be realized by port-based communication to a direct function call. These optimizations are currently not in the scope of this template and therefore are not further described.)
Each description of such a runnable entity (of a specific implementation) can include an arbitrary number of execution-time descriptions. Thereby this execution time description may also depend on code or data variant of the implementation.
It is expected that many runnable entities will not have execution-time descriptions. For runnable-entities that do have execution-time descriptions, the component-implementor could provide several execution-time descriptions: for example one per specific ECU on which the implementation can run and on which the time was measured or estimated.
If a RunnableEntity is defined to be running in an ExclusiveArea (see section 7.9.2.1 RunnableEntityRunsInExclusiveArea) the execution time of the whole RunnableEntity can be considered to allow the RTE configuration an optimization of the data consistency mechanis.
If a RunnableEntity is defined to be able to enter an ExclusiveArea (see section 7.9.2.2 RunnableEntityCanEnterExclusiveArea) the execution time can be specified for each section. The time provided is the time consumed AFTER the RTE call to enter the ExclusiveArea and BEFORE the RTE call the leave the ExclusiveArea.

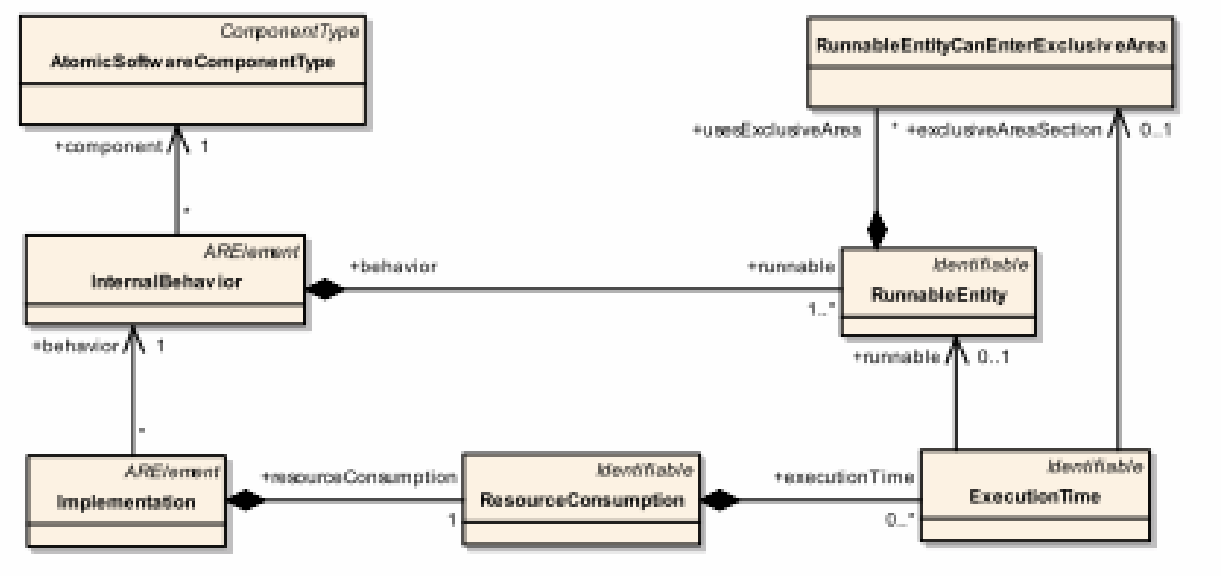

Figure 95: Position of execution-time description in the overall model

9.4.5.2 Detailed Structure of an Execution-Time Description

Figure 96 shows the details of an execution time description. The following paragraphs describe aspects of this model in more detail.

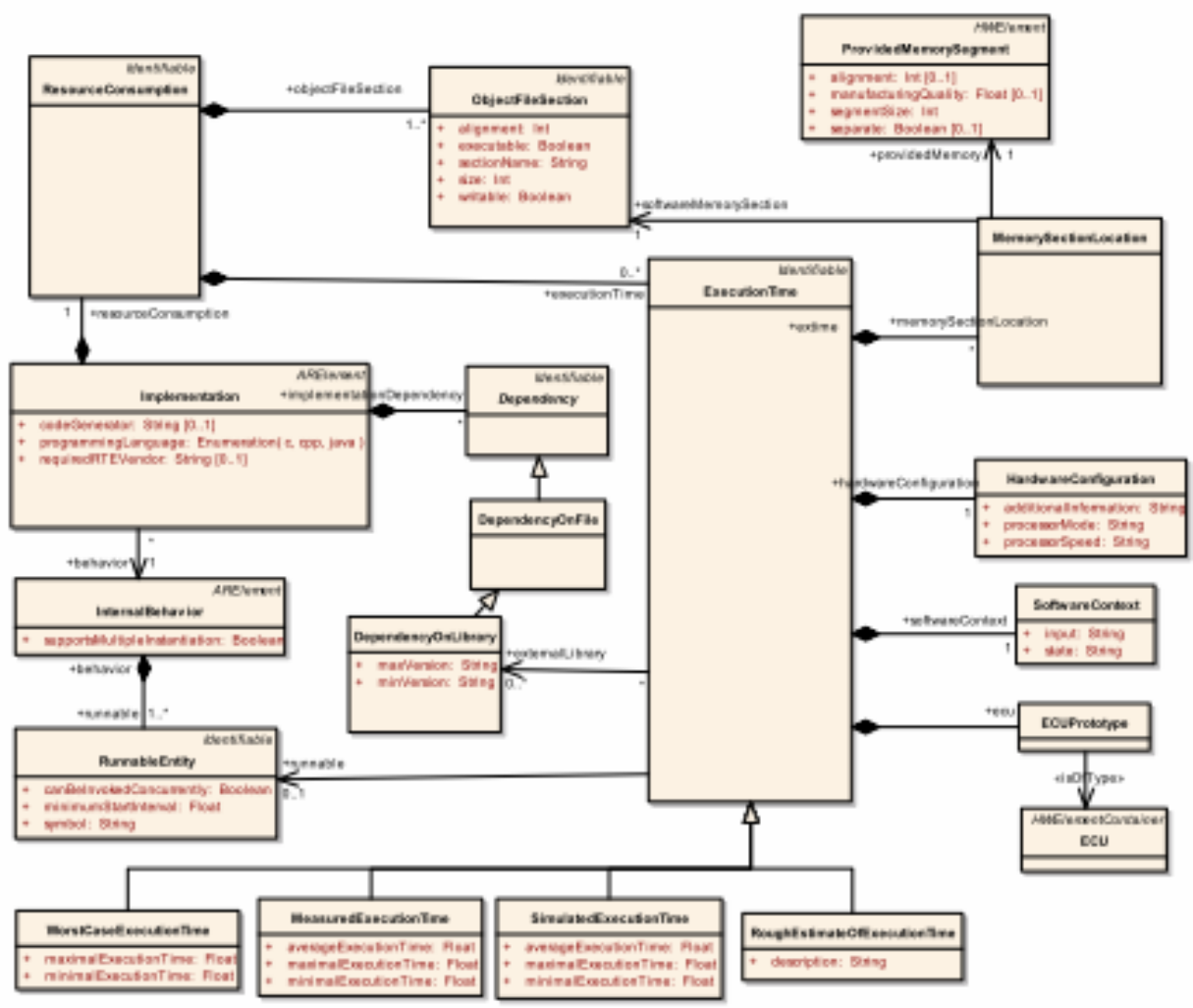

Figure 96: Detailed structure of an Execution-Time Description

The following shows the attributes of the “ExecutionTime” in tabular form:

Class	ExecutionTime			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::Execution-Time			
Class Description				
Base Class(es)	Identifiable			
Attribute	Datatype	Mul.	Link Type	Attribute Description
ecu	ECUPrototype	1	aggregation	
exclusiveAreaSection	RunnableEntity-CanEnterExclusiveArea	0..1	reference	Reference to the RunnbaleEntityCanEnterExclusiveArea this execution time is provided for.
externalLibrary	Dependency-OnLibrary	0..*	reference	If this dependency is specified, the execution time of the library code is included in the execution time data for the runnable.
hardwareConfiguration	Hardware-Configuration	1	aggregation	
memorySectionLocation	MemorySection-	0..*	aggregation	

	Location			
runnable	RunnableEntity	0..1	reference	Reference to the runnable this execution time is provided for.
softwareContext	SoftwareContext	1	aggregation	

9.4.5.3 ExecutionTime References an “ECU”

The ExecutionTime references an "ECU" (the concept “ECU” is defined by the ECU-Resource-Template). This ECU-reference uniquely describes the hardware for which the execution-time is provided. This includes: the kind of processor, the type of MMU, the type of caches, type of memory available,…
Note that this reference to an ECU has a different semantic than the attribute “processor” in the Implementation. The “processor” defines the family of processors on which the provided implementation may run (it is a requirement on the hardware on which the component may be deployed). The “ECU” on the other hand (of which the processor only is one part) is a statement on the context of the execution-time.
Of course, the processor of the ECU should be equal to the processor specified in the “Implementation”.
Note that the ECU might include specific hardware that has no influence on the execution-time. Despite this, it seems currently better to specify a reference to the entire hardware-platform used rather than introduce another hardware sub-system that includes all hardware-elements that influence the execution-time of software.

9.4.5.4 ExecutionTime Includes a HW-configuration

The ECU described through the “ecu” attribute can still run in several HW-modes.
For example, many ECUs can run in several “speed”-modes (for example a normal fast-mode and a low-power slow mode). The goal of the HWConfiguration is to describe this.
The attributes “processorSpeed” and “processorMode” should describe the specific mode of the ECU.
Because of the potential dependency on many other hw-configuration settings (such as caching policy, MMU-settings,…), a generic attribute “additionalInformation” is provided.
Because the exact structure of the information seems to depend so much on the specific case, all attributes are unstructured text.

Class	HardwareConfiguration			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::ExecutionTime			
Class Description				
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
additionalInformation	String	1	aggregation	
processorMode	String	1	aggregation	
processorSpeed	String	1	aggregation	

9.4.5.5 ExecutionTime Includes a MemorySectionLocation

For each memorySection of the Implementation (including the per-instance memory sections of the instance for which the execution time is described), the Execution-Time must specify where this section was located on the physical memory of the ECU.
The memory-sections on the software-side are described in the StaticMemoryUsage-description of the implementation. The available memory-regions on the hardware are described inside the description of the ECU.
The ExecutionTime description contains descriptions of the location of the memory sections (MemorySectionLocation) which link a software memory section to a hard-ware memory section on the ECU.

Class	MemorySectionLocation			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::Execution-Time			
Class Description				
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
providedMemory	ProvidedMemorySegment	1	reference	
softwareMemorySection	ObjectFileSection	1	reference	

9.4.5.6 ExecutionTime Includes a SoftwareContext

The SoftwareContext is the logical context for which the execution time is given. This includes two aspects:

a. the values of the input-parameters to the runnable entity
b. the state the logic of the runnable depends on

In the current form, both attributes are of type “String” and can contain free-form text describing this state.
For the attribute “input”, it might be appropriate to refine this into a more formal de-scription of the values of the parameters.
For the attribute “state”, it is difficult to go beyond an informal text-field, because the state is a private matter of the component and there currently is no explicit mecha-nism in AUTOSAR to describe the value of this state.
Further, it is possible to provide several execution times of a runnable entity, for ex-ample, in case of different values of the input-parameters. This is one of the reasons why the template supports an arbitrary number of execution times per runnables.

Class	SoftwareContext			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::ExecutionTime			
Class Description				
Base Class(es)	Identifiable, PerInstanceMemory, NVRAMMapping, ARObject			
Attribute	Datatype	Mul.	Link Type	Attribute Description
input	String	1	aggregation	
state	String	1	aggregation	

9.4.5.7 Dependency on External “Libraries”

The execution time measurements can depend on the precise version of external libraries (such as a math-emulation library) that have been used.
This information can be included by adding a reference to an object of type DependencyOnLibrary which must be aggregated by the corresponding Implementaion. If such a reference is specified, the execution time includes the execution time of that specific library version.
In case the Implementation aggregates attributes of type DependencyOnLibrary, to which the ExecutionTime does not refer, it means that the executuion time of the libray code is NOT included in the execution time of the runnable.

9.4.5.8 Several Qualities of Execution Times

9.4.5.8.1 WorstCaseExecutionTime

The “worst case execution time” is used to build the application schedule. It is an overall approximation of a runnable entity which will be running on a hardware ECU context.
Further "worst-case" means that an "analytic" method was used to find the worst-case (=guaranteed) boundaries. But this boundary has a lower-limit and an upper-limit.
Considering the cache processor ECU, an execution time could be computed, and it depends on cache level. A maximalExecutionTime and a minimalExecutionTime as to be filled.

Class	WorstCaseExecutionTime			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::ExecutionTime			
Class Description				
Base Class(es)	Identifiable, ExecutionTime			
Attribute	Datatype	Mul.	Link Type	Attribute Description
maximalExecutionTime	Float	1	aggregation	
minimalExecutionTime	Float	1	aggregation	

9.4.5.8.2 MeasuredExecutionTime

The measured execution time describes the runnable entity runtime on ECU.

Class	MeasuredExecutionTime			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::ExecutionTime			
Class Description				
Base Class(es)	Identifiable, ExecutionTime			
Attribute	Datatype	Mul.	Link Type	Attribute Description

averageExecutionTime	Float	1	aggregation	
maximalExecutionTime	Float	1	aggregation	
minimalExecutionTime	Float	1	aggregation	

9.4.5.8.3 SimulatedExecutionTime

A simulated execution time describes the time information which are coming from a simulation. Simulation could be based on:

- Runnable entity modelisation on specific hardware with time weighting to simulate processor time behaviour
- Runnable entity Model before generation code

Class	SimulatedExecutionTime			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::Execution-Time			
Class Description				
Base Class(es)	Identifiable, ExecutionTime			
Attribute	Datatype	Mul.	Link Type	Attribute Description
averageExecutionTime	Float	1	aggregation	
maximalExecutionTime	Float	1	aggregation	
minimalExecutionTime	Float	1	aggregation	

9.4.5.8.4 RoughEstimateOfExecutionTime

Class	RoughEstimateOfExecutionTime			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Resources::ExecutionTime			
Class Description				
Base Class(es)	Identifiable, ExecutionTime			
Attribute	Datatype	Mul.	Link Type	Attribute Description
description	String	1	aggregation	

10 HW, ECU Abstraction and AUTOSAR SW-components

10.1 Overview

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. The SW-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 this document (see chapter 9). In this chapter the overall system view of the interface between sensors/actuators and software is described and the consequences for the SW-Component template are derived.

10.2 High Level Hardware and Software Architecture

AUTOSAR defines a software architecture (see Figure 97) and within this layered architecture the interfaces between the hardware and the software are explicitly modeled.

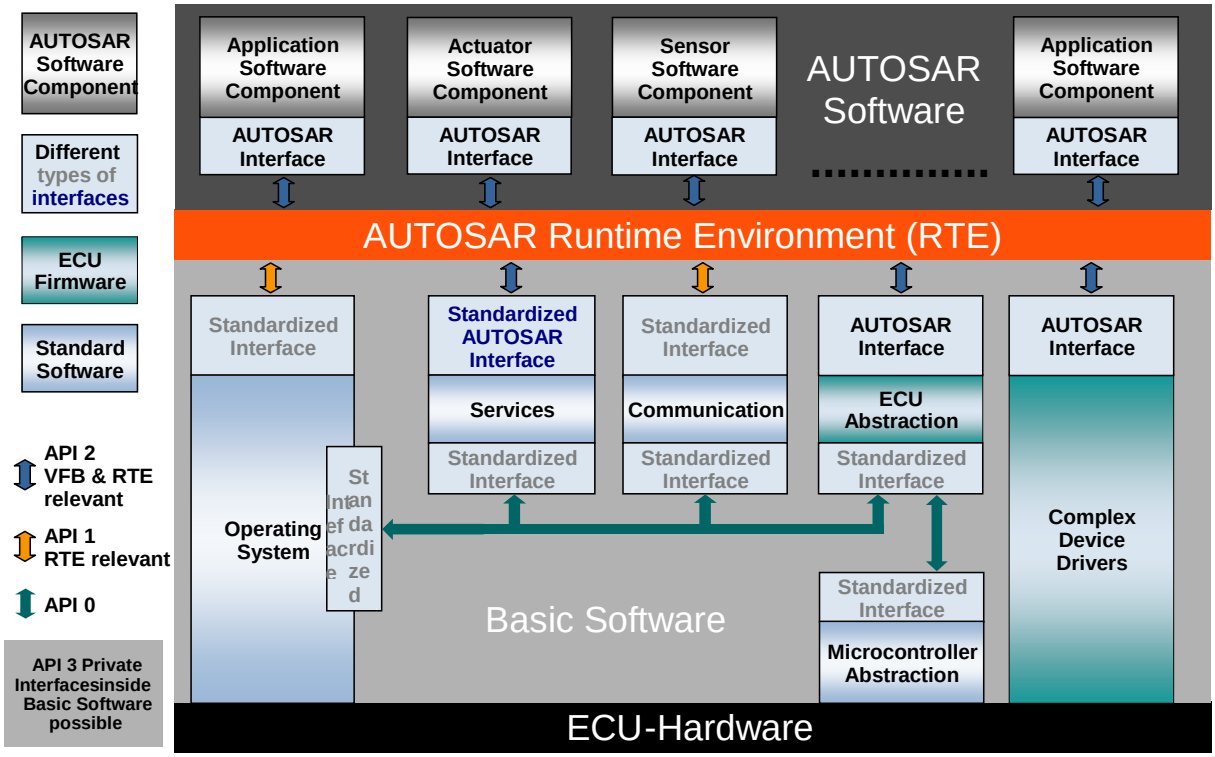

Figure 97: AUTOSAR ECU SW Architecture

The signal[30] flow from a hardware to software and vice versa will be described in the following sections.
A sensor[31] is converting a physical value (1) in Figure 98 (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 shown in the top part of Figure 98.

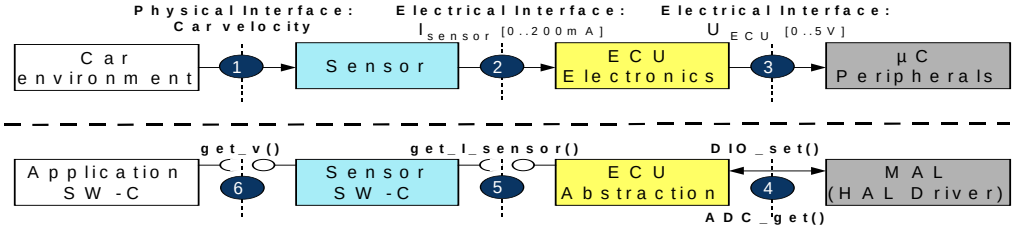

Figure 98: Interfaces between hardware and software

This signal chain is represented one to one in the AUTOSAR software architecture and shown in the lower part of Figure 98.
In an implementation of AUTOSAR only the *Microcontroller Abstraction* 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 a current is the input to the microcontroller peripheral, the MAL will deliver a data value that represents this current.
As the ECU Electronics has enhanced and converted the electrical signal before 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 a current and the ECU Electronics has converted this current into a voltage (2→3), the ECU Abstraction will convert the data value voltage into an AUTOSAR signal representing a current (4→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

[30] 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.
[31] For the simplicity of the explanation a sensor is discussed here, but of course the analog is applicable for an actuator.

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 (5→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 Sensor Software Component due to potentially high communication effort.
In Figure 99 a complete signal flow from a sensor input to an actuator output is shown.

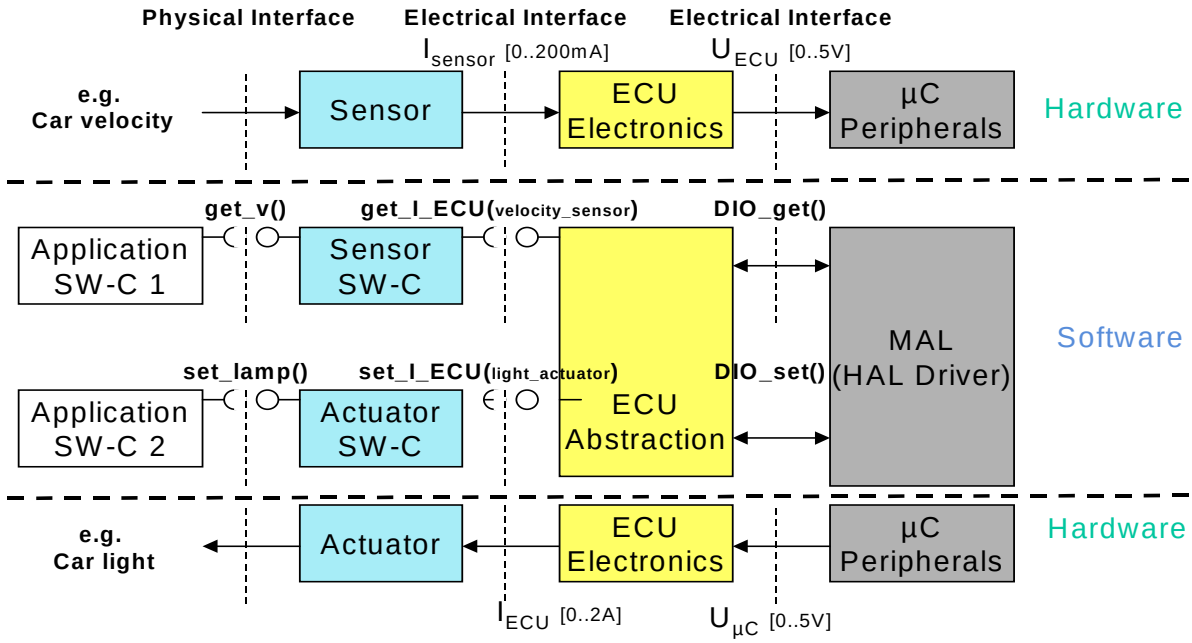

Figure 99: Sensor and Actuator Signal Flow

In the next section the interfaces between the involved software modules are discussed.

10.3 Interfaces and APIs

Two fundamentally different interfaces are involved when converting from sensors/actuators to software components: API 0 and API 2 (see Figure 97).

10.3.1 API 0 Interface

The interface between the Microcontroller Abstraction and the ECU Abstraction is a Standardized Interface (see AUTOSAR Glossary). This interface is not visible on the

Virtual Function Bus and therefore the MAL 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.

10.3.2 API 2 Interfaces

While the ECU Abstraction has an API 0 interface to the MAL, the interface to the sensor/actuator software components is API 2. This means that this interface is visible on the virtual function bus. So the ECU Abstraction and the sensor/actuator software components do not need to be present on the same ECU but can be separated. In general the sensor/actuator software component should be on the same ECU as the ECU hardware abstraction.
Also the interface between the sensor/actuator software component and the actual application software components is an API 2 interface.
To describe the data that is going to be exchanged at this API 2 interface the standard AUTOSAR Interface description mechanisms are used (see chapter 4).

10.3.3 ECU Abstraction and its AUTOSAR Interfaces

Since AUTOSAR is designed with the focus on the integration of software components coming from different contractors, the interfaces between the different components 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 Port that represents the AUTOSAR Signal that is delivered by the sensor or the AUTOSAR Signal that is consumed by the actuator. Because the ports are implementing an AUTOSAR Interface, in the following figures these interfaces are shown as the main distinguishing criteria.

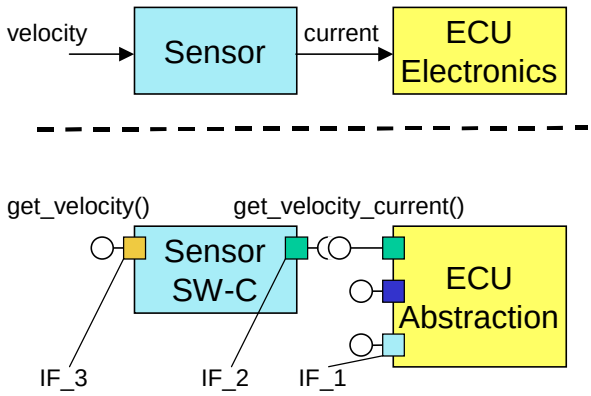

Figure 100: Interfaces of the signals in software

Each sensor and actuator has an AUTOSAR Port at the ECU Abstraction. Connected to this port is the Sensor/Actuator SW-Component. The Sensor SW-Component has one Port to the ECU Abstraction (*IF_2*) where it gets the AUTOSAR signals from the

hardware, and one Port to AUTOSAR SW-Components (*IF_3*) where it provides the actual physical value to the rest of AUTOSAR on the RTE.
The Interfaces between the ECU Abstraction and the Sensor SW-Component have to be compatible like defined in the Compatibility of Interfaces section (see chapter 4.6.5.3).

10.4 Shipment of Sensors/Actuators

In the described layered architecture each hardware sensor/actuator is coupled to a sensor/actuator SW-Component. Since the SW-Component template is going to be used to describe the sensor/actuator SW-Components 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.

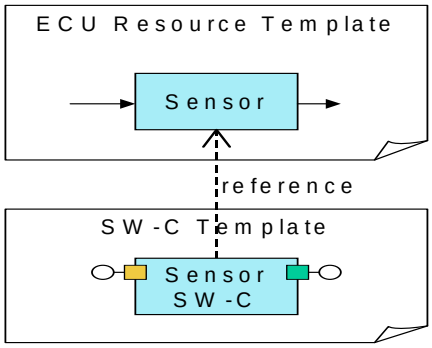

Figure 101: Shipment of a Sensor

So each time a sensor/actuator is selected to be connected to an ECU also the corresponding sensor/actuator SW-Component is available.

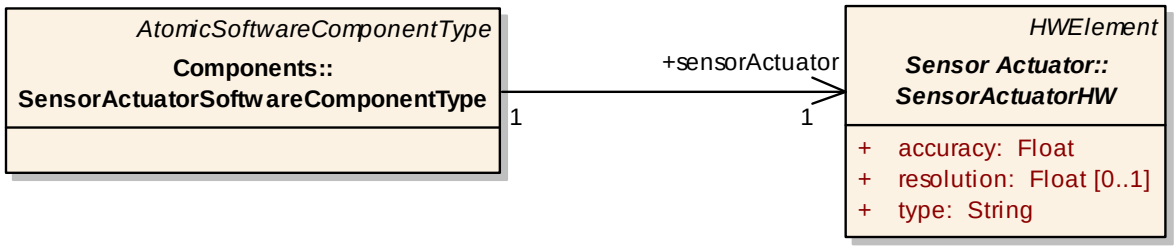

Figure 102: Sensor/Actuator to hardware relationship

In Figure 102 the reference in the meta-model is shown. The SensorActuatorSoftwareComponentType is 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. And in contrast to an AtomicSoftwareComponentType, an SensorActuatorSoftwareComponentType may use the I/O hardware abstraction directly.

Class	SensorActuatorSoftwareComponentType			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Class Description	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)	Identifiable, PackageableElement, NonSplitableElement, ARElement, ComponentType, AtomicSoftwareComponentType			
Attribute	Datatype	Mul.	Link Type	Attribute Description
sensorActuator	SensorActuatorHW	1	reference	Reference from the Sensor Actuator Software Component Type to the description of the actual hardware.

11 AUTOSAR Services

11.1 Overview

This chapter covers the description and handling of AUTOSAR service configuration. Due to the similarity to AUTOSAR services, the information given in this chapter also applies to the I/O Hardware Abstraction Layer configuration.
AUTOSAR Services can be seen as a hybrid between basic software modules providing access to low level ECU wide services and standardized AUTOSAR components: “AtomicSoftwareComponents” requiring services use standardized AUTOSAR interfaces to communicate with these 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. The following list of paragraphs presents a short overview over the steps required for the configuration of 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 not to be entered manually within authoring tools.

1. The dependency of an “AtomicSoftwareComponent” from an AUTOSAR service is modeled by aggretating 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. Additionally, the “PortPrototype” may 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 toplevel “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 *“Specification of System Template”*, this contains an ECU centric view onto the system description, including a reduced version of the System’s software composition where SoftwareComponentPrototypes 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 application SW-Components: An adequate number of “PortPrototypes” are created on this “ServiceComponentType” for each needed Port at the application SW-Components.
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 Atomic

Software Components 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, “Runnable Entities” 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.
8. An “Implementation“ is being created for each “InternalBehavior” created in the previous step. For the RTE BSW module configuration an “ImplementationSelection” is set up in order to select the “Implementation” belonging to each “ServiceComponentPrototype”, thus assigning the “InternalBehavior”.
9. In ECU Configuration phase the remaining service parameters are specified.

11.2 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 Software Component Types, 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 Software Composition Hierarchies. Authoring tools however shall forbid the users to manually create instances of these meta-model classes in Software Component descriptions.

11.2.1 EcuSwComposition

As explained in 11.1, 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 103. “EcuSwComposition” represents the whole Software Composition on an ECU, including both the application 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. Special connectors of type “ServiceConnecorPrototype” are used for connecting service-requiring “PortPrototype” instances of application components with the actual service “PortPrototpye” instances defined in the ServiceComponentType.

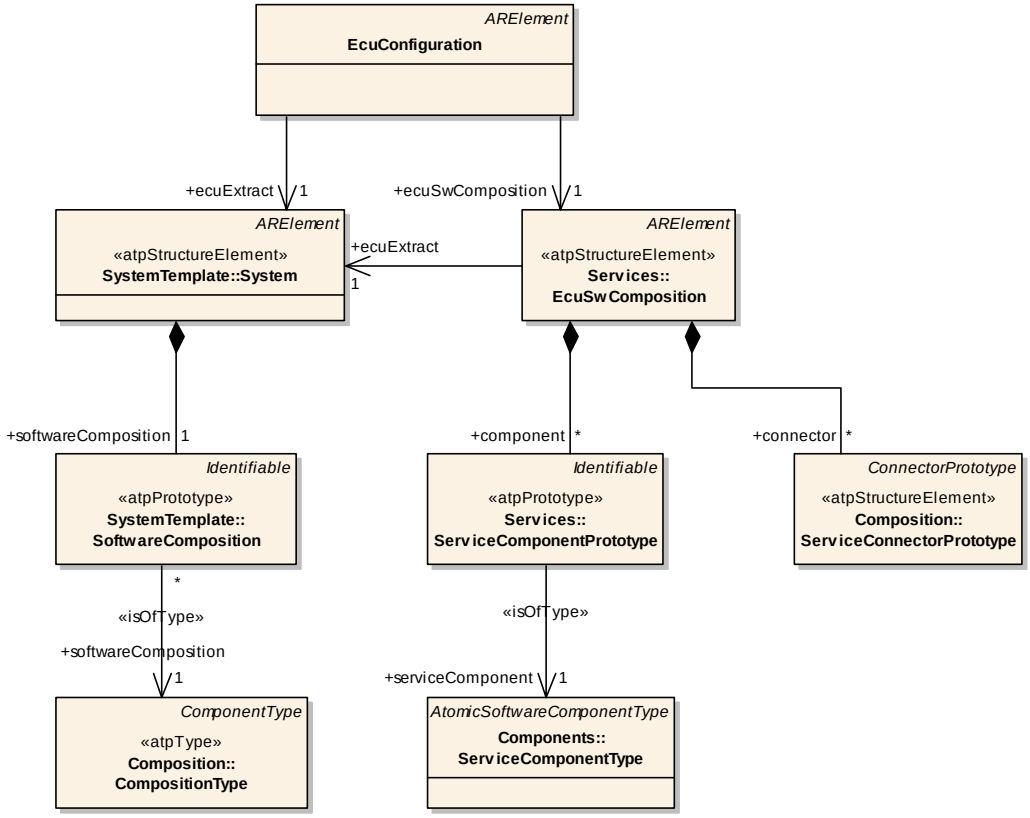

Figure 103: EcuSwComposition

11.2.2 ServiceComponentType

AUTOSAR Services are represented by a meta model class of their own, the “ServiceComponentType”. As can be seen in Figure 104 “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.

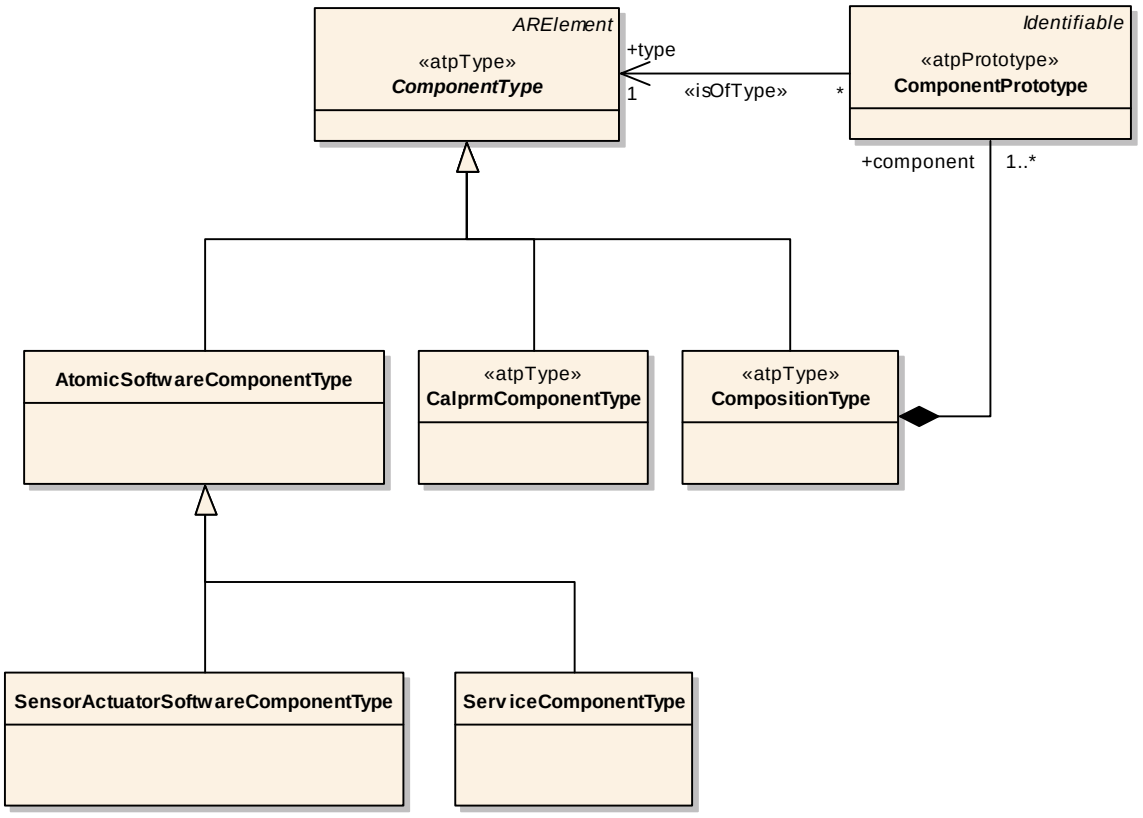

Figure 104: ServiceComponentType

"ServiceComponentType" must not be used when modeling normal application software compositions 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 each needed Service Port in the Software Composition exactly one port on the "ServiceComponentType" is created. More explicitly, all instances of "AtomicSoftwareComponentType" need to be checked for PortPrototypes of «isService» PortInterfaces, and for each of these PortInterface instances belonging to the service to be configured one PortPrototype implementing the same PortInterface needs to be created on the "ServiceComponentType". The roles of the PortPrototypes (required/provided) on the Application Component and the Service Component side are reversed accordingly.

Class	ServiceComponentType
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components
Class Description	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)	Identifiable, PackageableElement, NonSplitableElement, ARElement, ComponentType, AtomicSoftwareComponentType

11.2.3 ServiceConnectorPrototype

The “ServiceConnectorPrototype” 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”.

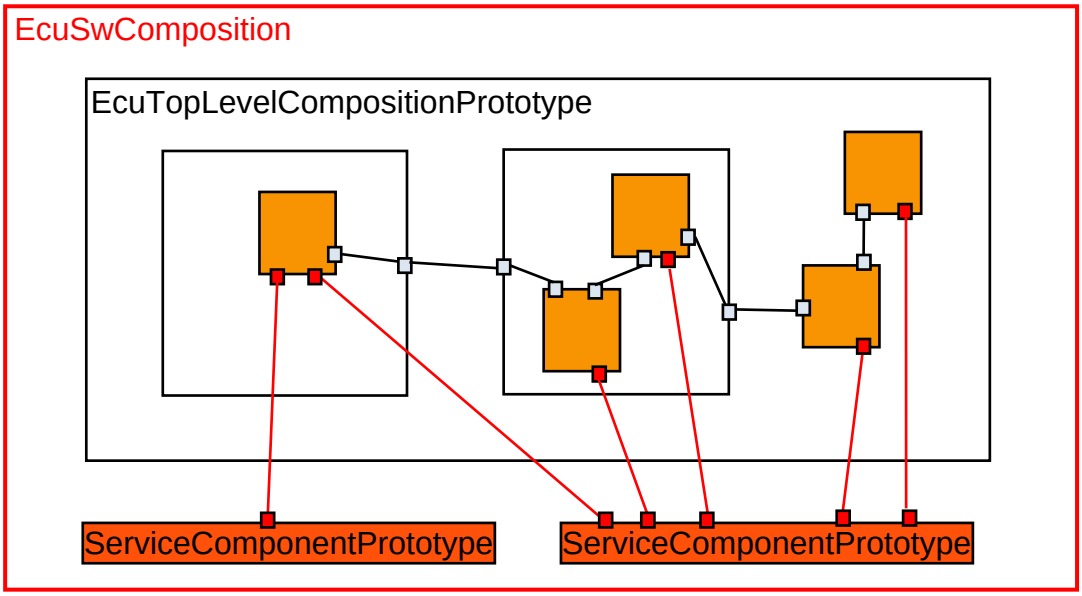

Figure 105: ServiceConnectorPrototypes connecting Application Component Service Ports to ServiceComponentPrototype Service Ports

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.

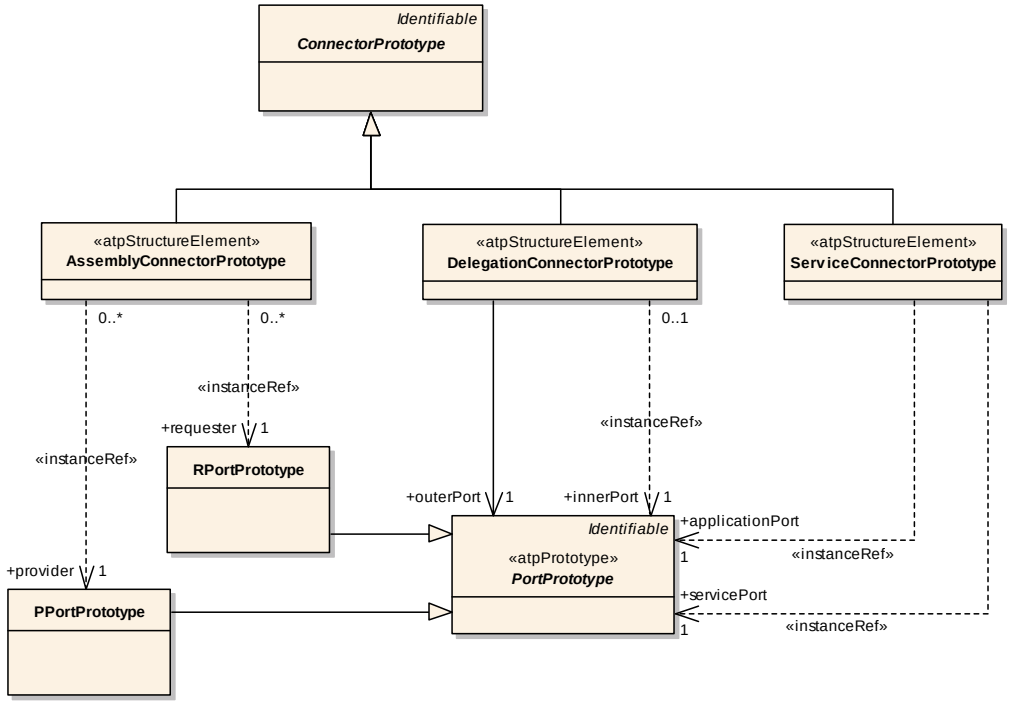

Figure 106: ServiceConnectorPrototype

Class	ServiceConnectorPrototype			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Composition			
Class Description	A ServiceConnectorPrototype connects an application component required-service port with the service port provided by the service component. They are only added to the model in ECU Configuration phase for the specific purpose of configuring services within an EcuSwComposition.			
Base Class(es)	Identifiable, ConnectorPrototype			
Attribute	Datatype	Mul.	Link Type	Attribute Description
applicationPort	ServiceConnectorPrototype_-applicationPort	1	reference to instance	Service port to be connected on application component side
servicePort	ServiceConnectorPrototype_service-Port	1	reference to instance	Service port to be connected on service component side

12 Appendix

12.1 References

12.1.1 Normative References

The relevant documents that need to be considered to understand the content of this document are:

[1] Requirements on Software Component Teemplate,
AUTOSAR_RS_SoftwareComponentTemplate.pdf

[2] Template UML Profile and Modeling Guide,
AUTOSAR_TemplateModelingGuide.pdf

[3] Model Persistence Rule for XML,
AUTOSAR_ModelPersistenceRulesforXML.pdf

12.1.2 Informative References

Additionally the following documents should be read:

[4] Glossary,
AUTOSAR_Glossary.pdf

[5] Specification of the Virtual Functional Bus,
AUTOSAR_Spec_of_VFB.pdf.

[6] ASAM Specifications. General
http://www.asam.net

[7] ASAM Specifications. Meta Data Exchange Format for Software Module Sharing (MDX
http://www.asam.net

[8] ASAM Specifications. Container Catalog XML Model Specification.
http://www.asam.net

[9] ASAM Specifications. Calibration Data Format (CDF).
http://www.asam.net

[10] ASAM Specifications. Harmonized Data Objects.
http://www.asam.net

[11] ASAM Specifications. ASAP 2 Interface Specifiation.
http://www.asam.net

[12] MSR-TR-CAP. http://www.msr-wg.de/medoc/download/literature/msr-tr-medoc-en/msr-tr-medoc-en.pdf

[13] Specification of ECU Configuration Parameters
AUTOSAR_ECU_ConfigurationParameters.pdf