Document Title	Modeling Guidelines of Basic Software EA UML Model
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	117

Document Status	Final
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	4.4.0

Document Change History

Date	Release	Changed by	Description
2018-10-31	4.4.0	AUTOSAR Release Management	• minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2017-12-08	4.3.1	AUTOSAR Release Management	• minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2018-04-17	4.4.0	AUTOSAR Technical Office	• Removed obsolete elements.
2016-11-30	4.3.0	AUTOSAR Release Management	• Editorial changes
2014-10-31	4.2.1	AUTOSAR Administration	• Editorial changes
2013-03-15	4.1.1	AUTOSAR Administration	• Finalized for Release 4.1
2010-02-02	3.1.4	AUTOSAR Administration	• Modeling of header files has been revised • Description of parameter modeling has been reworked • Legal disclaimer revised
2008-08-13	3.1.1	AUTOSAR Administration	• Legal disclaimer revised

2007-12-21	3.0.1	AUTOSAR Administration	• Added description for range stereotype • Change Requirements for function parameter and structure attributes • Document meta information extended • Small layout adaptations made
2006-11-28	2.1.1	AUTOSAR Administration	• Usage of packages clarified • Sequence diagram modeling clarified • Legal disclaimer revised
2006-05-16	2.0	AUTOSAR Administration	• Initial release

Disclaimer

Table of Contents

References

[1] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture

[2] List of Basic Software Modules
AUTOSAR_TR_BSWModuleList

[3] Glossary
AUTOSAR_TR_Glossary

[4] Specification of Standard Types
AUTOSAR_SWS_StandardTypes

[5] Generic Structure Template
AUTOSAR_TPS_GenericStructureTemplate

[6] Standardized M1 Models used for the Definition of AUTOSAR
AUTOSAR_MOD_GeneralDefinitions

1 Introduction

This modeling guide describes the applied modeling techniques and rules, used to specify the AUTOSAR Basic Software within a UML model.

The information contained in the BSW model is processed by the AUTOSAR Meta Model Tool (MMT) and provides a major input of the several Software Specifications (SWS) defined by AUTOSAR. In order to make the BSW model accessible by the MMT, it is essential that the model observes the rules described in this document.

1.1 Artifacts

The main purpose of the AUTOSAR BSW UML model is keeping the 99+ documents synchronous with respect to file structure, provided and required interfaces, sequence diagrams, state machines etc. Therefore, all the relevant information is kept in the BSW model according to the modeling rules specified in chapter 2, Modeling Guide.

The following artifacts are contributed to the SWS documents by the BSW UML model:

1.1.1 Header Files

Chapter 5.1 of each SWS document contains the BSW module's file structure, in particular its file inclusion structure. Most modules' include file relationships have a similar structure, in fact some parts are actually identically modeled. Therefore, the Header File structure is being modeled using a class diagram, with stereotyped classes representing the source code- and header files; see section 2.4.1.

1.1.2 Imported Type Definitions

SWS chapter 8.1 contains a tabular list of imported types. This table is automatically generated from the module dependency as explained in section 2.3.5.

1.1.3 Type Definitions

SWS chapter 8.2 contains detailed descriptions of all types defined within a given BSW module. For details on the modeling of type definitions refer to section 2.3.8.

1.1.4 Function Definitions

SWS chapter 8.3 contains a detailed description for each function provided by the BSW module. The description is presented in form of a table with a specific layout.

The individual fields of the table are filled from the API function definitions according to section 2.3.3.

1.1.5 Callback Notifications

Very similar to the Function Definitions, SWS chapter 8.4 contains the callback definitions the BSW module provides. These are callbacks which will be called by other BSW modules, where the lower layer module is typically the caller. A table for each callback notification will be generated for a module's specified callbacks according to section 2.3.7.

1.1.6 Scheduled Functions

Scheduled Functions are described in SWS chapter 8.5. The definition of scheduled functions in the BSW UML model is described in section 2.3.3.1.

1.1.7 Mandatory Interfaces

SWS chapter 8.6.1 contains a list of "mandatory interfaces" expected by the module. The list is generated from the BSW UML model according to the mandatory dependencies as described in section 2.3.5.2.

1.1.8 Optional Interfaces

Similarly, the list of "optional interfaces" contained in SWS chapter 8.6.2 is generated from the BSW UML model according to the optional dependencies as described in section 2.3.5.3.

1.1.9 Configurable Interfaces

SWS Chapter 8.6.3 contains a BSW module's "Configurable Interfaces". These are interfaces whose called function name can be configured using ECU configuration parameters. In AUTOSAR, these are typically used for issuing callback notifications, i.e. the module owning the configurable interface uses it to notify a (configurable) upper layer module's callback. In other words the module defining a "Configurable Interface" calls an other module that implements these interface definition. A table for each callback notification will be generated for a module's specified callbacks according to section 2.3.7.2.

1.1.10 Sequence Diagrams

In order to visualize the interaction of a BSW module with other modules, SWS Chapter 9 contains UML Sequence Diagrams for the module's typical use cases. In order to keep such Sequence Diagrams consistent between different modules within the AUTOSAR BSW stack, they are also modeled within the BSW UML model. The diagrams are being exported to image files by the mmt tool; they are then being included by the SWS document files. For the detailed modeling guidelines see section 2.4.2

1.1.11 Various Diagrams

The SWS documents of various BSW modules use additional UML diagrams e.g. for either specifying core functionality, or for additionally illustrating dependencies between modules. Some concrete examples are the various state machines used throughout the AUTOSAR BSW stack, for example in the CAN State Manager or in COM manager. Whenever possible, such diagrams should also be modeled in the BSW UML model. This ensures that the sources of the document diagrams will not get lost, and also facilitates their maintenance and keeping a uniform modeling style.

1.1.12 Modeling of services

BSW Modules belonging to the Service Layer of the AUTOSAR Basic Software Architecture may offer their services in the form of AUTOSAR Service Interfaces. AUTOSAR Service Interfaces are described in terms of the Software Component Template rather than C-language interfaces, and they come in different flavors, e.g. ClientServerInterface, SenderReceiverInterface, ModeSwitchInterface. Consequently, their properties require a different style of modeling than the standard BSW API functions. Modeling of AUTOSAR services is described in section 2.3.9

2 Modeling Guide

This Chapter contains the modeling rules that shall be followed when modeling AUTOSAR BSW artifacts within the BSW UML model. It is important that these rules are used consistently throught the model for the following reasons: The model stays readable, additions and modifications are done in a reproducible way preventing the duplication of elements, and most importantly, the automated artifact generation using the MMT tool depends on nonambiguous modeling conventions.

2.1 Terminology

The agreed tool for UML modeling in AUTOSAR is *Enterprise Architect* by Sparx Systems. Accordingly the BSW model is being maintained using Enterprise Architect version 7.5 and above. This guide focusses on modeling techniques rather than tools, therefore this document strives to describe the concepts in terms of UML. Nevertheless, in order to be precise, sometimes terms specific to Enterprise Architect are used.

2.2 Model Structure

The root structure of the BSW UML model consists of the following packages:

ReadMe: Contains diagrams providing version number, known limitations and disclaimer.

Interaction Views: Contains sequence charts for modeling interactions of different modules. Only sequence diagrams shall be placed into this packages. The modules are arranged by stack vertically.

SoftwarePackages: Contains the BSW modules definitions including interfaces and type definitions. Moreover state and header diagrams are modelled here. The modules are arranged by layer horizontally.

Generic Elements: Contains common interface definitions e.g. configurable callback definitions.

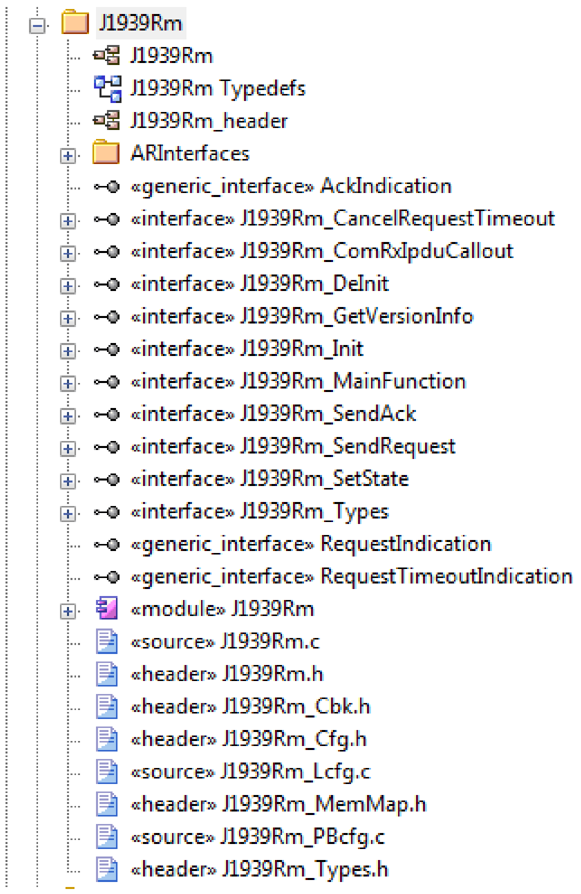
2.3 Modeling of BSW Modules

2.3.1 Modules

2.3.1.1 Packages

[TR_BSWMG_00001] BSW Module Packages ⌈ For each basic software module a UML package (the "module package") shall be placed within the package structure according to the module's role in the Layered Software Architecture[1]. ⌋*()*

[TR_BSWMG_00002] Naming of BSW Module Packages ⌈ The name of the *module package* shall be the 'module abbreviation' as specified in the List of Basic Software Modules[2]. ⌋*()*



```
J1939Rm
    J1939Rm
    J1939Rm Typedefs
    J1939Rm_header
    ARInterfaces
    «generic_interface» AckIndication
    «interface» J1939Rm_CancelRequestTimeout
    «interface» J1939Rm_ComRxIpduCallout
    «interface» J1939Rm_DeInit
    «interface» J1939Rm_GetVersionInfo
    «interface» J1939Rm_Init
    «interface» J1939Rm_MainFunction
    «interface» J1939Rm_SendAck
    «interface» J1939Rm_SendRequest
    «interface» J1939Rm_SetState
    «interface» J1939Rm_Types
    «generic_interface» RequestIndication
    «generic_interface» RequestTimeoutIndication
    «module» J1939Rm
    «source» J1939Rm.c
    «header» J1939Rm.h
    «header» J1939Rm_Cbk.h
    «header» J1939Rm_Cfg.h
    «source» J1939Rm_Lcfg.c
    «header» J1939Rm_MemMap.h
    «source» J1939Rm_PBcfg.c
    «header» J1939Rm_Types.h
```

Figure 2.1: Example of a module package

2.3.1.2 Components

[TR_BSWMG_00003] BSW Module Components ⌈Each basic software module shall be modeled as an UML component with stereotype «module» (the "module component"). ⌋*()*

[TR_BSWMG_00004] Naming of BSW Module components ⌈ The name of the *module component* shall be the 'module abbreviation'[2]. ⌋*()*

[TR_BSWMG_00036] BSW Module ID ⌈ The tagged value "bsw.moduleId" shall be set to the module ID as specified in the List of Basic Software Modules[2]. ⌋*()*

[TR_BSWMG_00005] Location of BSW Module components ⌈ Each *module component* shall be modeled as a top-level element of its containing module package. ⌋*()*

[TR_BSWMG_00094] SWS Item ID of the Mandatory Interfaces table of the module ⌈ The tagged value "bsw.mandatory.swsItemId" is used to specify the SWS Item ID of a API function. ⌋*()*

[TR_BSWMG_00095] Up-traces of the Mandatory Interfaces table of the module ⌈ The tagged value "bsw.mandatory.traceRefs" is used to specify up-traces to requirements. Multiple requirement IDs have to be separated by a comma. ⌋*()*

[TR_BSWMG_00096] SWS Item ID of the Optional Interfaces table of the module ⌈ The tagged value "bsw.optional.swsItemId" is used to specify the SWS Item ID of a API function. ⌋*()*

[TR_BSWMG_00097] Up-traces of the Optional Interfaces table of the module ⌈ The tagged value "bsw.optional.traceRefs" is used to specify up-traces to requirements. Multiple requirement IDs have to be separated by a comma. ⌋*()*

[TR_BSWMG_00098] SWS Item ID of the Imported Types table of the module ⌈ The tagged value "bsw.importedTypes.swsItemId" is used to specify the SWS Item ID of a API function. ⌋*()*

[TR_BSWMG_00099] Up-traces of the Imported Types table of the module ⌈ The tagged value "bsw.importedTypes.traceRefs" is used to specify up-traces to requirements. Multiple requirement IDs have to be separated by a comma. ⌋*()*

2.3.1.3 Component Diagrams

[TR_BSWMG_00006] Component Diagrams ⌈The module package shall contain a "component diagram" (Enterprise Architect: UML Component Diagram). ⌋*()*

[TR_BSWMG_00007] Naming of Component Diagrams ⌈ The name of the component diagram shall be identical to the name of the *module component* (module abbreviation). ⌋*()*

[TR_BSWMG_00008] Content of Component Diagrams ⌈ The component diagram contains the module component as well as all of the module's interface relationships. ⌋*()*

2.3.1.4 Type Diagrams

[TR_BSWMG_00009] Type Diagram ⌈ If a BSW module defines data types, its module package shall contain a "types diagram" (Enterprise Architect: UML Class Diagram). ⌋ *()*

[TR_BSWMG_00010] Naming of Types Diagram ⌈ The name of the types diagram shall be the name of the *module component* followed by a space character followed by `Types`, e.g. `FrTp Types`. ⌋*()*

[TR_BSWMG_00011] Content of Types Diagram ⌈ The *types diagram* shall contain all types defined by the BSW module. ⌋*()*

2.3.2 Function interfaces

An AUTOSAR BSW modules provides services to other BSW modules in the form of C-syntax functions. These functions are also the underlying implementation of AUTOSAR Services accessed by Software Components over the RTE.

This section explains how each such function is modeled in the form of an UML operation. Each operation is placed in an UML interface owned by the BSW module realizing the service. This UML interface is hereinafter called Function interface.

[TR_BSWMG_00012] Function interfaces ⌈ For each function to be provided by a BSW module, an UML interface (the "function interface") shall be created in its module package. The stereotype of the interface shall be "interface". ⌋*()*

[TR_BSWMG_00013] Naming of function interfaces ⌈ The *function interface* shall have the same name as the actual function. (depends on TR_BSWMG_00017, TR_BSWMG_00030) ⌋*()*

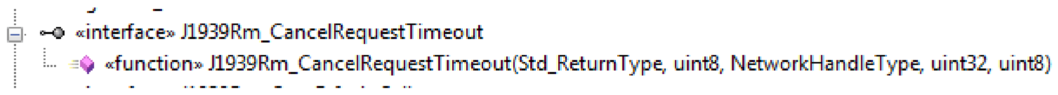

«interface» J1939Rm_CancelRequestTimeout
«function» J1939Rm_CancelRequestTimeout(Std_ReturnType, uint8, NetworkHandleType, uint32, uint8)

Figure 2.2: Naming example of a function interface

[TR_BSWMG_00014] API functions in component diagrams ⌈ API functions shall be visible in the providing BSW module's component diagram. ⌋*()*

Note: The easiest way to achieve this is to drag the new Interface directly into the providing module's component diagram when creating the interface.

[TR_BSWMG_00015] Realization relationships ⌈ The BSW module providing the service shall have a directed "Realization" association to the interface. The association shall be stereotyped «realize». ⌋*()*

Note: To differ associations from explanatory diagrams from generation relevant associations the stereotype «realize» has to be added to each generation relevant realize association.

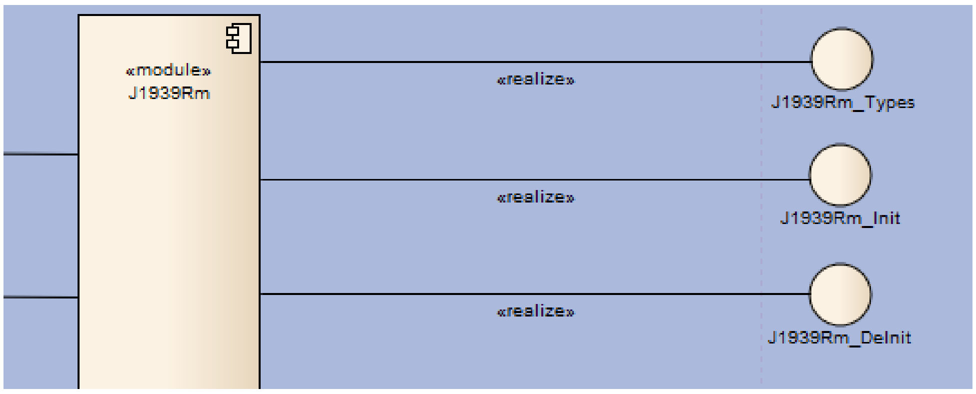


Figure 2.3: Realization example of an interface

2.3.3 API Functions

[TR_BSWMG_00016] API Functions ⌈ The function itself shall be modeled as an UML operation (the “operation”) having one of the following stereotypes: «function», «scheduled_function», «callout», «callback». ⌋ *()*

[TR_BSWMG_00017] Naming of API Functions ⌈ The name of the *operation* shall be the API function name. ⌋ *()*

[TR_BSWMG_00030] Name prefixes ⌈ The name of the *operation* shall be prefixed with the name of the realizing module (module abbreviation) followed by an underscore, i.e.: `<Ma>_<operation_name>` (*Ma = Module Abbreviation*) ⌋ *()*

[TR_BSWMG_00018] Location of Operations ⌈ The operation shall be placed into its corresponding provider’s realized interface. ⌋ *()*

[TR_BSWMG_00019] API Function documentation ⌈ Each API function shall provide a short description. ⌋ *()*

Note: EA provides a text-field called ‘Notes’ to take the operation’s description.

[TR_BSWMG_00034] “Return Type” field in operation ⌈ The operation’s “Return Type” field shall be left empty. See TR_BSWMG_00023 for modeling return parameters. ⌋ *()*

[TR_BSWMG_00024] Service ID ⌈ The tagged value “ServiceID” shall contain a service identifier (the ‘Service ID’) which shall be unique within the BSW module. The parameter is specified in hexadecimal notation using lowercase characters and shall be padded to two hexadecimal digits, e.g. 0x0d ⌋ *()*

[TR_BSWMG_00025] Reentrancy ⌈ The tagged value “Reentrant” shall determine whether the function needs to be implemented as reentrant or not. Allowed values are “Reentrant”, “Non Reentrant”, “Conditionally Reentrant”. Reentrancy conditions shall not be in the scope of the BSW UML model; instead, they shall be moved into individual SWS items (i.e.: “Non Reentrant for the same device.”) ⌋ *()*

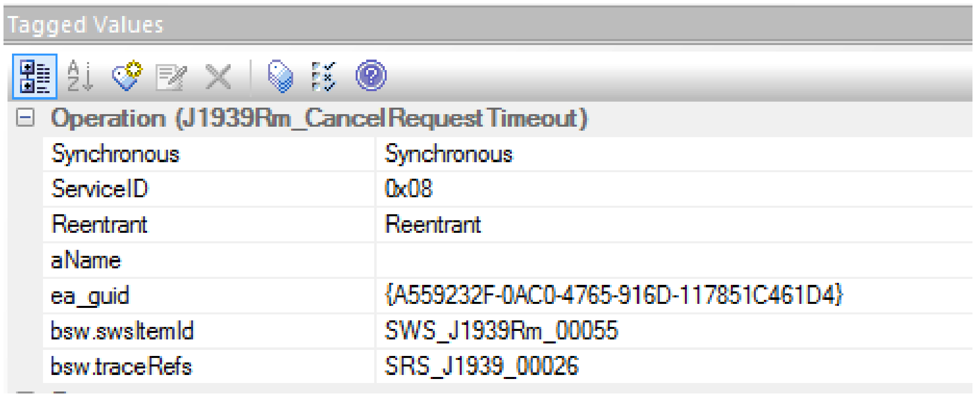
[TR_BSWMG_00026] Synchronicity ⌈ The tagged value “Synchronous” shall be set either to “Synchronous” or “Asynchronous”. Some modules may specify additional clauses. ⌋*()*

[TR_BSWMG_00031] Alternative Anchor Name ⌈ The optional tagged value “aName” is used to specify an alternative anchor name for the operation to be used when generating html-references within the BSW artifacts. This alternative anchor name shall be used in cases where the default anchor name is not suitable for usage, i.e. because it exceeds the length limit for links in Word or it includes non-standard characters. ⌋*()*

[TR_BSWMG_00150] SWS Item ID of a API function ⌈ The tagged value “bsw.swsItemId” is used to specify the SWS Item ID of a API function. ⌋*()*

[TR_BSWMG_00151] Up-traces of a API function ⌈ The tagged value “bsw.traceRefs” is used to specify up-traces to requirements. Multiple requirement IDs have to be separated by a comma. ⌋*()*

[TR_BSWMG_00140] Header File Reference of a API function ⌈ The tagged value “bsw.headerFile” is used to specify the header file where the API function is provided. ⌋*()*



Operation (J1939Rm_CancelRequestTimeout)

Synchronous	Synchronous
ServiceID	0x08
Reentrant	Reentrant
aName	
ea_guid	{A559232F-0AC0-4765-916D-117851C461D4}
bsw.swsItemId	SWS_J1939Rm_00055
bsw.traceRefs	SRS_J1939_00026

Figure 2.4: TaggedValues example of a API function

2.3.3.1 Scheduled Functions

[TR_BSWMG_00037] Stereotype for Scheduled Functions ⌈ Scheduled Functions shall be modeled by setting the operation’s stereotype to «scheduled_function». ⌋*()*

Note: The Schedule attribute of a Scheduled Function is not used in any artifact any more.

2.3.4 API Function Parameters

[TR_BSWMG_00020] Function Parameters ⌈ The function parameters shall have mandatory entries for “Name”, “Type”, “Direction” and “Notes”. ⌋*()*

[TR_BSWMG_00032] Parameter types ⌈ The parameter “Type” shall be one of the existing types defined in the BSW model. ⌋*()*

[TR_BSWMG_00033] C-Style Pointers ⌈ Parameters may be modeled as C-Style pointers by appending * to the parameter type, e.g. `PduInfoType*` ⌋*()*

[TR_BSWMG_00027] Mandatory pointers for output parameters ⌈ Parameters must be modeled as pointers if their “Direction” attribute is set to `out` or `inout`. ⌋*()*

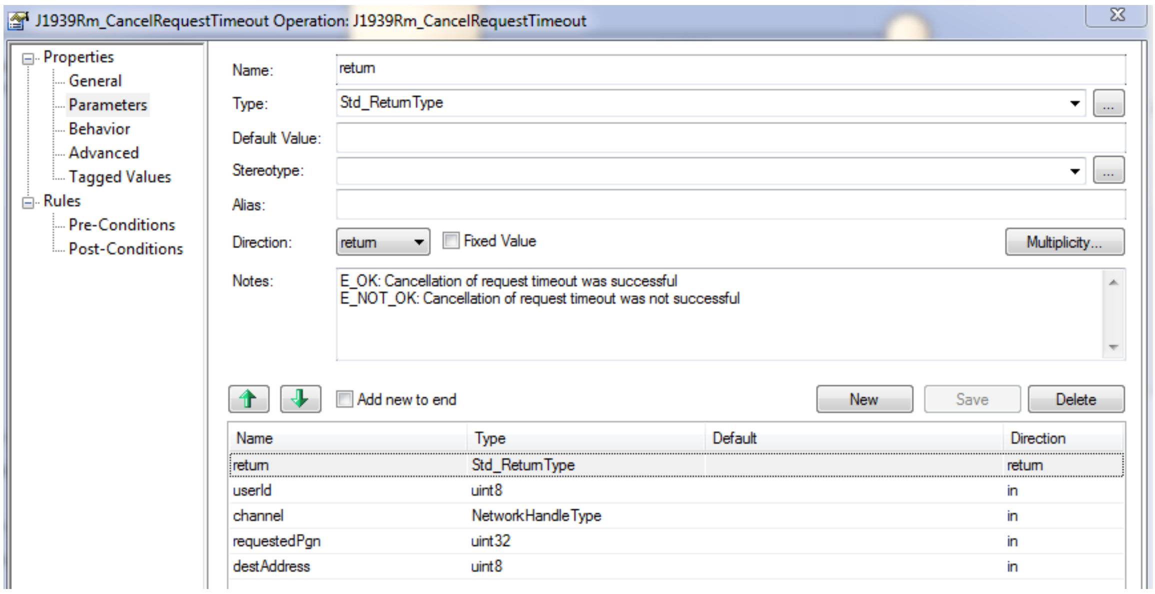
[TR_BSWMG_00035] Mandatory constant types for pointers of input parameters ⌈ Pointer-type parameters of direction type “in", i.e. parameters that represent read-only structures or arrays, may prepend the parameter type with the `const` keyword. This enforces that the data pointed to by the parameter is read-only and will not be altered by the function. Example: `const FrIf_ConfigType*` ⌋*()*

[TR_BSWMG_00021] Parameter Direction ⌈ The parameter’s direction type attribute shall be set to one of the values `in`, `out`, `inout`, `return`. ⌋*()*

[TR_BSWMG_00022] Parameter Description ⌈ Each parameter shall provide a short description about its purpose. ⌋*()*

Note: EA provides a textfield called ‘Notes’ to take the parameters description.

[TR_BSWMG_00023] Return Parameter ⌈ If the function’s return type is not equal to `void`, the return value shall be modeled like an operation parameter with the following exceptions: It shall be the first parameter in the list. Additionally, it shall be the only parameter to have its “Direction” set to “return”. The notes field shall concisely describe the possible return values. ⌋*()*



Name: return
Type: Std_ReturnType
Default Value:
Stereotype:
Alias:
Direction: return
Notes: E_OK: Cancellation of request timeout was successful
E_NOT_OK: Cancellation of request timeout was not successful

Name	Type	Default	Direction
return	Std_ReturnType		return
userId	uint8		in
channel	NetworkHandleType		in
requestedPgn	uint32		in
destAddress	uint8		in

Figure 2.5: Parameter example of a API function

[TR_BSWMG_00129] Optional Parameters ⌈ The existence of a parameter may depend on the module configuration. In this case, the parameter shall have the stereotype «optional». ⌋*()*

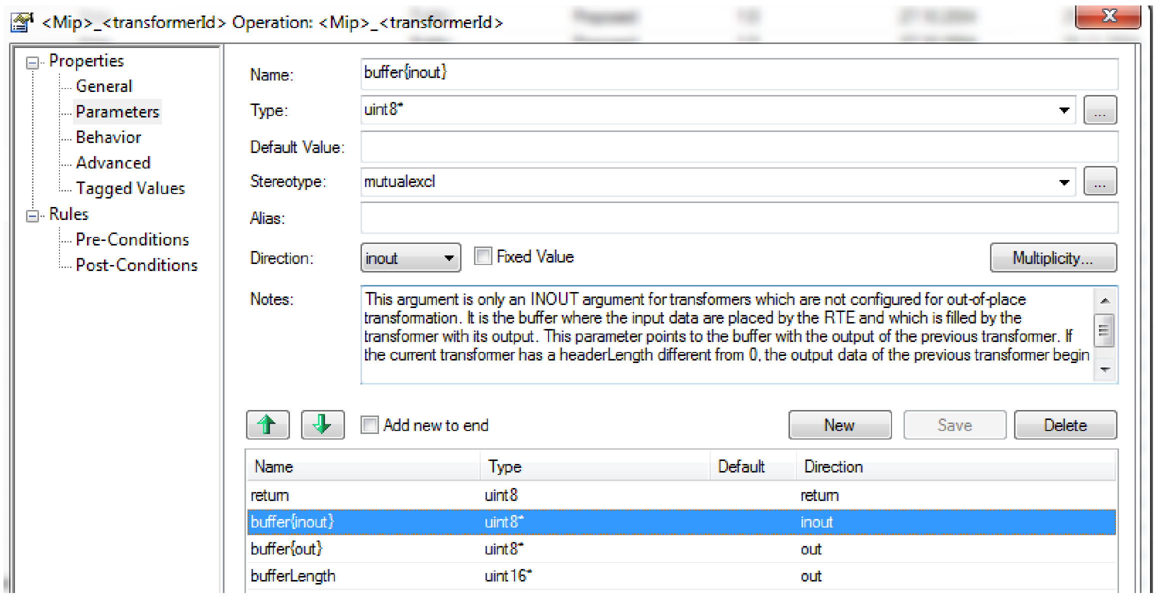
[TR_BSWMG_00130] Multiplicity of Parameters ⌈ A parameter with a given type may occur several times, where the multiplicity is specified by configuration. In this case, the parameter shall have the stereotype «multiple». ⌋*()*

Hint: Don't use the multiplicity button in the parameter edit mask to configure the multiplicity.

[TR_BSWMG_00131] Mutual Exclusive Variants of Parameters ⌈ A parameter may appear in different variants within the same position of the function signature, where one specific variant will be selected by configuration. In this case, the parameter shall be modeled several times in all its possible variants and each variant of the parameter shall have the stereotype «mutualexcl». ⌋*()*

Example: The parameter "buffer" of the Xfrm function "<Mip>_<transformerId>" can be configured either as "inout" or "out". It therefore shall be modeled two times, one time with Direction "inout" and the second time with Direction "out". Since Enterprise Architect requires parameter names to be unique, the first parameter variant may be named "buffer{inout}" and the second one "buffer{out}".

Hint: All variants of a mutual exclusive parameter shall have the same name; the name without the curly brackets (including the text) have to be the same.



<Mip>_<transformerId> Operation: <Mip>_<transformerId>

Name:	buffer{inout}
Type:	uint8*
Default Value:	
Stereotype:	mutualexcl
Alias:	
Direction:	inout ☐ Fixed Value
Notes:	This argument is only an INOUT argument for transformers which are not configured for out-of-place transformation. It is the buffer where the input data are placed by the RTE and which is filled by the transformer with its output. This parameter points to the buffer with the output of the previous transformer. If the current transformer has a headerLength different from 0, the output data of the previous transformer begin

Name	Type	Default	Direction
return	uint8		return
buffer{inout}	uint8*		inout
buffer{out}	uint8*		out
bufferLength	uint16*		out

Figure 2.6: Mutual Exclusive Parameter example of a API function

2.3.5 Module Dependencies

2.3.5.1 Virtual Interfaces

In general AUTOSAR BSW modules require functions from the APIs of other BSW modules in order to fulfill their own functionality. The general modeling pattern of dependencies between one BSW module and another uses so called *function interfaces* and *virtual interfaces*.

First, due to the fact that dependencies between APIs sometimes have to be expressed on a single API level of detail, each API function requires a representation on module level. For this purpose, the *function interfaces* have been introduced. (see 2.3.2)

Second, in order to further enhance the expressiveness of the BSW module, the concept of function interfaces is extended by *virtual interfaces*. *Virtual interfaces* are derived from *function interfaces* to merge a certain set of API functions. Recursive structures of *virtual interfaces* are also allowed, so a *virtual interface* is allowed to be derived from other *virtual interfaces*. This concept basically allows to reduce the number of module dependencies on the 'client' side, by providing a single *virtual interface* per providing module, collecting all functions required by this module.

[TR_BSWMG_00028] Virtual Interfaces ⌈ A *virtual interface* shall be modeled as an interface with the stereotype «interface» (just like a normal interface). ⌋*()*

[TR_BSWMG_00029] Naming of Virtual Interface ⌈ The name of a *virtual interface* shall consist of the name of the depending BSW module, the name of the providing module and the kind of dependency, each part separated by an underscore.

`<NameOfDependingModule>_<NameOfProvidingModule>_<KindOfRelation>`

⌋*()*

[TR_BSWMG_00039] Virtual Interface Multiplicity ⌈ One *virtual interface* refers to exactly one pair of modules of realizing and depending modules. ⌋*()*

[TR_BSWMG_00040] Virtual Interface Location ⌈ A *virtual interface* shall be placed in the module package of the depending BSW module. ⌋*()*

[TR_BSWMG_00041] Virtual Interface Contents ⌈ A *virtual interface* shall inherit its functions from the realizer's function interfaces. ⌋*()*

[TR_BSWMG_00042] No Mixed Usage of Function Interfaces and Virtual Interfaces ⌈ Depending modules shall either directly depend on another module's function interfaces approach or depend on a virtual interface towards the other module. The two approaches shall not be mixed. ⌋*()*

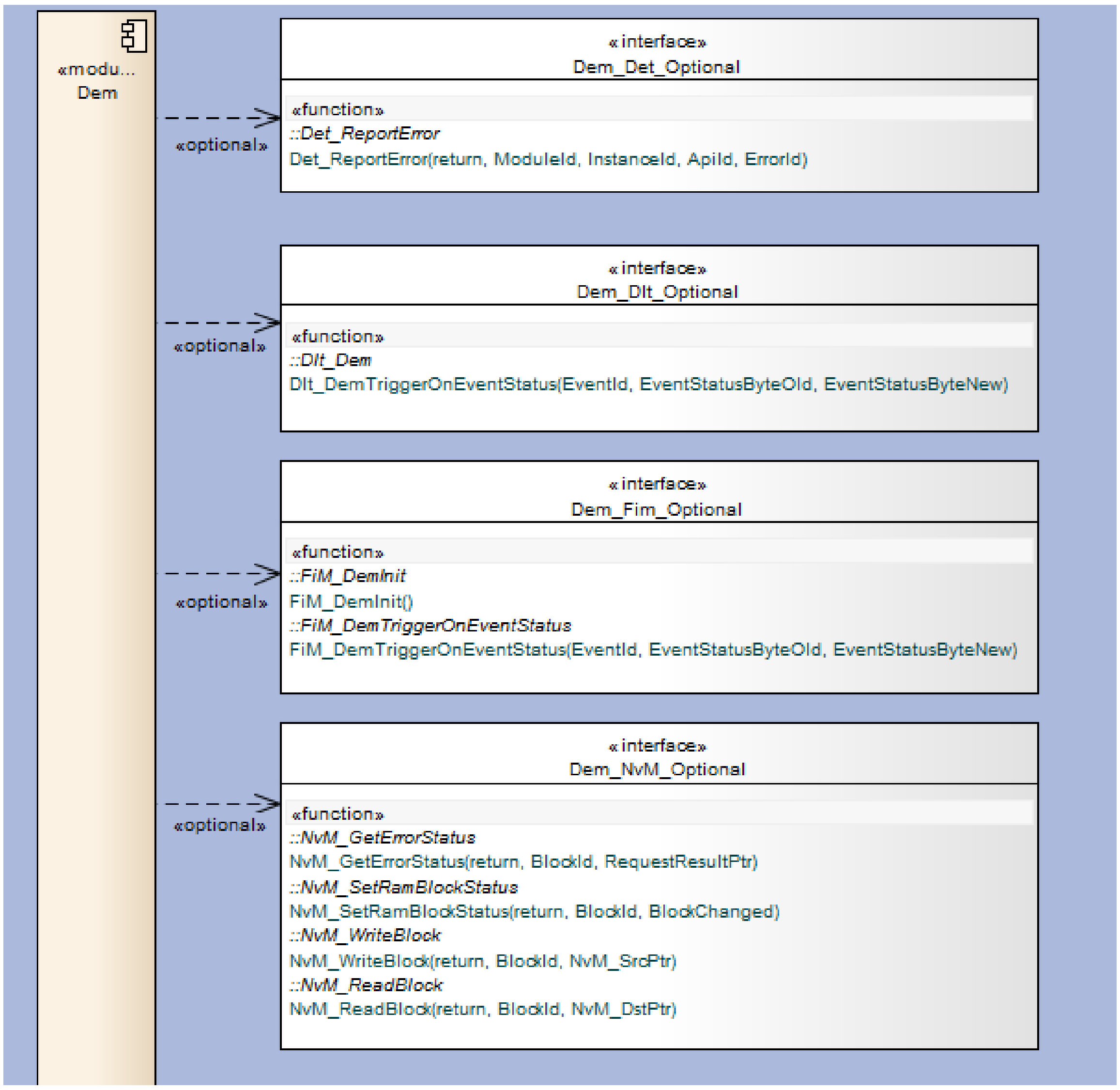

«modu...
Dem
«optional»
«interface»
Dem_Det_Optional
«function»
::Det_ReportError
Det_ReportError(return, ModuleId, InstanceId, ApiId, ErrorId)
«optional»
«interface»
Dem_Dlt_Optional
«function»
::Dlt_Dem
Dlt_DemTriggerOnEventStatus(EventId, EventStatusByteOld, EventStatusByteNew)
«optional»
«interface»
Dem_Fim_Optional
«function»
::FiM_DemInit
FiM_DemInit()
::FiM_DemTriggerOnEventStatus
FiM_DemTriggerOnEventStatus(EventId, EventStatusByteOld, EventStatusByteNew)
«optional»
«interface»
Dem_NvM_Optional
«function»
::NvM_GetErrorStatus
NvM_GetErrorStatus(return, BlockId, RequestResultPtr)
::NvM_SetRamBlockStatus
NvM_SetRamBlockStatus(return, BlockId, BlockChanged)
::NvM_WriteBlock
NvM_WriteBlock(return, BlockId, NvM_SrcPtr)
::NvM_ReadBlock
NvM_ReadBlock(return, BlockId, NvM_DstPtr)

Figure 2.7: Virtual interfaces example for defining optional interfaces

2.3.5.2 Mandatory Interfaces

[TR_BSWMG_00043] Stereotype for Mandatory Dependencies on Interfaces ⌈ The user module shall have dependencies with stereotype «mandatory» to all its mandatory interfaces. ⌋*()*

[TR_BSWMG_00044] Mandatory Dependencies ⌈ All *mandatory* dependencies held by a user module onto a provider module shall be modeled as exactly one virtual interface. ⌋*()*

[TR_BSWMG_00045] Naming of Mandatory Usage Collections ⌈ The virtual interface shall have the name <user_module>_<provider_module>_Mandatory. ⌋*()*

2.3.5.3 Optional Interfaces

[TR_BSWMG_00046] Stereotypes for Optional Dependencies on Interfaces ⌈ The user module shall have dependencies with stereotype «optional» to all its optional interfaces. ⌋*()*

[TR_BSWMG_00047] Optional Dependencies ⌈ All *optional* dependencies held by a user module onto a provider module shall be modeled as exactly one virtual interface. ⌋*()*

[TR_BSWMG_00048] Naming of Optional Usage Collections ⌈ The virtual interface shall have the name <user_module>_<provider_module>_Optional. ⌋*()*

2.3.6 Generic Interface

In some occasions the AUTOSAR BSW stack defines a number of interfaces which have basically the same function signature but slightly differ with regards to a module-specific naming. In these cases, redundant definitions of interfaces shall be prevented by usage of only one interface definition. To define a specific naming, a "Custom Interface" shall be used. A "Custom Interface" inherits from the interface definition and can override the naming rules.

The following modeling pattern shall be used for defining "Generic Interfaces":

[TR_BSWMG_00061] Generic Interface Definition ⌈ The function definition shall be placed into an UML interface having the stereotype «generic_interface». ⌋*()*

[TR_BSWMG_00132] Generic Interface Definition ⌈ This "generic interface" definition shall be considered abstract and not directly be referenced by any module. ⌋*()*

[TR_BSWMG_00133] Generic Interface Definition ⌈ A Generic Interface Definition shall contain ONE operation with stereotype «function_blueprint». ⌋*()*

[TR_BSWMG_00134] Custom Interface ⌈ In order to assign a “Custom Interface” to a “Generic Interface Definition”, a generalization association shall be modeled (target “Generic Interface Definition”). ⌋ *()*

[TR_BSWMG_00062] Custom Interface ⌈ In order to assign a concrete naming pattern to a function defined within a generic interface, another interface shall be defined having the same stereotype «generic_interface». ⌋ *()*

Note: Due to prevent reworking of the existing model and the generator tooling the interface of “Generic Interface Definition” and “Custom Interface” uses the same stereotype.

[TR_BSWMG_00156] Custom Interface ⌈ A Custom Interface shall not contain a function definition. ⌋ *()*

[TR_BSWMG_00063] Provider Naming Scheme ⌈ The naming pattern for a provider association («realize») of a module on a custom interface shall be configured using the tagged value “naming_provider”. ⌋ *()*

[TR_BSWMG_00064] User Naming Scheme ⌈ The naming pattern for a user dependency («mandatory» or «optional») of a module on a custom interface shall be configured using the tagged value “naming_user”. ⌋ *()*

[TR_BSWMG_00065] User-Configurable Naming Scheme ⌈ The naming pattern for a «configurable» dependency of a module on a custom interface shall be configured using the tagged value “naming_configurable”. ⌋ *()*

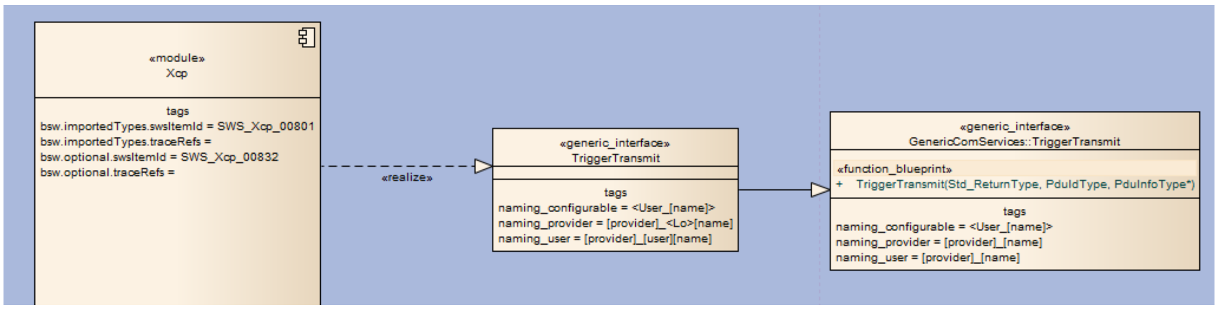

«module»
Xcp
tags
bsw.importedTypes.swsItemId = SWS_Xcp_00801
bsw.importedTypes.traceRefs =
bsw.optional.swsItemId = SWS_Xcp_00832
bsw.optional.traceRefs =
«realize»
«generic_interface»
TriggerTransmit
tags
naming_configurable = <User_[name]>
naming_provider = [provider]_<Lo>[name]
naming_user = [provider]_[user][name]
«generic_interface»
GenericComServices::TriggerTransmit
«function_blueprint»
+ TriggerTransmit(Std_ReturnType, PduIdType, PduInfoType*)
tags
naming_configurable = <User_[name]>
naming_provider = [provider]_[name]
naming_user = [provider]_[name]

Figure 2.8: Generic Interface/Custom Interface example

2.3.7 Callback Notifications

In AUTOSAR, a “callback” is defined as functionality in an upper-layer BSW module that is called by a lower-layer module in order to provide notification as required[3].

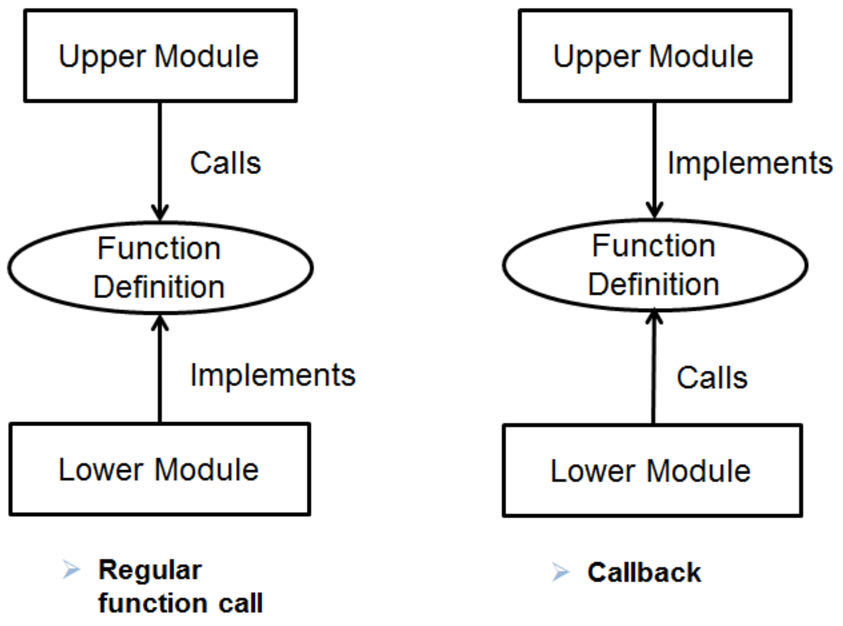


Figure 2.9: Difference between a callback and a regular function call

2.3.7.1 Callback definition and usage (non Configurable Callback)

[TR_BSWMG_00157] Callback definition ⌈ Callback definitions shall be modeled as UML operations and shall use the stereotype «callback». ⌋*()*

[TR_BSWMG_00158] Callback interface ⌈ For each callback to be called by a BSW module, an UML interface (the "function interface") shall be created in its module package. The stereotype of the interface shall be «interface». ⌋*()*

[TR_BSWMG_00159] Naming of callback interfaces ⌈ The *callback interface* shall have the same name as the actual function. (depends on TR_BSWMG_00017, TR_BSWMG_00030) ⌋*()*

[TR_BSWMG_00161] Callback function definition ⌈ The *callback interface* shall contain one function with stereotype «callback». The modeling of the function shall be as described in 2.3.3. ⌋*()*

[TR_BSWMG_00162] Callback function usage/call ⌈ For all callbacks a lower module can call, a dependency of stereotype «mandatory» or «optional» (target callback interfaces) shall be modeled, see chapter 2.3.5.2 and 2.3.5.3. ⌋*()*

[TR_BSWMG_00163] Callback function realization/implementation ⌈ For all callbacks a upper module implements, a realization of stereotype «realize» (target callback interfaces) shall be modeled. ⌋*()*

[TR_BSWMG_00164] Callback function realization/implementation ⌈ Each callback interface shall be referenced by exactly ONE dependency of stereotype «mandatory» or «optional» or «configurable» (There is only one caller, but multiple implementation can exit and will be assigned by configuration). ⌋*()*

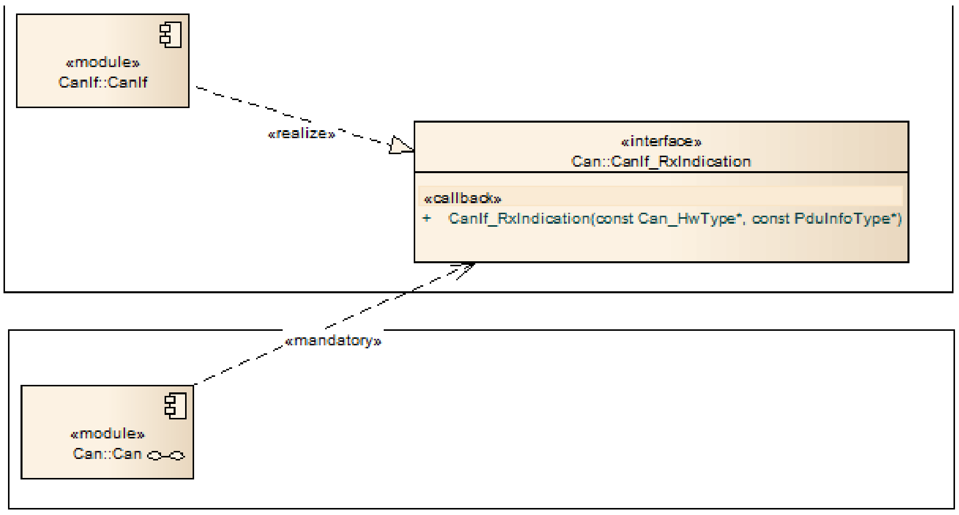


Figure 2.10: Example of a Callback definition and usage

2.3.7.2 Configurable Callback definition and usage

Lower-layer modules are caller of callbacks. Often, these modules can configure which actual instance of a callback definition will be called, i.e. which upper layer will be called in the callback situation. In the SWS the configurability of a callback is described in two parts: An API table for the configurable callback function including details like the callback signature and arguments, and the actual ECU configuration parameters described in chapter 10.

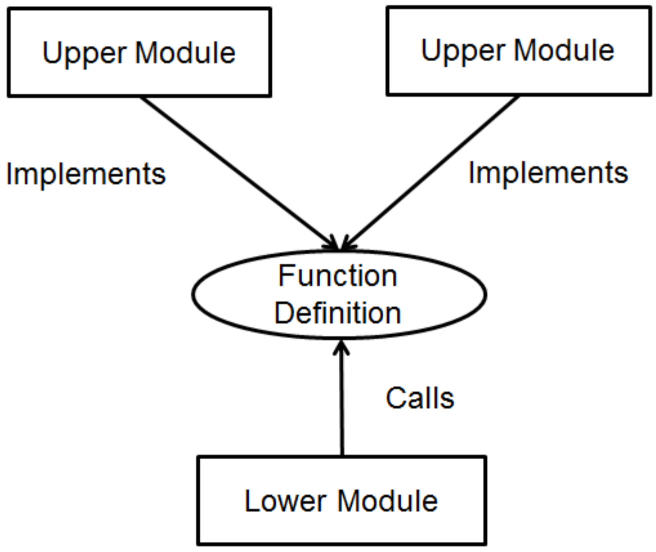


Figure 2.11: Configurable Callback: The lower module have to be configured which implementation of an upper module will be called.

[TR_BSWMG_00059] Configurable Dependency ⌈ A lower-layer module shall have dependencies with stereotype «configurable» to each of its configurable callback definitions. ⌋*()*

[TR_BSWMG_00060] Target of a Configurable Dependency is a Generic Definition ⌈ A lower-layer module shall have a generic callback definition as target of the configurable dependency. ⌋*()*

The naming of callback functions currently differs between BSW modules, and in particular between BSW stacks. Therefore, no definitive rules for naming patterns of callbacks can be stated here. However, as a guideline for adding new callback functions, either one of the following patterns should be followed:

(1) Module abbreviation, followed by an underscore, followed by the callback function name. When used in connection with generic callback definitions, the UML interfaces should get the tagged value `naming_configurable = [user]_[name]`.

(2) The literal string "User", followed by an underscore, followed by the callback function name. Additionally, the whole function name is put in angular brackets "<>" in order to emphasize that this is just a placeholder for the real, configurable name. When used in connection with generic callback definitions, the UML interfaces should get the tagged value `naming_configurable = <User_[name]>`.

2.3.7.3 Callback Generic Interfaces

Similar to the definition of *Generic Interfaces*, it is possible to define *Generic Interfaces* for Callbacks. The purpose of a *Callback Generic Interface* is to serve as a one-time definition of a callback. The callback may then be referenced in different contexts, using differently constructed names in the contexts of different modules, and also varying in attributes like Service ID.

[TR_BSWMG_00049] Callback Generic Interface Definitions ⌈ Callback definitions shall be modeled as UML operations and shall use the stereotype «callback» and «function_blueprint». ⌋*()*

[TR_BSWMG_00050] Callback Generic Interface Name ⌈ The Callback definition name shall just be the name part describing the actual functionality. In particular, it shall not contain the name of the provider or user, or literal "<User>" string etc. ⌋*()*

[TR_BSWMG_00051] Callback Blueprint Interface placement ⌈ Each callback blueprint definition shall be placed inside an UML interface having the same name as the contained operation with stereotype «generic_interface». ⌋*()*

[TR_BSWMG_00052] Callback Blueprint interface model location ⌈ The *generic interface* definition shall be located within the top-level package "Generic Elements". ⌋ *()*

[TR_BSWMG_00053] Callback Blueprint interface subpackage location ⌈ Within package “Generic Elements” the Callback Blueprint interface shall be placed in a sub package representing the BSW stack associated with the interface:

- ComStack
- IoStack
- MemoryStack

⌋*()*

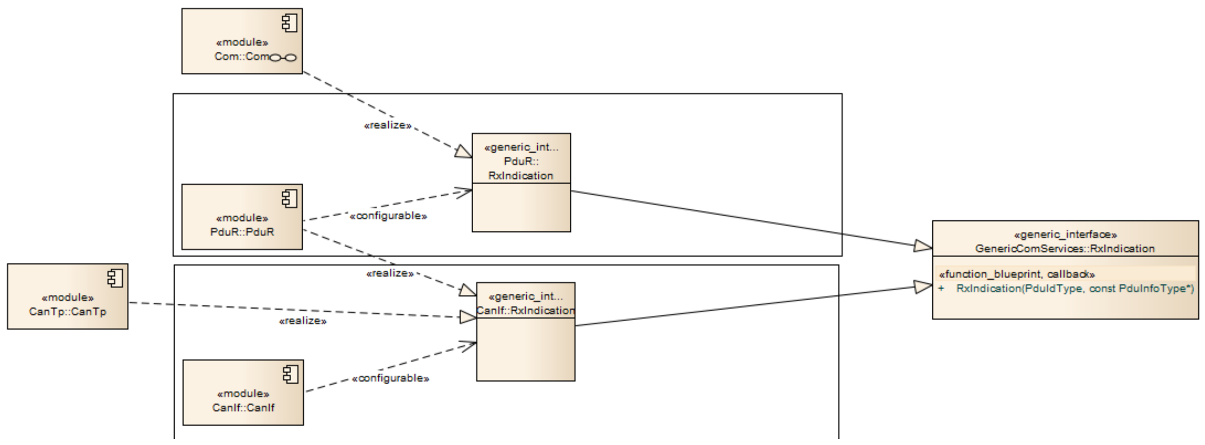


Figure 2.12: Example of Configurable Callback realized with Generic Interfaces

2.3.8 Data Type Definitions

2.3.8.1 Simple Types

[TR_BSWMG_00066] Simple Type Definition ⌈ Each simple type definition, i.e. a type definition which is directly derived from another type or which defines a basic type like ‘int’ shall be modeled as an UML Class with stereotype «type». ⌋*()*

[TR_BSWMG_00067] Base Types vs. Derived Types ⌈ A simple type definitions shall either define a base type or be derived from another data type definition. ⌋*()*

[TR_BSWMG_00068] Base Type Dependencies ⌈ A base type shall not derive from another data type. ⌋*()*

[TR_BSWMG_00069] Derived Types Dependencies ⌈ Normally, a derived type shall be derived from exactly only one other data type. However, if the type depends on platform or is configuration specific, it may be derived from more than one type. ⌋*()*

[TR_BSWMG_00071] Range ⌈ If a simple type has a restricted set of ranges, an attribute with stereotype «range» has to be created for each such range. The name of the attribute specifies the range label and the notes field describes the range. ⌋*()*
Example: Name: “0..2^16-1”

Hint: “Name” is the name of the editing field in the EA edit mask.

[TR_BSWMG_00152] SWS Item ID of a type ⌈ The tagged value “bsw.swsItemId” is used to specify the SWS Item ID of a API function. ⌋*()*

[TR_BSWMG_00153] Up-traces of a type ⌈ The tagged value “bsw.traceRefs” is used to specify up-traces to requirements. Multiple requirement IDs have to be separated by a comma. ⌋*()*

[TR_BSWMG_00141] Header File Reference of a type ⌈ The tagged value “bsw.headerFile” is used to specify the header file where the type is provided. ⌋*()*

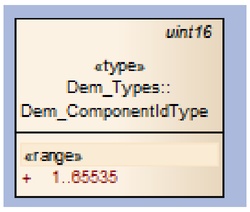


Figure 2.13: Simple type example

2.3.8.2 Enumerations

[TR_BSWMG_00072] Enumeration Definition ⌈ Each type definition representing an enumeration shall be modeled as UML Class with stereotype «enumeration». It shall be placed inside the interface specifying it. ⌋*()*

[TR_BSWMG_00073] Enumeration Literal Definition ⌈ All possible literals of the enumeration shall be modeled as attributes of this class. The order of the attributes from top to bottom shall represent the order of the enumeration specified. ⌋*()*

[TR_BSWMG_00074] Enumeration Literal Details ⌈ The following shall be respected for attributes:

- The Name field shall contain the literal name.
- The field “Type” shall be empty.
- The field “Stereotype” shall be empty.
- The field “Scope” shall be “Public”.
- The flag “Is Literal” shall be set.
- The field “Notes” shall contain the literal description.

⌋*()*

[TR_BSWMG_00075] Enumeration Literal Value ⌈ Literals may have a specified value; in this case it shall be placed in the field “Initial Value”. ⌋*()*

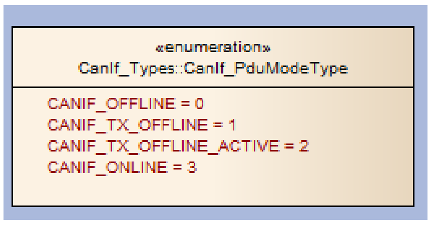


Figure 2.14: Enumeration example

2.3.8.3 Std_ReturnType Extensions

AUTOSAR defines a standard API return type that is being used throughout the BSW stack. It is also the only return type that can be used in ClientServer type Service Interface Operations.

“Std_ReturnType” is being defined in the SWS Standard Types [4] ([SRS_BSW_00377]). Additionally, two standard values `E_OK` and `E_NOT_OK` are defined which should normally be used with `Std_ReturnType`.

If these two return values are not sufficient, a BSW module is allowed to define additional return values to be used with Std_ReturnType. Such user defined values shall be prefixed with the module prefix and can be in the range $0x02–0x3f$.

[TR_BSWMG_00089] Std_ReturnType Extension Definition ⌈ The definition of a BSW module specific Std_ReturnType extension shall be modeled as an UML Class with stereotype «extra_literals». ⌋*()*

[TR_BSWMG_00090] Std_ReturnType Extension Name ⌈ The UML Class containing the Std_ReturnType extensions shall be named <Ma>_ReturnType (*Ma = Module Abbreviation*). ⌋*()*

[TR_BSWMG_00091] Std_ReturnType Extension Literal Definition ⌈ All BSW module specific possible return type extension literals shall be modeled as attributes of this class. The order of the attributes from top to bottom shall represent the order of the enumeration specified. ⌋*()*

[TR_BSWMG_00092] Std_ReturnType Extension Literal Details ⌈ The following fields shall be used for specifying the return type extension literals:

- The “Name” field shall contain the Std_ReturnType extension literal name.
- The field “Type” shall be empty.
- The field “Stereotype” shall be empty.
- The field “Scope” shall be “Public”.
- The flag “Is Literal” shall be set.
- The field “Notes” shall contain the description of the custom return value.

⌋()

[TR_BSWMG_00093] Std_ReturnType Extension Literal Value ⌈ Custom Std_ReturnType values shall always be defined with a specified unsigned integer value larger than 1 (i.e. E_NOT_OK). The integral value shall be placed in the field "Initial Value". ⌋()

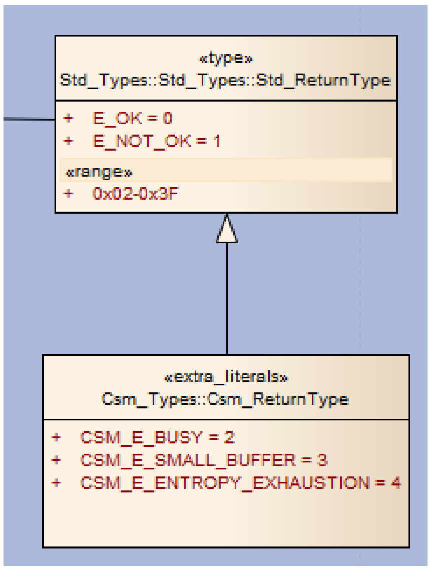


Figure 2.15: Std_ReturnType Extensions example

2.3.8.4 Structures

[TR_BSWMG_00076] Structure Type Definition ⌈ Each type definition representing a structure declaration shall be modeled as a UML Class with the stereotype «structure». ⌋()

[TR_BSWMG_00077] Structure Member Definition ⌈ All members of a structure shall be defined as attributes of this class. The ordering of the attributes shall be the same as expected in the generated table. ⌋()

[TR_BSWMG_00078] Structure Member Details ⌈ The following shall be respected for attributes:

- The "Name" field shall contain the name of the attribute.
- The field "Type" shall select the existing type.
- The field "Scope" shall be "Public".
- The "Containtment" shall be "Not Specified".

⌋()

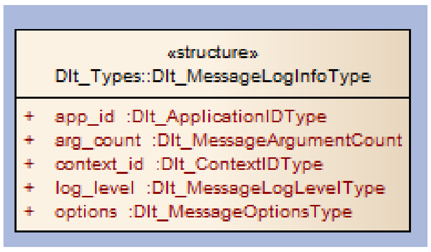

«structure»
Dlt_Types::Dlt_MessageLogInfoType
+ app_id :Dlt_ApplicationIDType
+ arg_count :Dlt_MessageArgumentCount
+ context_id :Dlt_ContextIDType
+ log_level :Dlt_MessageLogLevelType
+ options :Dlt_MessageOptionsType

Figure 2.16: Structure example

2.3.8.5 Bitfields

Bitfield types represent an efficient way of encoding a number of independent variables within one type. This is done by breaking down the bitfield type into compartments of individual bit flags or bit ranges containing a series of bits.

One typical application is the implementation of independent boolean variables or “bit flags” (i.e. binary flags); each of these flags takes up one bit in the bitfield, and can be set to true or false independently of all other flags contained in the type.

However, Bitfields are not limited to bit flags: They can also contain one or more groups of bits that can be interpreted as a small-range enumeration type per group (“bit range”).

Both use cases can be mixed and implemented in several instances within the same bitfield type. The definition of the bitfield compartments is done using bitmasks.

The bitfield type can additionally define value literals in order to assign concrete meaning to the masked values.

[TR_BSWMG_00079] Bitfield Type Definition ⌈ Each type definition representing a bitfield declaration shall be modeled as a UML Class with the stereotype «bitfield». ⌋*()*

[TR_BSWMG_00080] Bitfield: Bit Flag Definitions ⌈ Binary “bit flags” shall be modeled as attributes of the bitfield type using the stereotype «bitflag». ⌋*()*

[TR_BSWMG_00081] Bitfield: Bit Flag Value interpretation ⌈ The two possible values of “bit flags” shall always be interpreted as “TRUE” and “FALSE”. In case a binary value shall be interpreted differently, it shall be modeled as a Bit Range (see below). ⌋*()*

[TR_BSWMG_00082] Bitfield: Bit Flag Details ⌈ The following shall be respected for Bit Flag attributes:

- The “Name” field shall contain the name of the bit flag. (ARXML: Short-Label of CompuScale)
- The field “Type” shall be empty.
- The field “Initial Value” shall contain the bit flag value either in hexadecimal (e.g. 0x10) or in binary (e.g. 0b00010000) representation.

- The field “Stereotype” shall be «bitflag».
- The field “Alias” shall be empty.
- The field “Scope” shall be “Public”.
- The flag “Is Literal” shall not be set.
- The field “Notes” shall describe the meaning of the bit flag.

⌋*()*

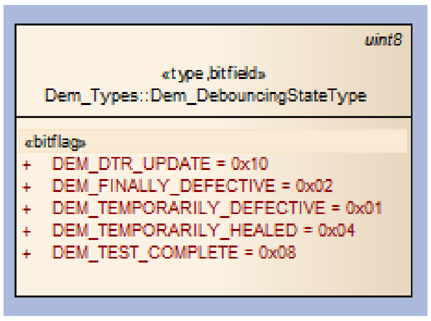

uint8
«type,bitfield»
Dem_Types::Dem_DebouncingStateType
«bitflag»
+ DEM_DTR_UPDATE = 0x10
+ DEM_FINALLY_DEFECTIVE = 0x02
+ DEM_TEMPORARILY_DEFECTIVE = 0x01
+ DEM_TEMPORARILY_HEALED = 0x04
+ DEM_TEST_COMPLETE = 0x08

Figure 2.17: Bitfield bitflag example

[TR_BSWMG_00083] Bitfield: Bit Range Definitions ⌈ Bit Ranges are continuous bit regions containing one or more bits. Bit Ranges shall be modeled as attributes of the bitfiled type using the stereotype «bitrange>. ⌋*()*

[TR_BSWMG_00084] Bitfield: Bit Range Details ⌈ The following shall be respected for Bit Range attributes:

- The “Name” field shall contain the name of the bit range. (ARXML: Short-Label)
- The field “Type” shall be empty.
- The field “Initial Value” shall contain a bit mask representing the bit range, see below.
- The field “Stereotype” shall be «bitrange».
- The field “Alias” shall be empty.
- The field “Scope” shall be “Public”.
- The flag “Is Literal” shall not be set.
- The field “Notes” shall describe the meaning of the bit range.

⌋*()*

[TR_BSWMG_00085] Bitfield: Bit Range Mask Value ⌈ The size and location of the bits used for the bit range shall be specified as a bitmask using either hexadecimal (e.g. 0x1c) or GCC binary notation, e.g. 0b00011100. This mask value shall be placed in the field “Initial Value”. ⌋*()*

[TR_BSWMG_00086] Bitfield: Bit Mask and Bit Range Order ⌈ The order of the attributes from top to bottom shall be increasing with bit flag or bit mask values (i.e. the value specified in field "Initial Value". ⌋*()*

[TR_BSWMG_00087] Bitfield: Bit Range Value Definition ⌈ A bit range can assume a number of different values. The meaning of these values shall be specified using tagged values on the bitfield type's attributes. ⌋*()*

[TR_BSWMG_00088] Bitfield: Bit Range Value Details ⌈ Each bit range value specification consists of a group up to three tagged values.

Each group starts with the keyword "value.", followed by a number starting from '0', followed by one of three keywords.

- "name" (e.g.: value.0.name): Name of the value. Example: "PHYS_REQ"
- "value" (e.g.: value.0.value): Hexadecimal or binary value; this shall lie within the range of one of the bit masks. Example: 0b00010000
- "description" (e.g.: value.0.description): Optional description of the value. Example: "physical request"

⌋*()*

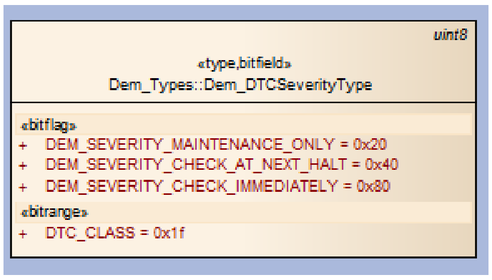
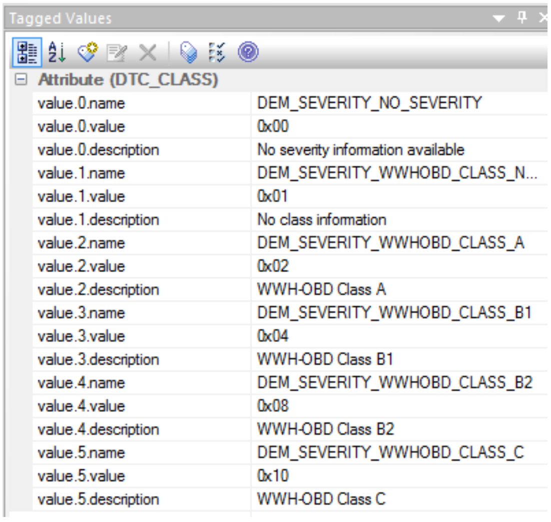

uint8
«type,bitfield»
Dem_Types::Dem_DTCSeverityType
«bitflag»
+ DEM_SEVERITY_MAINTENANCE_ONLY = 0x20
+ DEM_SEVERITY_CHECK_AT_NEXT_HALT = 0x40
+ DEM_SEVERITY_CHECK_IMMEDIATELY = 0x80
«bitrange»
+ DTC_CLASS = 0x1f

Figure 2.18: Bitfield Bitflag and Bitrange combined example



Attribute (DTC_CLASS)

value.0.name	DEM_SEVERITY_NO_SEVERITY
value.0.value	0x00
value.0.description	No severity information available
value.1.name	DEM_SEVERITY_WWHOBD_CLASS_N...
value.1.value	0x01
value.1.description	No class information
value.2.name	DEM_SEVERITY_WWHOBD_CLASS_A
value.2.value	0x02
value.2.description	WWH-OBD Class A
value.3.name	DEM_SEVERITY_WWHOBD_CLASS_B1
value.3.value	0x04
value.3.description	WWH-OBD Class B1
value.4.name	DEM_SEVERITY_WWHOBD_CLASS_B2
value.4.value	0x08
value.4.description	WWH-OBD Class B2
value.5.name	DEM_SEVERITY_WWHOBD_CLASS_C
value.5.value	0x10
value.5.description	WWH-OBD Class C

Figure 2.19: Taggedvalues of the Bitrange "DTC_CLASS"

2.3.8.6 Modeling of variability in data types

Many data types are configurable since they depend on the configuration of the basis software. Therefore so called “blueprint conditions” have been introduced into BSW model to express e.g. configurable inheritance.

[TR_BSWMG_00070] Derived Types Blueprint Conditions ⌈ If a type is derived from more than one other data type, the generalization dependency shall have a tagged value “Vh.BlueprintCondition” which specifies the exact conditions for selecting the associated data type. (e.g., Vh.BlueprintCondition = platform dependend) ⌋ *()*

[TR_BSWMG_00503] Blueprint Policies for CompuMethods ⌈ To define blueprint policies for a type’s compu method the tagged value “Vh.compuMethod.BlueprintPolicy” with the legal values “not-modifiable”, “list”, or “single” shall be used.

The blueprint derivation guide shall be described by the tagged value “Vh.compuMethod.BlueprintPolicy.DerivationGuide” or for multi-line guides by

- “Vh.compuMethod.BlueprintPolicy.DerivationGuide.1”
- “Vh.compuMethod.BlueprintPolicy.DerivationGuide.2”
- ...

It is not applicable for blueprint policy not-modifiable.

To explicitly set the maximal and minimal number of elements for a blueprint policy list the tagged values “Vh.compuMethod.BlueprintPolicy.maxElements” and “Vh.compuMethod.BlueprintPolicy.minElements” shall be used. ⌋ *()*

```
Vh.compuMethod.BlueprintPolicy:
  list
Vh.compuMethod.BlueprintPolicy.maxElements:
  3
Vh.compuMethod.BlueprintPolicy.minElements:
  1
Vh.compuMethod.BlueprintPolicy.DerivationGuide.1:
  0x00 is locked
Vh.compuMethod.BlueprintPolicy.DerivationGuide.2:
  0x01...0x3F is configuration dependent
Vh.compuMethod.BlueprintPolicy.DerivationGuide.3:
  0x40...0xFF is Reserved by Document
```

[TR_BSWMG_00504] Blueprint Policies for DataContraints ⌈ To define blueprint policies for a type’s data constr the tagged value “Vh.dataConstr.lowerLimit.BlueprintPolicy.DerivationGuide” for the lower limit and the tagged value “Vh.dataConstr.upperLimit.BlueprintPolicy.DerivationGuide” for the upper limit shall be used. ⌋ *()*

[TR_BSWMG_00505] Blueprint Policies for DataContraints explicit limits ⌈ To explicitly set the lower and upper limit the tagged values “Vh.dataConstr.lowerLimit.value” and “Vh.dataConstr.upperLimit.value” shall be used. ⌋ *()*

[TR_BSWMG_00506] Blueprint Policies for DataContraints explicit blueprintValues ⌈ To explicitly set the lower and upper blueprint-Value the tagged values "Vh.dataConstr.lowerLimit.blueprintValue" and "Vh.dataConstr.upperLimit.blueprintValue" shall be used. ⌋*()*

```
Vh.dataConstr.lowerLimit.BlueprintPolicy.DerivationGuide:
        For each user, a unique value must be defined at system
           generation time. Maximum number of users is 255. Legal user
            IDs are in the range 0 .. 254;
Vh.dataConstr.lowerLimit.blueprintValue:
        min 0
Vh.dataConstr.lowerLimit.value:
        undefined

Vh.dataConstr.upperLimit.BlueprintPolicy.DerivationGuide:
        For each user, a unique value must be defined at system
           generation time. Maximum number of users is 255. Legal user
            IDs are in the range 0 .. 254;
Vh.dataConstr.upperLimit.blueprintValue:
        max 254
Vh.dataConstr.upperLimit.value:
        undefined
```

[TR_BSWMG_00411] Configurable literals for Enumeration Types ⌈ For each BlueprintCondition delivering names of literals an attribute have to be defined. The name of the attribute have to be the namepattern e.g. ResetMode. ⌋*()*

[TR_BSWMG_00412] Configurable literals for Enumeration Types ⌈ The Blueprint-Condition delivering names of literals an attribute have to be defined on the attribute using the tagged value "Vh.BlueprintCondition". ⌋*()*

[TR_BSWMG_00413] Configurable literals for Enumeration Types ⌈ The value of configurable literals shall be defined on the attribute using the tagged value "Vh.BlueprintValue". The value field shall be set to the variable used in tagged value "Vh.BlueprintValue". ⌋*()*

Example of configurable literals for an enumeration type (EcuM_ShutdownModeType): The literals of this data type is a union of configured EcuMResetModes and EcuM-SleepModes. Therefor two attribute have to be modeled containing the condition to get the literals names and the condition to get the IDs for the literals.

```
Attribute name: {ResetMode}
Attribute value: {ResetModeId}

Vh.BlueprintCondition:
        ResetMode = {ecuc(EcuM/EcuMConfiguration/EcuMFlexConfiguration/
           EcuMResetMode.SHORT-NAME)}
Vh.BlueprintValue:
        ResetModeId = {256 + ecuc(EcuM/EcuMConfiguration/
           EcuMFlexConfiguration/EcuMResetMode.EcuMResetModeId)}
```

```
Attribute name: {SleepMode}
Attribute value : {SleepModeId}

```

```
Vh.BlueprintCondition:
        SleepMode = {ecuc(EcuM/EcuMConfiguration/
           EcuMCommonConfiguration/EcuMSleepMode.SHORT-NAME)}
Vh.BlueprintValue:
        SleepModeId = {ecuc(EcuM/EcuMConfiguration/
           EcuMCommonConfiguration/EcuMSleepMode.EcuMSleepModeId)}
```

2.3.9 Modeling of services (MoS)

Services are provided through ports. A port implements a port interface. A port interface can be a ClientServerInterface, a SenderReceiverInterface or a ModeSwitchInterface.

A ClientServerInterface defines the available service operations. A service operation defines return, input and output parameters. Each service operation has a relationship to an existing api function (c function). Blueprints allow to configure things like parameters, services, ...

A SenderReceiverInterface defines DataElements. Each data element have to be linked to a data type.

A ModeSwitchInterface defines modes within a ModeDeclarationGroup.

Note: To get a better understanding of the modeling, listing examples of the informal textual definition of service interfaces in AUTOSAR R4.0.3 SWS documents are used. If you are not familiar with this old definition, please ignore these listings.

2.3.9.1 Modeling of Client Server Interfaces

The following listing shows the old syntax of modeling / defining ClientServerInterface in AUTOSAR R4.0.3 SWS documents.

```
ClientServerInterface Csm_Hash {

        // errors assisioated with the ProtInterface
        PossibleErrors {
                CSM_E_NOT_OK       = 1
                CSM_E_BUSY         = 2
                CSM_E_SMALL_BUFFER = 3
        };


        //containing operations

        //parameter kinds can be IN, OUT and INOUT
        //
        //ERR is not a parameter
        //      -> should be a associated error to an operation
        HashStart (
                ERR(CSM_E_NOT_OK, CSM_E_BUSY)
        );

        HashUpdate (
                IN    HashDataBuffer    dataBuffer,
                IN    uint32    dataLength,
                ERR(CSM_E_NOT_OK, CSM_E_BUSY)
        );

        HashFinish (
                OUT   HashResultBuffer  resultBuffer,
                INOUT HashLengthBuffer  resultLength,
                IN    boolean   TruncationIsAllowed,
                ERR(CSM_E_NOT_OK, CSM_E_BUSY, CSM_E_SMALL_BUFFER)
        );
};
```

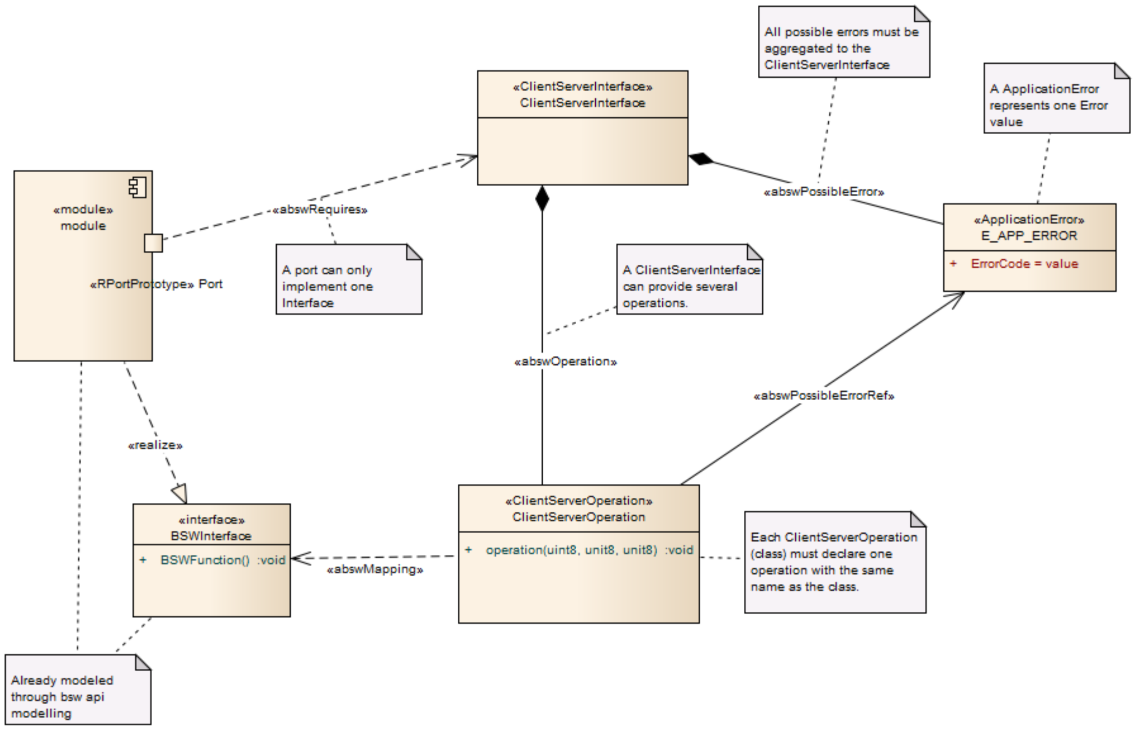

All possible errors must be aggregated to the ClientServerInterface
A ApplicationError represents one Error value
«ClientServerInterface» ClientServerInterface
«module» module
«RPortPrototype» Port
«absw Requires»
«absw PossibleError»
«ApplicationError» E_APP_ERROR
+ ErrorCode = value
A port can only implement one Interface
A ClientServerInterface can provide several operations.
«abswOperation»
«abswPossibleErrorRef»
«realize»
«interface» BSWInterface
+ BSWFunction() :void
«abswMapping»
«ClientServerOperation» ClientServerOperation
+ operation(uint8, unit8, unit8) :void
Each ClientServerOperation (class) must declare one operation with the same name as the class.
Already modeled through bsw api modelling

Figure 2.20: Schematic overview of Client Server Interfaces

[TR_BSWMG_00160] MoS ⌈ All additional elements of service modeling shall be placed in a package “ARInterfaces”. This packages shall be a child package of the module package. ⌋*()*

[TR_BSWMG_00100] MoS Port ⌈ A port shall be modeled as an UML port having one of the following stereotype «RPortPrototype», «PPortPrototype», «PRPortPrototype». The port shall be provided by the module component. ⌋*()*

[TR_BSWMG_00101] MoS Port ⌈ The port interface, that the port implements, shall be modeled as a dependency of stereotype «abswRequires» to a ClientServerInterface, a SenderReceiverInterface or a ModeSwitchInterface. ⌋*()*

[TR_BSWMG_00102] MoS Port Interface ⌈ A port interface shall be modeled as a class of stereotype «ClientServerInterface», a «SenderReceiverInterface» or a «ModeSwitchInterface». The stereotype «ClientServerInterface» shall be used to model a Client Server Interface. The stereotype «SenderReceiverInterface» shall be used to model a Sender Receiver Interface. The stereotype «ModeSwitchInterface» shall be used to model a Mode Switch Interface. ⌋*()*

[TR_BSWMG_00154] MoS Port Interface isService attribute ⌈ The value of the isService attribute of an interface is true by default. To set the attribute to false the tagged value "bsw.isService" shall be used. The value of the tagged value shall be "false". ⌋*()*

[TR_BSWMG_00103] MoS ClientServerInterface ⌈ The operations defined through a Client Server Interface shall be modeled as a class of stereotype «ClientServerOperation» per operation (for each operation a separate class). ⌋*()*

[TR_BSWMG_00104] MoS ClientServerInterface ⌈ The relationship between ClientServerInterface and ClientServerOperation shall be modeled as an aggregation of stereotype «abswOperation» (target ClientServerOperation). ⌋*()*

[TR_BSWMG_00105] MoS ClientServerOperation ⌈ Each class of stereotype «ClientServerOperation» shall contain one operation with the same name as the class. ⌋*()*

[TR_BSWMG_00106] MoS ClientServerOperation ⌈ The parameters of an operation shall be modeled as parameters of the UML operation. «ClientServerOperation» Class -> Operation -> Parameter ⌋*()*

[TR_BSWMG_00107] MoS ClientServerOperation ⌈ The parameter's "Kind" attribute shall be set to one of the values 'in', 'out', 'inout'. ⌋*()*

[TR_BSWMG_00108] MoS ClientServerInterface ⌈ For each possible error a class of stereotype «ApplicationError» shall be created. The name of the class shall be the error abbreviation (e.g. E_FORCE_RCRRP). The error code shall be modeled as a public attribute. The name shall be ErrorCode and the error code shall be modeled as initial value. ⌋*()*

[TR_BSWMG_00109] MoS ClientServerInterface ⌈ All possible errors of a Client Server Interface shall be referenced by an aggregation of stereotype «abswPossibleError» (target ApplicationError). ⌋*()*

[TR_BSWMG_00110] MoS ClientServerOperation ⌈ All possible errors of a Client Server Interface Operation shall be referenced by a dependency of stereotype «abswPossibleErrorRef» (target ApplicationError). ⌋*()*

[TR_BSWMG_00111] MoS ClientServerOperation ⌈ Each ClientServerOperation shall have a relationship to the corresponding bsw api function. So the relationship between ClientServerOperation and bsw api function (interface of the function) shall be modeled as a dependency of stereotype «abswMapping» (target bsw api function). ⌋*()*

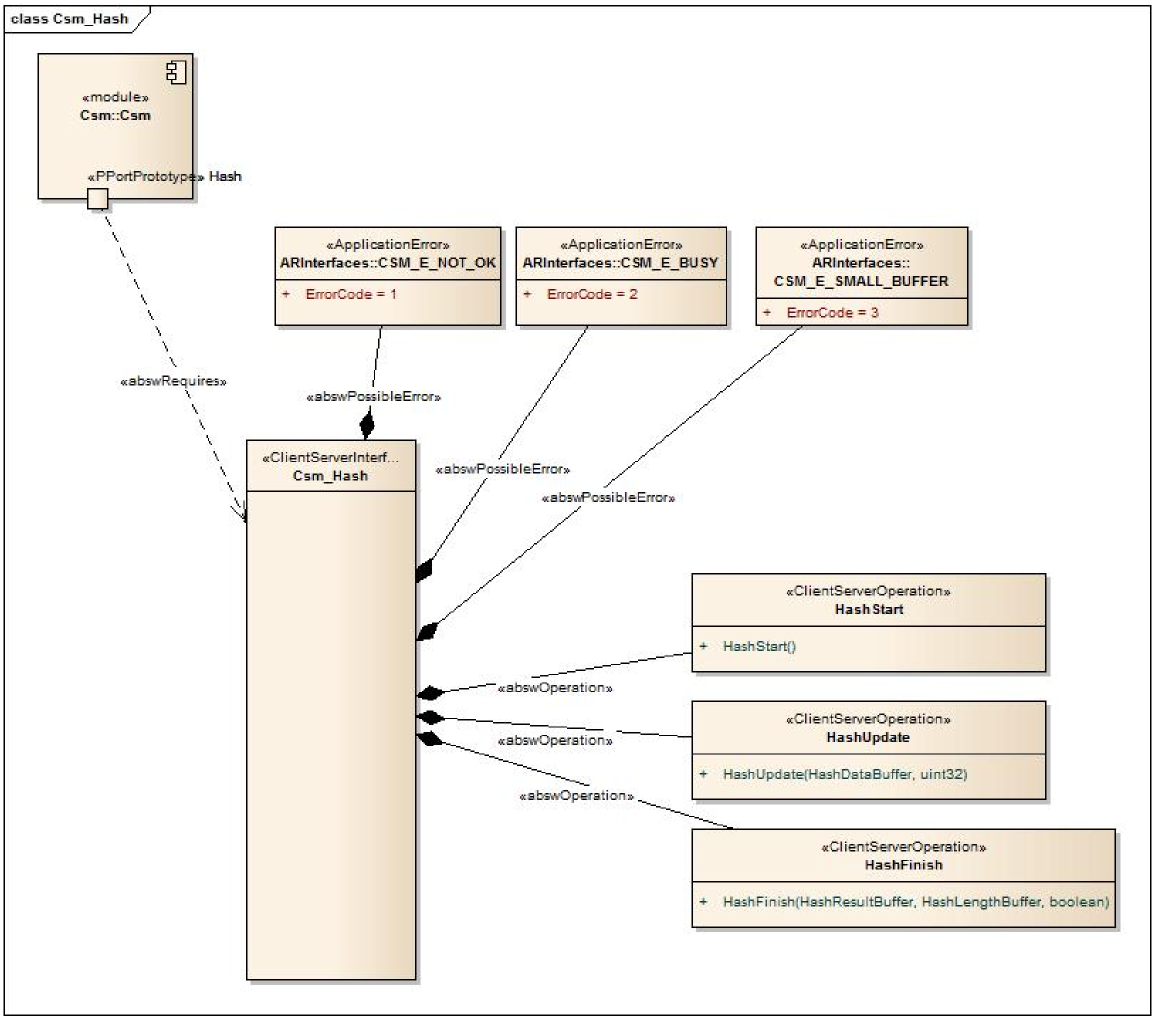


Figure 2.21: Example ClientServerInterface diagamm (CSI diagram)

[TR_BSWMG_00112] MoS ClientServerInterface ⌈ For each Client Server Interface a class diagram (CSI diagram) shall be created. The name of the diagramm shall be the name of the Client Server Interface. ⌋*()*

[TR_BSWMG_00113] MoS ClientServerInterface ⌈ A CSI diagram shall contain the module, the ClientServerInterface, the Application Errors of the ClientServerInterface and the ClientServerOperations of the ClientServerInterface. ⌋*()*

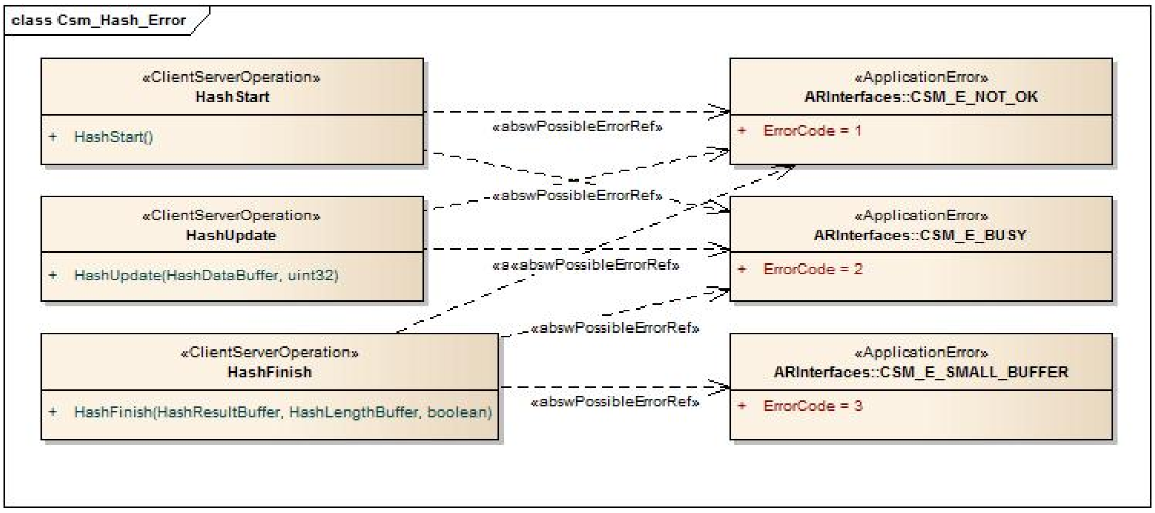


Figure 2.22: Example ClientServerInterface errors diagram (CSI errors diagram)

[TR_BSWMG_00114] MoS ClientServerInterface ⌈ For each Client Server Interface a class diagram (CSI errors diagram) shall be created. The name of the diagram shall be the name of the Client Server Interface concatenated with "_Error". ⌋*()*

[TR_BSWMG_00115] MoS ClientServerInterface ⌈ A CSI errors diagram shall contain the Application Errors of the ClientServerInterface and the ClientServerOperations of the ClientServerInterface. ⌋*()*

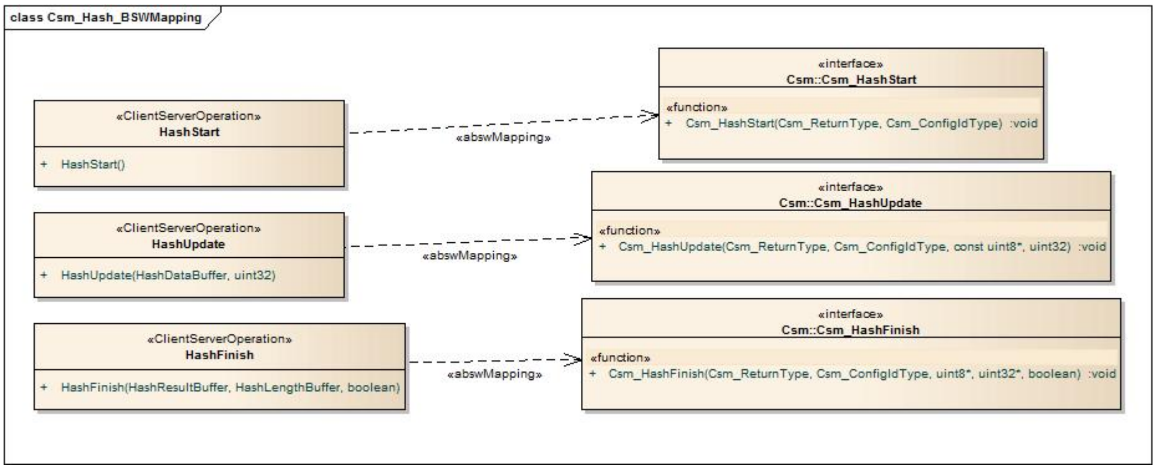

class Csm_Hash_BSWMapping
«ClientServerOperation»
HashStart
+ HashStart()
«ClientServerOperation»
HashUpdate
+ HashUpdate(HashDataBuffer, uint32)
«ClientServerOperation»
HashFinish
+ HashFinish(HashResultBuffer, HashLengthBuffer, boolean)
«abswMapping»
«abswMapping»
«abswMapping»
«interface»
Csm::Csm_HashStart
«function»
+ Csm_HashStart(Csm_ReturnType, Csm_ConfigIdType) :void
«interface»
Csm::Csm_HashUpdate
«function»
+ Csm_HashUpdate(Csm_ReturnType, Csm_ConfigIdType, const uint8*, uint32) :void
«interface»
Csm::Csm_HashFinish
«function»
+ Csm_HashFinish(Csm_ReturnType, Csm_ConfigIdType, uint8*, uint32*, boolean) :void

Figure 2.23: Example ClientServerInterface BSW mapping diagamm (CSI bsw mapping diagram)

[TR_BSWMG_00116] MoS ClientServerInterface ⌈ For each Client Server Interface a class diagram (CSI mapping diagram) shall be created. The name of the diagram shall be the name of the Client Server Interface concatenated with "_BSWMapping". ⌋*()*

[TR_BSWMG_00117] MoS ClientServerInterface ⌈ A CSI errors diagram shall contain the ClientServerOperations of the ClientServerInterface and the corresponding bsw api interfaces. ⌋*()*

2.3.9.2 Modeling of Mode Switch Interfaces

The following listing shows the old syntax of modeling / defining ModeSwitchInterfaces in AUTOSAR R4.0.3 SWS documents.

```
ModeSwitchInterface WdgM_IndividualMode {
        isService = true;
        WdgMMode currentMode;
};
```

Corresponding ModeDeclarationGroup:

```
ModeDeclarationGroup WdgMMode {
        { SUPERVISION_OK,
          SUPERVISION_FAILED,
          SUPERVISION_EXPIRED,
          SUPERVISION_STOPPED,
          SUPERVISION_DEACTIVATED
        }
        initialMode = SUPERVISION_OK
};
```

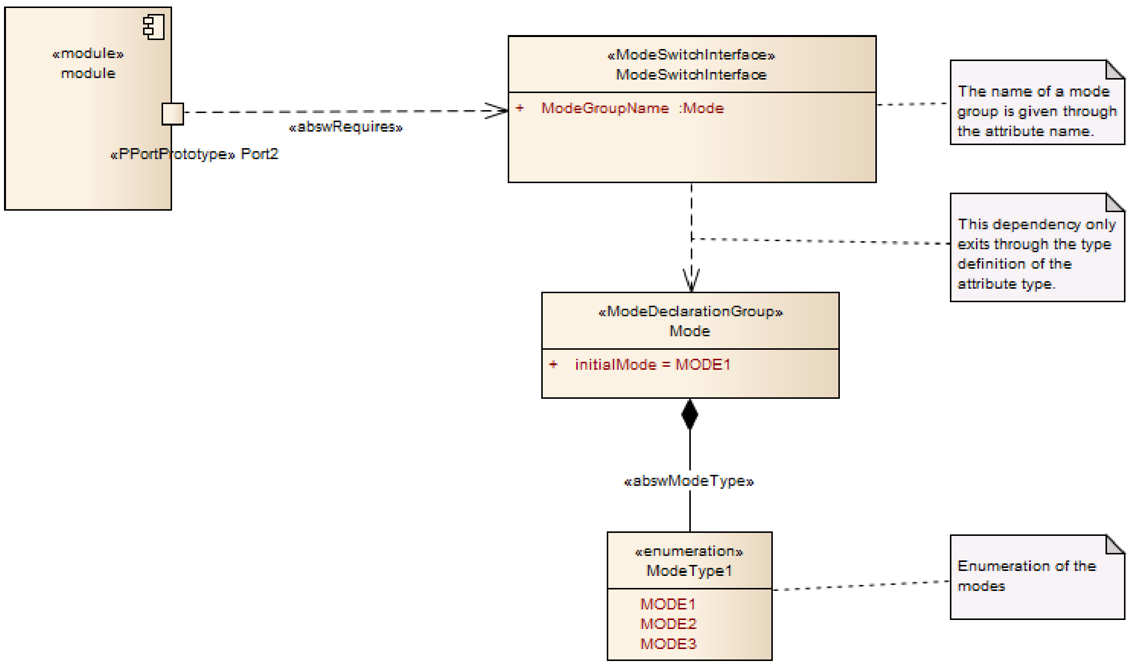


Figure 2.24: Schematic overview of a Mode Switch Interfaces

[TR_BSWMG_00203] MoS ModeDeclarationGroup ⌈ For each ModeDeclarationGroup a class of stereotype «ModeDeclarationGroup» shall be created. ⌋*()*

[TR_BSWMG_00209] MoS ModeDeclarationGroup initialMode ⌈ The initial mode of the ModeDeclarationGroup shall be modeled as a public attribute of the ModeDeclarationGroup class. The attribute's name shall be "initialMode", its initial value shall be set to one of the ModeDeclarationGroup's defined modes. ⌋*()*

[TR_BSWMG_00210] MoS ModeDeclarationGroup onTransitionValue ⌈ A ModeDeclarationGroup's optional "onTransitionValue" shall be modeled as a public attribute of the ModeDeclarationGroup class. The attribute's name shall be "onTransitionValue", its initial value shall be set to a positive integer. ⌋*()*

[TR_BSWMG_00204] MoS ModeDeclarationGroup Mode declarations ⌈ The modes of a ModeDeclarationGroup (e.g. SUPERVISION_OK, SUPERVISION_FAILED, ...) shall be modeled as a UML class of stereotype «ModeDeclaration». Each mode shall be the name of a public attribute. ⌋*()*

[TR_BSWMG_00211] MoS ModeDeclarationGroup Mode declaration integers ⌈ It is possible to assign concrete integer values to ModeDeclarations. In this case, the mode attribute's initial value shall be set to a positive integer. ⌋*()*

[TR_BSWMG_00212] MoS ModeDeclarationGroup category ⌈ The category of the ModeDeclarationGroup shall be inferred from the existing information in the following way:

- EXPLICIT_ORDER if all of its associated ModeDeclaration attributes have a numerical value assigned to them.
- ALPHABETIC_ORDER otherwise.

⌋*()*

[TR_BSWMG_00205] MoS ModeSwitchInterface ⌈ The relationship between ModeDeclarationGroup and the enumeration of modes shall be modeled as aggregation of stereotype «abswModeType» (target enumeration). ⌋*()*

[TR_BSWMG_00206] MoS ModeSwitchInterface ⌈ The ModeSwitchInterface class shall be containing a public attribute with a reference name to the current ModeDeclarationGroup (e.g. currentMode). The type of the attribute shall be the ModeDeclarationGroup. ⌋*()*

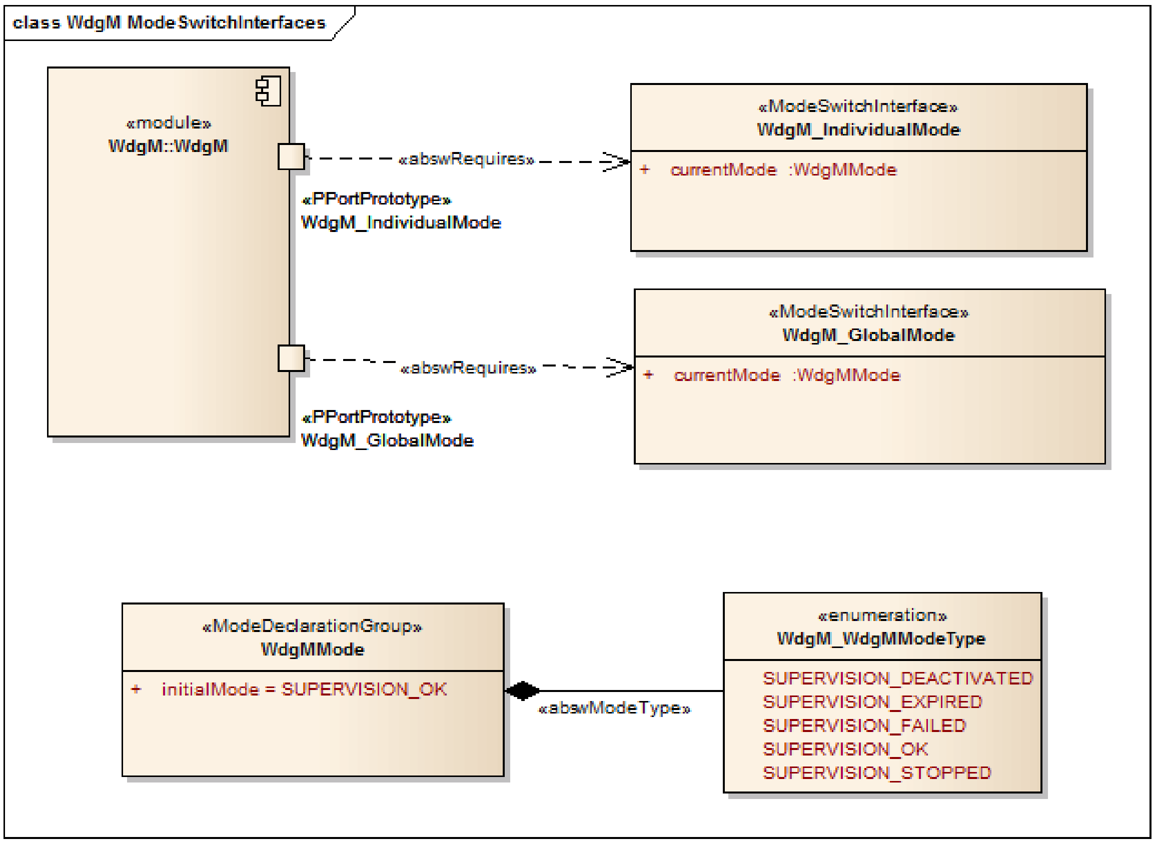


Figure 2.25: Example of a Mode Switch Interface

[TR_BSWMG_00207] MoS ClientServerInterface ⌈ For each Mode Switch Interface a class diagram shall be created. The name of the diagramm shall be the name of the Mode Switch Interface. ⌋*()*

[TR_BSWMG_00208] MoS ClientServerInterface ⌈ A Mode Switch Interface diagram shall contain the module, the ModeSwitchInterface, the ModeDeclarationGroup and the enumeration of the modes. ⌋*()*

2.3.9.3 Modeling of Sender Receiver Interfaces

The following listing shows the old syntax of modeling / defining SenderReceiverInterface in AUTOSAR R4.0.3 SWS documents.

```
SenderReceiverInterface AppModeRequestInterface {
  isService = true;
  AppModeRequestType requestedMode;
};
```

Corresponding Type:

```
ImplementationDataType AppModeRequestType {
  lowerLimit = 0;
  upperLimit = 2;
};
```

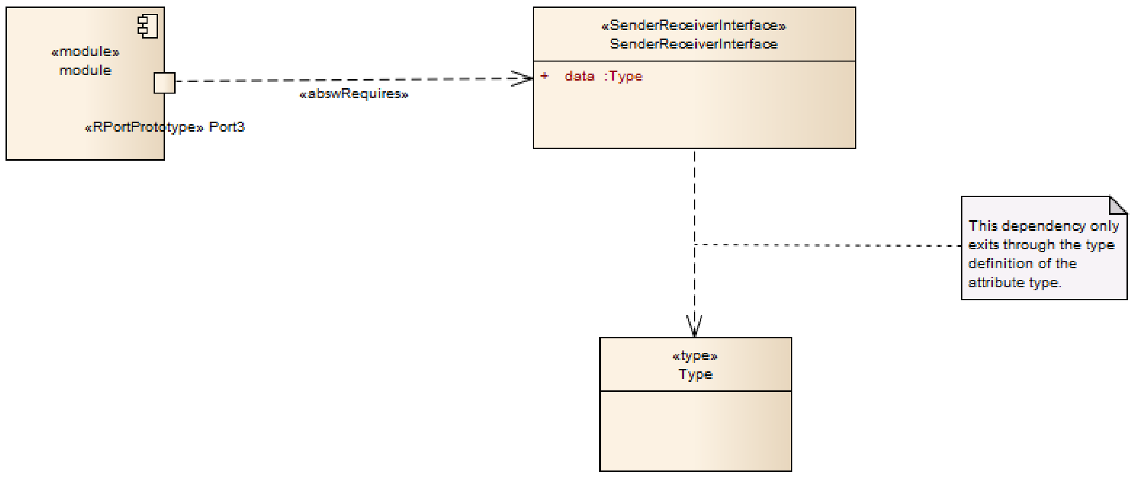

«module»
module
«RPortPrototype» Port3
«abswRequires»
«SenderReceiverInterface»
SenderReceiverInterface
+ data :Type
This dependency only exits through the type definition of the attribute type.
«type»
Type

Figure 2.26: Schematic overview of a Sender Receiver Interfaces

[TR_BSWMG_00301] MoS SenderReceiverInterface ⌈ Type of the sending/receiving data shall be modeled as a bsw api type or as a MoS type, see chapter 2.3.8. ⌋ *()*

[TR_BSWMG_00302] MoS SenderReceiverInterface ⌈ The SenderReceiverInterface class shall contain a public attribute with a reference name to the current sending/receiving Type (e.g. data). The type of the attribute shall be a valid type, see TR_BSWMG_00301. ⌋ *()*

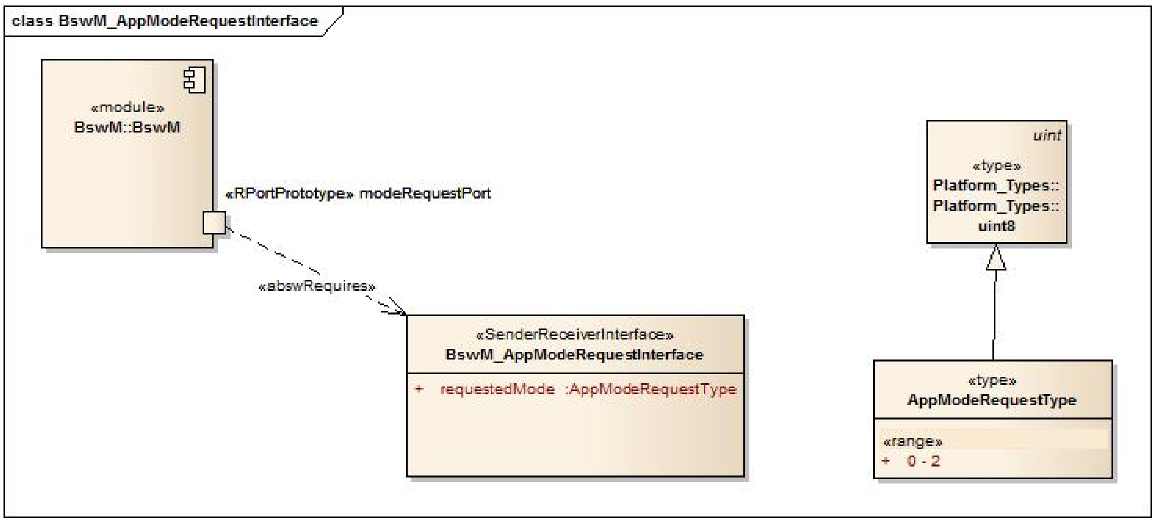

class BswM_AppModeRequestInterface
«module»
BswM::BswM
«RPortPrototype» modeRequestPort
«abswRequires»
«SenderReceiverInterface»
BswM_AppModeRequestInterface
+ requestedMode :AppModeRequestType
uint
«type»
Platform_Types::
Platform_Types::
uint8
«type»
AppModeRequestType
«range»
+ 0 - 2

Figure 2.27: Example of a Sender Receiver Interface

[TR_BSWMG_00307] MoS SenderReceiverInterface ⌈ For each Sender Receiver Interface a class diagram shall be created. The name of the diagram shall be the name of the Mode Switch Interface. ⌋ *()*

[TR_BSWMG_0308] MoS SenderReceiverInterface ⌈ A Sender Receiver Interface diagram shall contain the module, the SenderReceiverInterface and the Type of the sending/receiving data. ⌋*()*

2.3.9.4 Modeling of special Types in Service Interfaces

The following listing shows examples of type definitions in the old syntax of modeling / defining types in AUTOSAR R4.0.3 SWS documents.

Definition of arrays on arguments of ClientServerOperations:

```
ClientServerInterface Dcm_RequestControlServices
{
PossibleErrors {
  E_NOT_OK = 1,
  };
RequestControl(
  OUT uint8 OutBuffer[<DcmDspRequestControlOutBufferSize>],
  IN uint8 InBuffer[<DcmDspRequestControlInBufferSize>],
  ERR{E_NOT_OK });
}
```

Definition of pointer types:

```
//The data type DataPtr refers to an address and is defined as follows:
uint32* DataLengthPtr;
```

Definition of DataConstraints for simple types:

```
ImplementationDataType Dem_DTCStatusMaskType {
   LOWER-LIMIT = 0;
   UPPER-LIMIT = 255;
}
```

[TR_BSWMG_00400] Mos Types ⌈ Valid stereotypes of types are «type», «array», «pointer», «structure» and «eumeration». ⌋*()*

[TR_BSWMG_00401] MoS Simple Types ⌈ A simple type shall be modeled as described in 2.3.8.1. ⌋*()*

[TR_BSWMG_00403] MoS Array Types ⌈ An array type shall be modeled as a class of stereotype «array». To define the type of the array elements, a generalization relationship to the type shall be created. ⌋*()*

[TR_BSWMG_00409] MoS Array Types ⌈ The array size shall optionally be specified using the tagged value "Vh.ArraySize". ⌋*()*

[TR_BSWMG_00404] MoS Pointer Types ⌈ A pointer type shall be modeled as a class of stereotype «pointer». To define the type of the referenced data, a generalization relationship to the type shall be created. ⌋*()*

[TR_BSWMG_00405] MoS Structure Types ⌈ A structure type shall be modeled as described in 2.3.8.4. ⌋*()*

[TR_BSWMG_00407] MoS Enumeration Types ⌈ An enumeration type shall be modeled as described in 2.3.8.2. ⌋*()*

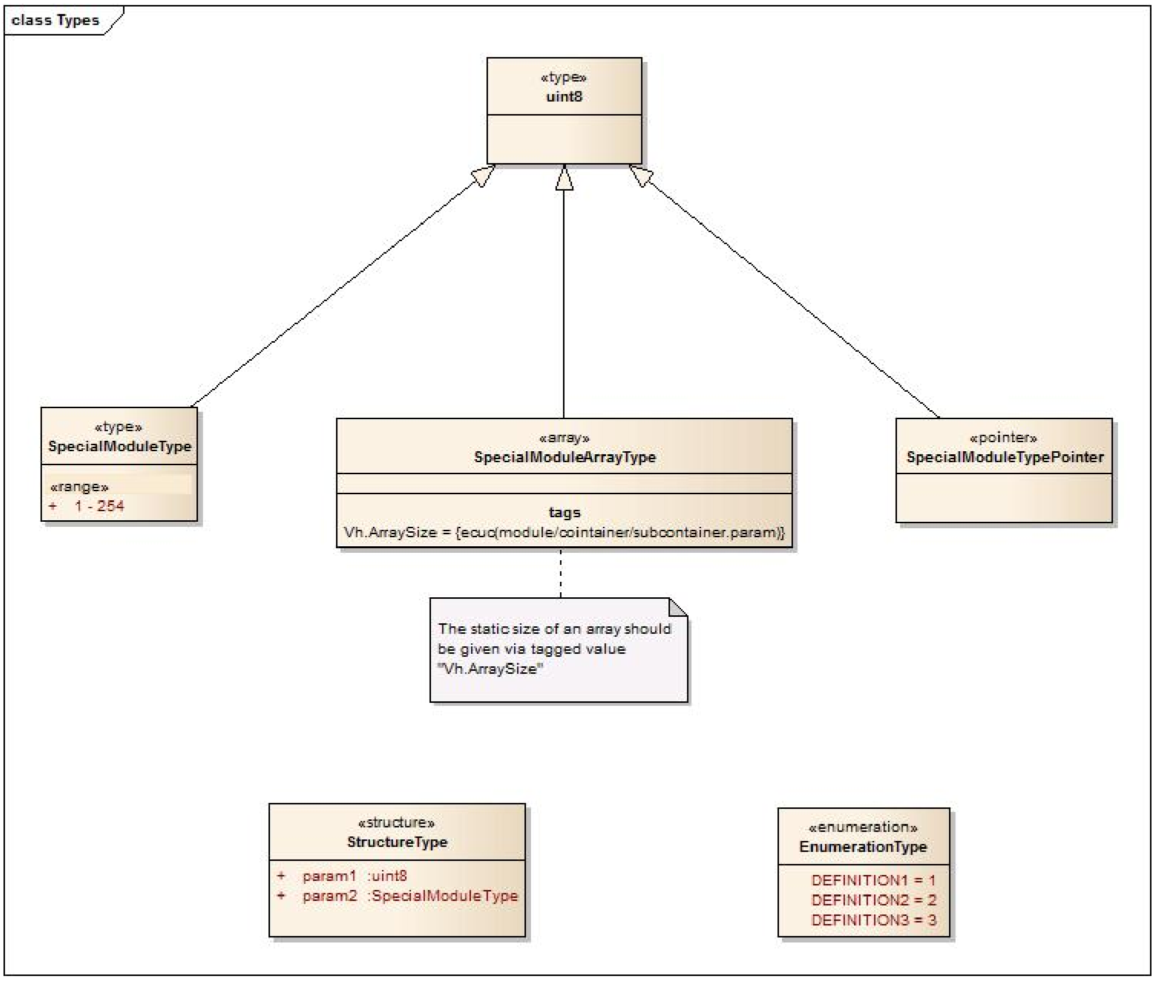


Figure 2.28: Schematic overview of type definitions

2.3.9.5 Modeling of variability of service interfaces

Many service interfaces are configurable since they depend on the configuration of the basis software. Therefore so called “blueprint conditions” have been introduced into BSW model to express e.g. that the existence of ports depends on the existence of specific EcuC parameters.

2.3.9.5.1 Examples of defining of variability in AUTOSAR R4.0.3 SWS documents

The following listing shows examples of the old syntax of modeling / defining of variability in AUTOSAR R4.0.3 SWS documents. The variability was defined informal as comments.

Variability in Ports:

```
Service ComM
{
...
        // port present for each channel
        // if ComMModeLimitationEnabled (see ECUC_ComM_00560);
        // there are NC channels;
        ProvidePort ComM_ChannelLimitation CL000;
        ...
        ProvidePort ComM_ChannelLimitation CL<NC-1>;
...
}
```

Variability in provided client server operations:

```
ClientServerInterface Dcm_SecurityAccess
{
...
//Request to application for synchronous comparing key
//(DcmDspSecurityUsePort = USE_SYNCH_CLIENT_SERVER)
CompareKey(IN uint8 Key[<DcmDspSecurityKeySize>],
           ERR{E_NOT_OK, E_COMPARE_KEY_FAILED});

//Request to application for asynchronous comparing key
//(DcmDspSecurityUsePort = USE_ASYNCH_CLIENT_SERVER)
CompareKey(IN uint8 Key[<DcmDspSecurityKeySize>],
           IN Dcm_OpStatusType OpStatus,
           ERR{E_NOT_OK, E_PENDING, E_COMPARE_KEY_FAILED});
}
```

Variability in provided client server operations parameters and types:

```
// ProtInterface type and name
ClientServerInterface Dcm_RoutineServices {
 ...
  // <datatype> dataIn1,..., defines multiple parameters of
        //   a parameterized type
  // uint8* dataInN for the last parameter is a concrete
        //   type defined (not parameterized)

        StartFlex(
              IN <datatype> dataIn1,..., IN uint8 dataInN[( <
                 DcmDspRoutineSignalLength of
                 DcmDspStartRoutineInSignal> +7)/8],
              IN Dcm_OpStatusType OpStatus,
              OUT <datatype> dataOut1,..., OUT uint8 dataOutN[( <
                 DcmDspRoutineSignalLength of
                 DcmDspStartRoutineOutSignal> +7)/8],
              INOUT uint16 currentDataLength,
              OUT Dcm_NegativeResponseCodeType ErrorCode,
              ERR{E_NOT_OK, DCM_E_PENDING, E_FORCE_RCRRP });
...
};
```

Variability in provided interface type:

```
ClientServerInterface DataServices:

Using the concepts of the SW-C template, the interface is defined as
    follows if ClientServer interface is used (DcmDspDataUsePort set to
     USE_DATA_SYNCH_CLIENT_SERVER or USE_DATA_ASYNCH_CLIENT_SERVER):
```

```
SenderReceiver DataServices:

Using the concepts of the SW-C template, the interface is defined as
    follows if SenderReceiver interface is used (DcmDspDataUsePort set
    to USE_DATA_SENDER_RECEIVER):
```

2.3.9.5.2 Modeling of variability in BSW UML model

[TR_BSWMG_00500] MoS Variability NamePattern (number of occurrences) ⌈ If the number of occurrences of e.g. a port is depending on a the occurrences of EcuC containers, the condition shall defined in tagged value "Vh.NamePattern.BlueprintPolicy.DerivationGuide" and the Namepattern shall be defined in tagged value "Vh.NamePattern". ⌋*()*

[TR_BSWMG_00501] MoS Variability (existence) ⌈ To define variability of e.g. a port the tagged value "Vh.BlueprintCondition" shall be used. ⌋*()*

[TR_BSWMG_00502] MoS Variability multiple conditions ⌈ To define multiple conditions on e.g. a port, '.' + number shall be append to the tagged value name e.g. "Vh.BlueprintCondition.1". ⌋*()*

Variability example of a port:

```
Vh.BlueprintCondition:
        {ecuc(ComM/ComMGeneral.ComMModeLimitationEnabled)} == true
Vh.NamePattern.BlueprintPolicy.DerivationGuide:
        Name = {ecuc(ComM/ComMConfigSet/ComMChannel)}
Vh.NamePattern:
        CL_{Name}
```

[TR_BSWMG_00507] MoS port interface reference is configurable ⌈ If the reference to a port interface is configurable by EcuC the tagged value "Vh.InterfaceRef.BlueprintPolicy.DerivationGuide" shall be used. ⌋*()*

Configurable interface reference of a port (BswM modeNotificationPort):

```
Vh.InterfaceRef.BlueprintPolicy.DerivationGuide:
        {ecuc(BswM/BswMConfig/BswMArbitration/BswMModeRequestPort/
           BswMModeRequestSource/BswMSwcModeNotification.
           BswMSwcModeNotificationModeDeclarationGroupPrototypeRef)}.
           parent
```

[TR_BSWMG_00155] MoS Port Interface configurable isService attribute ⌈ If the value of the isService attribute depends on a EcuC parameter the tagged value "Vh.isService.BlueprintPolicy.DerivationGuide" shall be used. The value of the tagged value shall be set to the blueprint condition referencing the EcuC parameter. ⌋*()*

2.3.9.6 Modeling of PortAPIOptions and PortDefinedArgumentValues

[TR_BSWMG_00118] MoS PortAPIOption ⌈ A PortAPIOption shall be modeled as an UML class of stereotype «PortAPIOption». The class shall be placed in the package <module>/ARInterfaces/<affected interfaces>. ⌋*()*

[TR_BSWMG_00119] MoS PortAPIOption Name ⌈ The name of the PortAPIOption class shall be composed of the port name followed by underscore followed by literal string “PortAPIOption”, e.g. “Func_PortAPIOption” for a port named “Func”. ⌋*()*

[TR_BSWMG_00120] MoS PortAPIOption reference to Port ⌈ The «PortAPIOption» class shall reference its affected port using a dependency with stereotype «abswPortRef». ⌋*()*

[TR_BSWMG_00121] MoS PortDefinedArgumentValue ⌈ Port Defined Argument Values shall be modeled as attributes of a «PortAPIOption» class. ⌋*()*

[TR_BSWMG_00122] MoS PortDefinedArgumentValue Stereotype ⌈ The attribute representing the Port Defined Argument Value shall have the stereotype «PDAV». ⌋*()*

[TR_BSWMG_00123] MoS PortDefinedArgumentValue Order ⌈ If a port uses more than one Port Defined Argument Values, the attribute order within the «PortAPIOption» class shall reflect the argument order in the BSW functions associated with the port’s provided ClientServerInterface operations. ⌋*()*

[TR_BSWMG_00124] MoS PortDefinedArgumentValue Name ⌈ The Port Defined Argument Value attribute’s ‘Name’ field shall match the corresponding BSW-functions’ parameter name. ⌋*()*

[TR_BSWMG_00125] MoS PortDefinedArgumentValue fixed Type (non-configurable) ⌈ If the Port Defined Argument Value is of a fixed type, i.e. it is not configurable by an EcuC parameter, the attribute’s ‘Type’ field shall reference a valid type that is either modeled as a BSW API type or as an MoS type. ⌋*()*

[TR_BSWMG_00126] MoS PortDefinedArgumentValue configurable Type ⌈ If the Port Defined Argument Value’s type is configurable by an EcuC parameter, the ‘Type’ field shall be set to the literal string {DataType}. The curly braces indicate that “DataType” is treated as a place holder for the EcuC-configured type rather than a valid data type itself. ⌋*()*

[TR_BSWMG_00127] MoS PortDefinedArgumentValue Type Configuration by EcuC ⌈ If the Port Defined Argument Value’s type is configurable by an EcuC parameter, the EcuC configuration dependency shall be expressed by a tagged value attached to the attribute: Tag “TypeRef”, Value: “DataType = {ecuc(some/ecuc/param/dataTypeRef)}” ⌋*()*

[TR_BSWMG_00128] MoS PortDefinedArgumentValue Value Configuration by EcuC ⌈ If the Port Defined Argument Value’s “value” is configurable by an EcuC parameter, the EcuC configuration dependency shall be expressed by a tagged value attached to the attribute: Tag: “Vh.Value.BlueprintPolicy.DerivationGuide”, Value: “{ecuc(some/ecuc/param/value)}”. ⌋*()*

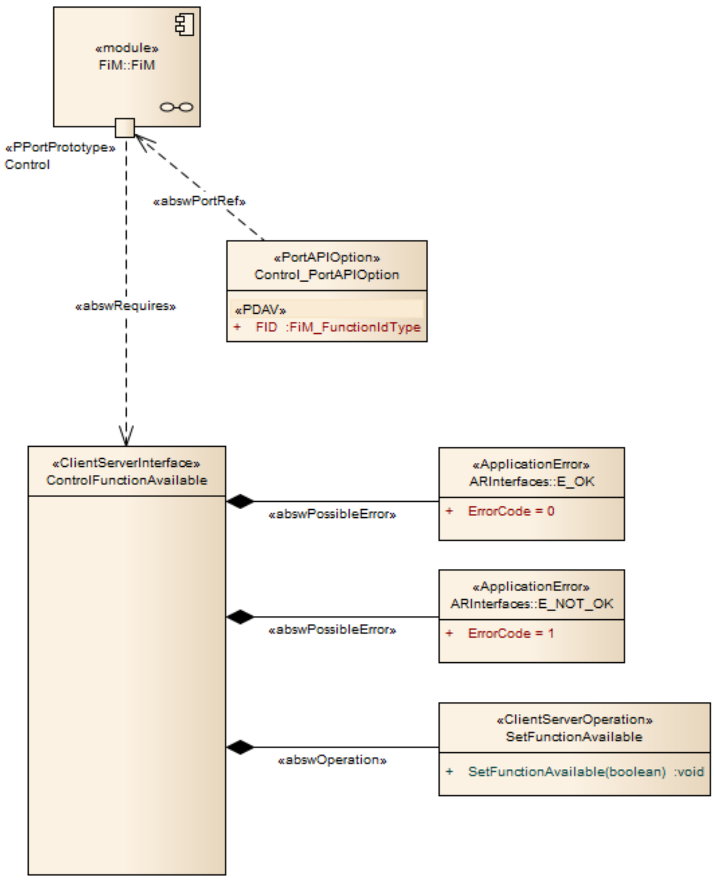


Figure 2.29: PortAPIOption example for module Fim ClientServerInterface ControlFunctionAvailable

2.4 Diagrams

2.4.1 Header File Modeling

[TR_BSWMG_00600] Header File Diagram ⌈ The module package shall contain a *header file diagram* (Enterprise Architect: UML Component Diagram). ⌋*()*

[TR_BSWMG_00601] Naming of Header File Diagram ⌈ The name of the header file diagram shall be the name of the *module component* followed by `_header`, e.g. `FrTp_header`. ⌋*()*

[TR_BSWMG_00602] Header and Source Code Artifacts ⌈ Document artifacts shall be declared either as header or source file using the stereotypes «header» and «source». ⌋*()*

[TR_BSWMG_00603] Document Artifact Location ⌈ Document artifacts shall be placed in the module package of the defining BSW module. ⌋*()*

[TR_BSWMG_00604] Include Dependency ⌈ Document artifacts can include other artifacts using a dependency with stereotype «include». With the additional stereotype «optional», optional inclusion can be expressed. ⌋*()*

[TR_BSWMG_00605] Optional Include Dependency ⌈ Optional inclusion can be expressed by specifying the additional stereotype «optional» on an include dependency. ⌋*()*

2.4.2 Sequence Diagrams

[TR_BSWMG_00901] Usage of Sequence Diagrams ⌈ For modeling interactions of different modules, sequence diagrams shall be used. ⌋*()*

[TR_BSWMG_00902] Location of Sequence Diagrams ⌈ All sequence diagrams shall be placed within the "Interaction Views Package" package. ⌋*()*

2.4.3 State Machine Diagrams

[TR_BSWMG_00801] Usage of State Machine Diagrams ⌈ For modeling state dependencies within and between elements, state machine diagrams shall be used. ⌋*()*

2.5 Support for Life Cycle concept in BSW Model

AUTOSAR introduced the possibility to attach life-cycle-related information to all (`Referable`) specification elements with the Life Cycle Concept in R4.1.1. In a nutshell, a LifeCycleInfo element can be created for a specification element to document its life cycle state - see [5], chapter 11.3.2.

[TR_BSWMG_00700] Model elements that support life cycle information ⌈ In the BSW model the following modeling elements shall be able to have life cycle information:

- Type definitions
- API Functions
- Services

⌋*()*

[TR_BSWMG_00701] LifeCycleInfo information in model elements ⌈ Life cycle information is represented by tagged values on the model element. ⌋*()*

[TR_BSWMG_00702] Valid tagged values for life cycle information on model elements ⌈ The following tagged values can be used to document life cycle information:

atp.Status A value from the official WP-M lifecycle definitions [6]

atp.StatusComment (optional) Explanatory comment

atp.StatusRevisionBegin Beginning of applicability of LifeCycleInfo

atp.StatusRevisionEnd (optional) End of applicability of LifeCycleInfo

atp.StatusUseInstead (optional) The element that replaces an “obsolete” or a “removed” model element

⌋*()*