Document Title	Specification of Language Binding for modeled AP data types
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	994

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R21-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2021-11-25	R21-11	AUTOSAR Release Management	• Initial release (previously part of [1])

Disclaimer

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

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

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

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

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction 5

1.1 Adaptive Platform Data Types 5
1.2 Language Bindings 6
1.3 Methodology 6

2 Abbreviations and Terms 7

3 Related documentation 9

3.1 Input documents & related standards and norms 9

4 Constraints and assumptions 9

4.1 Limitations 9

5 Dependencies to other modules 10

6 Requirements Tracing 11

7 Functional specification 12

7.1 C++ 12
7.1.1 ARA Language Binding Generator 12
7.1.1.1 CppImplementationDataTypes Header Files 13
7.1.1.2 Caveats 15
7.1.2 CppImplementationDataType 16
7.1.2.1 Sub-classes of CppImplementationDataType 17
7.1.2.1.1 StdCppImplementationDataType 18
7.1.2.1.2 CustomCppImplementationDataType 18
7.1.3 Primitive Data Type 20
7.1.3.1 Fixed Width Integer 21
7.1.4 String Data Type 21
7.1.4.1 No Allocator 21
7.1.4.2 Allocator 22
7.1.5 Array Data Type 22
7.1.5.1 StdCppImplementationDataType 23
7.1.5.1.1 One-dimensional 23
7.1.5.1.2 Multi-dimensional 24
7.1.5.2 CustomCppImplementationDataType 25
7.1.5.2.1 One-dimensional 25
7.1.5.2.2 Multi-dimensional 26
7.1.6 Vector Data Type 26
7.1.6.1 StdCppImplementationDataType 28
7.1.6.1.1 One-dimensional 28
7.1.6.1.2 Multi-dimensional 29
7.1.6.2 CustomCppImplementationDataType 30
7.1.6.2.1 One-dimensional 30
7.1.6.2.2 Multi-dimensional 31

7.1.7 Structure Data Type 31
7.1.7.1 StdCppImplementationDataType 31
7.1.7.2 Optional Elements 33
7.1.8 Enumeration Data Type 34
7.1.9 Associative Map Data Type 36
7.1.9.1 StdCppImplementationDataType 37
7.1.9.1.1 No Allocator 37
7.1.9.1.2 Allocator 38
7.1.9.2 CustomCppImplementationDataType 39
7.1.9.2.1 No Allocator 39
7.1.9.2.2 Allocator 40
7.1.10 Variant Data Type 40
7.1.10.1 StdCppImplementationDataType 41
7.1.10.2 CustomCppImplementationDataType 42
7.1.11 Redefinition of Implementation Data Type 42
7.1.12 Scale Linear And Texttable Data Type 43

8 API specification 44

A Mentioned Manifest Elements 45

B History of Specification Items 64

B.1 Specification Item evolution compared to AUTOSAR R20-11 64
B.2 Specification Item history of this document according to AUTOSAR R21-11 65
B.2.1 Added Traceables in R21-11 65
B.2.2 Changed Traceables in R21-11 67
B.2.3 Deleted Traceables in R21-11 67

1 Introduction

1.1 Adaptive Platform Data Types

The AUTOSAR data type model defined in [2] allows varying levels of granularity for specifying data types. The fundamentals of AUTOSAR data types are described in [3] chapter *"Data Types"* and further specialized for the Adaptive Platform (AP) in [4] chapter *"Data Type"*.

This specification is **not** concerned with ApplicationDataTypes, it is **only** concerned with concrete sub-classes of AbstractImplementationDataType - it is at this point in the data type model that the Language Binding is selected.

In general, the data types are used by typed sub-classes of PortInterface which model a particular function, e.g. ServiceInterface. Interface elements of these sub-classes of PortInterface may reference AutosarDataPrototypes, further typed by concrete sub-classes of AutosarDataTypes; specifically, as stated in [3] these are "Application" level and "Implementation" level data types.

Figure 1.1 shows on meta-model level the usage of AutosarDataPrototypes in Adaptive Platform Interfaces.

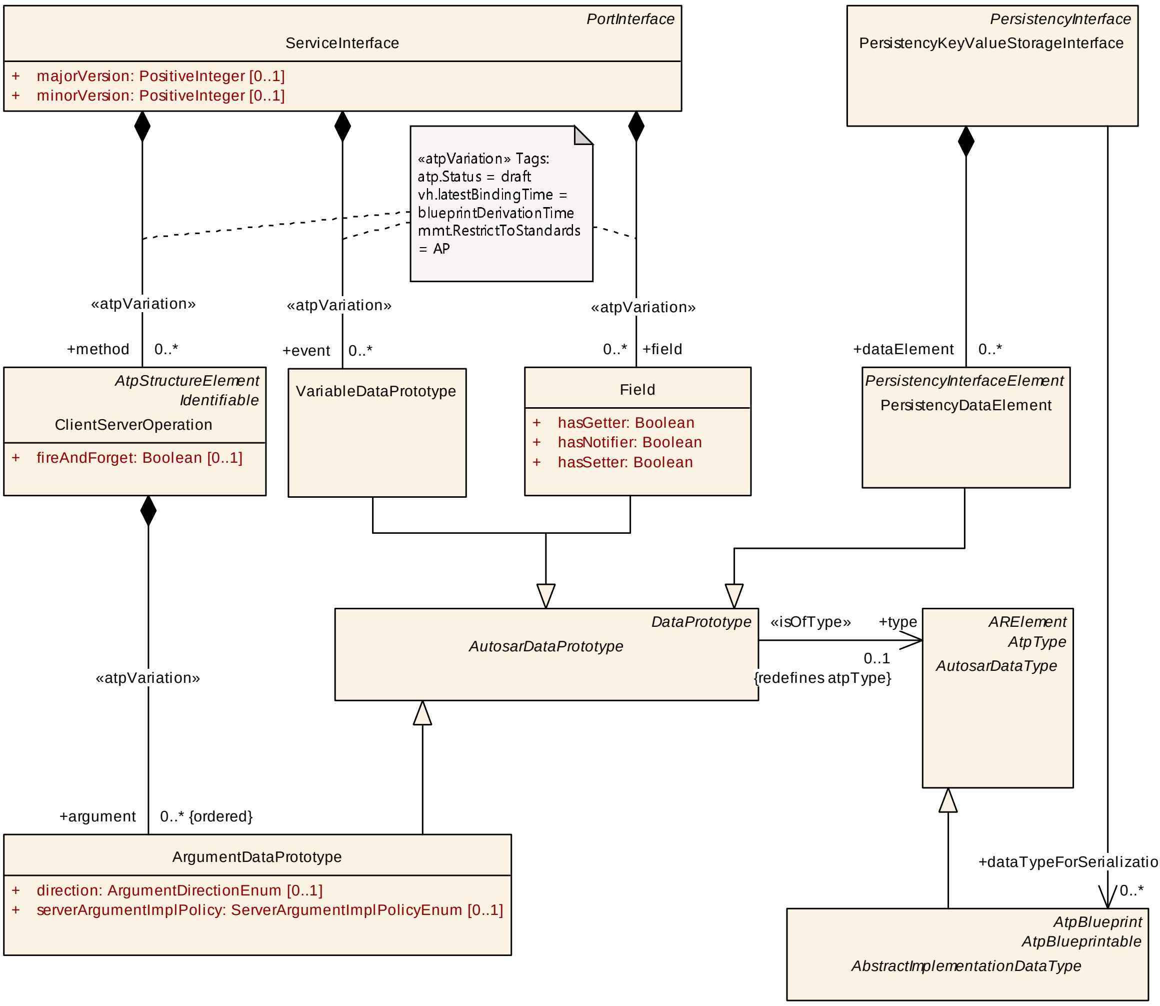

Figure 1.1: AUTOSAR data type usage in Adaptive Interfaces

1.2 Language Bindings

While the primary focus of the AP is targeted towards a C++ Language Binding (7.1), the chapter structure of the document allows for future versions to seamlessly insert "other" Language Bindings.

1.3 Methodology

This specification documents the generation/serialization[1] rules for transforming AP "modeled" Implementation Data Types to actual "language level" Data Types which can be processed by a compiler/interpreter of the bound language.

[1] the term "serialization" should not be mixed with (de-)serialization in the context of Communication

The general workflow step is described in *"Adaptive Software Generated Item"* in [5]; Figure 1.2 shows a very general workflow step for generation of data types from an Adaptive Platform Interface. Each "language specific" binding will have a "language specific" approach, and thus a respective chapter in this specification.

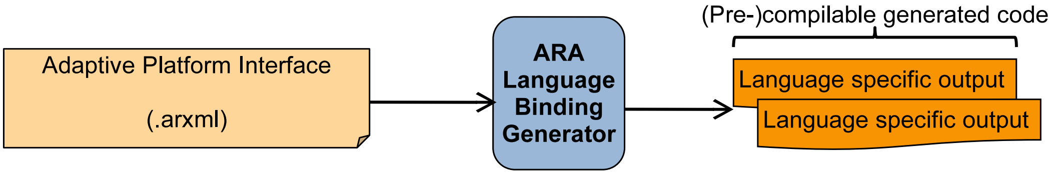
Adaptive Platform Interface
(.arxml)
ARA Language Binding Generator
(Pre-)compilable generated code
Language specific output
Language specific output

Figure 1.2: Methodology: Generic Language Binding generation

This specification is not concerned with the implementation details of an ARA Language Binding Generator, rather, the rules which an ARA Language Binding Generator must observe during generation/serialization.

[SWS_LBAP_00037]{DRAFT} **Principle of an ARA Language Binding Generator** ⌈The ARA Language Binding Generator is responsible for generating the Lanagauge Binding artifacts. These include data type declarations derived from the referenced AbstractImplementationDataTypes of the Adaptive Platform Interfaces.⌋*()*

2 Abbreviations and Terms

The main list of terms and abbreviations are defined in [6]. The following tables contain the list of terms and abbreviations used in the scope of this document which are not already defined in [6] along with the spelled-out meaning of each of the abbreviations.

Abbreviation	Meaning
LBAP	Language Binding for the Adaptive Platform

Table 2.1: Abbreviations used in the scope of this Document

Term	Meaning
Allocator	A language specific object responsible for (de-)allocation, (de-)initialization and ultimately limit impositions in memory/storage. C++ allocators must satisfy the requirements for an *Allocator* in ISO/IEC 14882 (version according to [RS_AP_00114]).

▽

△

Term	Meaning
ARA Language Binding Generator	A workflow tool (e.g. a script) with the purpose to read-/parse an ARXML model of data types in an Adaptive Platform Interface and generate a corresponding language specific representation thereof.
Adaptive Platform Interface	A typed (concrete) sub-class of PortInterface bound to the Adaptive Platform (in contrast to an "other" platform).
CppImplementation-Types Header File	A generated C++ header file created by an ARA Language Binding Generator.
C++ Bound Interface	An Adaptive Platform Interface which transitively references a CppImplementationDataType in it's usage (in contrast to an "other" language binding).
C++ Compound Type	See chapter *"Compound types"* in ISO/IEC 14882 (version according to [RS_AP_00114]).
C++ Fundamental Type	See chapter *"Fundamental types"* in ISO/IEC 14882 (version according to [RS_AP_00114]).
C++ Language Binding	A Language Binding in which the modeled representation is a CppImplementationDataType and the implementation language is C++.
Comparator	A language specific Functor responsible for binary comparison.
Functor	A language specific object which is treated as callable or executable. In C++ this is wrapped in std::function - ISO/IEC 14882 (version according to [RS_AP_00114])
Language Binding	A language binding is the point in which a representation on one side is selected (or bound) to a specific programming language on another side. In the context of this document a modeled representation is bound to a implementation language

Table 2.2: Terms used in the scope of this Document

3 Related documentation

3.1 Input documents & related standards and norms

[1] Specification of Communication Management
AUTOSAR_SWS_CommunicationManagement

[2] Meta Model
AUTOSAR_MMOD_MetaModel

[3] Software Component Template
AUTOSAR_TPS_SoftwareComponentTemplate

[4] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

[5] Methodology for Adaptive Platform
AUTOSAR_TR_AdaptiveMethodology

[6] Glossary
AUTOSAR_TR_Glossary

[7] Specification of Adaptive Platform Core
AUTOSAR_SWS_AdaptivePlatformCore

[8] Specification of Platform Types for Adaptive Platform
AUTOSAR_SWS_AdaptivePlatformTypes

[9] Requirements on Communication Management
AUTOSAR_RS_CommunicationManagement

[10] General Requirements specific to Adaptive Platform
AUTOSAR_RS_General

[11] Main Requirements
AUTOSAR_RS_Main

4 Constraints and assumptions

4.1 Limitations

- Although future versions of this specification may add further Language Bindings [RS_AP_00513], the primary focus of the AP (and therefore this specification) is a binding to the C++ language.

5 Dependencies to other modules

The LBAP is not an AUTOSAR Functional Cluster (FC) and therefore has no dependencies to other FCs.

This following software/template specifications serve as input documents to this specification:

- [4]: Specifies the Modeled Adaptive Platform data types - for any given modeled Adaptive Platform data type, there shall be a corresponding language binding
- [7]: Language binding for Adaptive Platform Core data types - depending on model configurations, generated Language Bindings may utilize ARA core types
- [8]: Language binding for Adaptive Platform Primitive data types - depending on model configurations, generated Language Bindings may utilize platform types

6 Requirements Tracing

The following tables reference requirements specified in [9], [10], [11] and links to the fulfillment of these. Please note that if column "Satisfied by" is empty for a specific requirement, this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[RS_AP_00114]	C++ interface shall be compatible with C++14.	[SWS_LBAP_00005] [SWS_LBAP_00006] [SWS_LBAP_00007] [SWS_LBAP_00008] [SWS_LBAP_00009] [SWS_LBAP_00010] [SWS_LBAP_00011] [SWS_LBAP_00012] [SWS_LBAP_00013] [SWS_LBAP_00014] [SWS_LBAP_00015] [SWS_LBAP_00016] [SWS_LBAP_00017] [SWS_LBAP_00018] [SWS_LBAP_00022] [SWS_LBAP_00023] [SWS_LBAP_00024] [SWS_LBAP_00025] [SWS_LBAP_00026] [SWS_LBAP_00027] [SWS_LBAP_00028] [SWS_LBAP_00031] [SWS_LBAP_00038]
[RS_AP_00122]	Type names.	[SWS_LBAP_00005]
[RS_AP_00127]	Usage of ara::core types.	[SWS_LBAP_00007] [SWS_LBAP_00012] [SWS_LBAP_00013] [SWS_LBAP_00015] [SWS_LBAP_00016] [SWS_LBAP_00017] [SWS_LBAP_00018] [SWS_LBAP_00023] [SWS_LBAP_00024]

7 Functional specification

The LBAP is not an ARA Functional Cluster (FC) and therefore has no functional specification. Rather, in the following sub-chapters the serialization/binding rules are laid out how the data types in the AUTOSAR meta-model are transformed to the respective language specific representation for use in ARA applications and FCs.

As explained in 1.1, AutosarDataTypes referenced by elements of any Adaptive Platform Interface, e.g.:

- ServiceInterface.event
- ServiceInterface.method
- ServiceInterface.field
- PersistencyKeyValueStorageInterface.dataElement

may be serialized/bound by a (generator/serializer) tool to an actual language bound compilable[1](or as near to as compilable as possible if they shall be further post-processed). The following sub-chapters specify the serialization rules for those Language Bindings supported by AUTOSAR.

7.1 C++

This section describes the overall methodology and principles of the ARA Language Binding Generator for a binding to the C++ language; specifically, the version stated in [RS_AP_00114] specifies the C++ standards version for the AP.

In the context of this specification, any reference to C++ language level aspects, pertain to the ISO C++ standards version given by [RS_AP_00114].

7.1.1 ARA Language Binding Generator

Figure 7.1 shows the workflow steps for code generation for a C++ Language Binding, other languages may have other workflows.

This is a more detailed pictorial view of the high-level AP workflow step *"Adaptive Software Generated Item"* in [5] and thus the Language Binding generation would typically be done together with the *other* generations in the context of this workflow step.

[1] the term "compilable" is used generically here (use the term "interpretable" if the Language Binding implies an interpreter instead of a compiler)

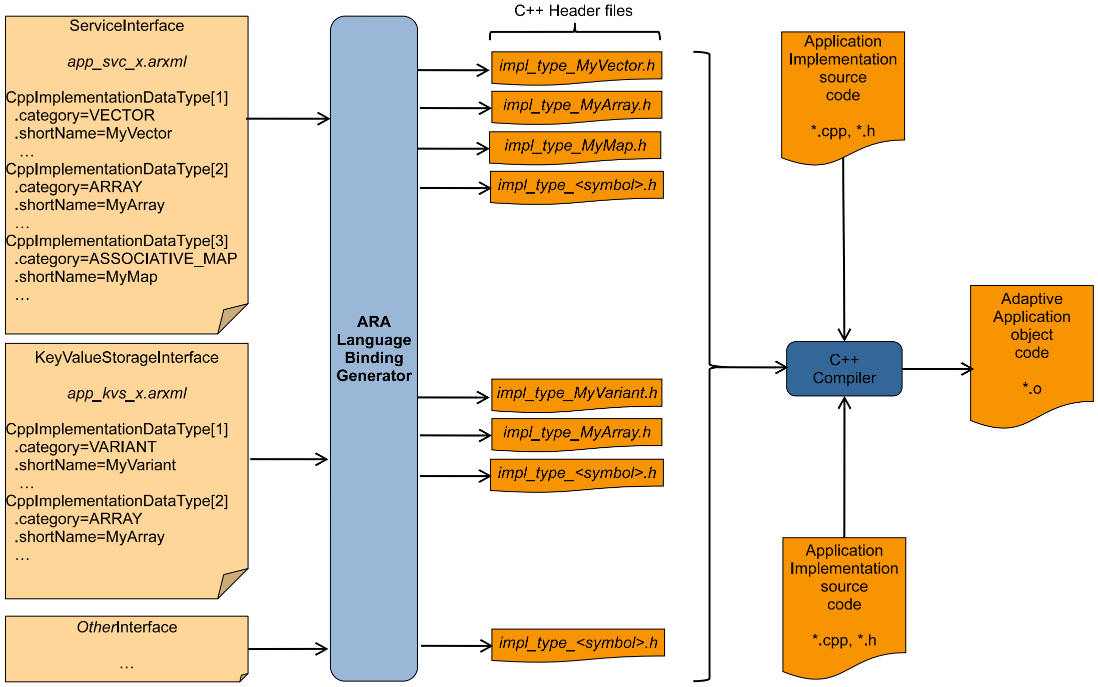

Figure 7.1: Methodology: C++ Language Binding generation

7.1.1.1 CppImplementationDataTypes Header Files

The attribute typeEmitter has an immediate direct influence on the behavior of the ARA Language Binding Generator i.e. whether generation shall take place or not.

[SWS_LBAP_00002]{DRAFT} **ARA Language Binding Generator usage of typeEmitter** ⌈The ARA Language Binding Generator shall generate a corresponding C++ Language Binding according to the rules defined in [TPS_MANI_01176], [TPS_MANI_01177] and [TPS_MANI_01212].⌋*()*

[SWS_LBAP_00004]{DRAFT} **Naming of data types by shortName** ⌈The Cpp Implementation Data Type symbol shall be the shortName of the CppImplementationDataType.⌋*()*

[SWS_LBAP_00032]{DRAFT} **CppImplementationTypes Header Files artifact generation** ⌈The ARA Language Binding Generator shall generate a discrete C++ header (.h) file with C++ type declaration for each CppImplementationDataType defined in an Adaptive Platform Interface.⌋*()*

Note: [SWS_LBAP_00032] obviously makes sense for C++ Compound Types, but it is accepted that this rule may be relaxed for simple types which resolve to C++ Fundamental Types, i.e. it makes less sense to create an own C++ header (.h) for a simple using declaration.

[SWS_LBAP_00033]{DRAFT} **CppImplementationTypes Header Files file names** ⌈The ARA Language Binding Generator shall construct the file name of each CppImplementationTypes Header File according to the format: impl_type_<symbol>.h where:

symbol : is the CppImplementationDataType.shortName converted to lower-case.

⌋()

[SWS_LBAP_00034]{DRAFT} **CppImplementationTypes Header Files directory names** ⌈The ARA Language Binding Generator shall construct the directory hierarchy of those CppImplementationTypes Header Files in [SWS_LBAP_00033] according to the format:

```
<namespace[0]><sep> ... <namespace[n]><sep><filename>

```

where:

namespace : are the namespace names as defined in [SWS_LBAP_00035]

sep : is the platform specific directory path separator, e.g. "/"

filename : is the file name according to [SWS_LBAP_00033]

example:

```
path/to/myfc/...

```

⌋()

[SWS_LBAP_00035]{DRAFT} **CppImplementationTypes Header Files namespace hierarchy** ⌈The ARA Language Binding Generator shall use the SymbolProps aggregated in the role CppImplementationDataType.namespace [TPS_MANI_01168], to construct the encapsulating C++ namespace hierarchy for the C++ data type inside the CppImplementationTypes Header File according to the format:

```
namespace <CppImplementationDataType.namespace_0.symbol>
{
namespace <CppImplementationDataType.namespace_i.symbol>
{
namespace <CppImplementationDataType.namespace_N.symbol>
{
    ...
} // namespace <CppImplementationDataType.namespace_N.symbol>
} // namespace <CppImplementationDataType.namespace_i.symbol>
} // namespace <CppImplementationDataType.namespace_0.symbol>

```

where:

CppImplementationDataType.namespace_0.symbol : is the first CppImplementationDataType.namespace in the ordered list, converted to lower-case.

CppImplementationDataType.namespace_i.symbol : are the intermediate CppImplementationDataType.namespaces in the ordered list, converted to lower-case.

CppImplementationDataType.namespace_N.symbol : is the last CppImplementationDataType.namespace in the ordered list, converted to lower-case.

example:

```
namespace mydomain
{
namespace myfc
{
    ...
} // namespace myfc
} // namespace mydomain

```

⌋*()*

[SWS_LBAP_00036]{DRAFT} **CppImplementationTypes Header Files multiple inclusion guard** ⌈The ARA Language Binding Generator shall generate a multiple inclusion guard around the whole header file in each CppImplementationTypes Header File according to the format:

```
#ifndef <path>_H_
#define <path>_H_
    ...
#endif // <path>_H_

```

where:

path : is the relative path of the header file according to [SWS_LBAP_00034] up to but omitting the file extension, with all path components separated by an underscore ("_"), converted to upper-case.

example:

```
#ifndef PATH_TO_MYFC_H_
#define PATH_TO_MYFC_H_
    ...
#endif // PATH_TO_MYFC_H_

```

See also [SWS_CORE_90002].⌋*()*

7.1.1.2 Caveats

An AP model may define AutosarDataPrototypes which can be typed by ApplicationDataTypes and/or by CppImplementationDataTypes.

Therefore it is required in the input configuration that every ApplicationDataType used for the typing of a DataPrototype is mapped by a DataTypeMap to an CppImplementationDataType.

The PortInterfaceToDataTypeMapping associates a particular PortInterface with a DataTypeMappingSet and defines thus the applicable DataTypeMaps.

[SWS_LBAP_00001]{DRAFT} **ARA generator rejection of unmapped data types** ⌈The ARA Language Binding Generator shall treat model configurations containing a AutosarDataPrototype which is typed by an ApplicationDataType but not mapped to an CppImplementationDataType as an error.⌋*()*

[SWS_LBAP_00003]{DRAFT} **ARA generator rejection of symbol clashes** ⌈The ARA Language Binding Generator shall treat a potential symbol clash in a generated Language Binding as an error.⌋*()*

A symbol clash results from a generated Language Binding containing 1+ symbols in the same namespace with same symbol name.

7.1.2 CppImplementationDataType

The basis for the C++ Language Binding is the C++ data type representation in [4] chapter *"CppImplementationDataType"*. The CppImplementationDataType is the point in the AUTOSAR data type tree where the implementation of the data type becomes bound to the C++ language.

For the following sub-chapters, it is **essential** to have an understanding of the AUTOSAR data type model from the perspective of CppImplementationDataType shown here in Figure 7.2.

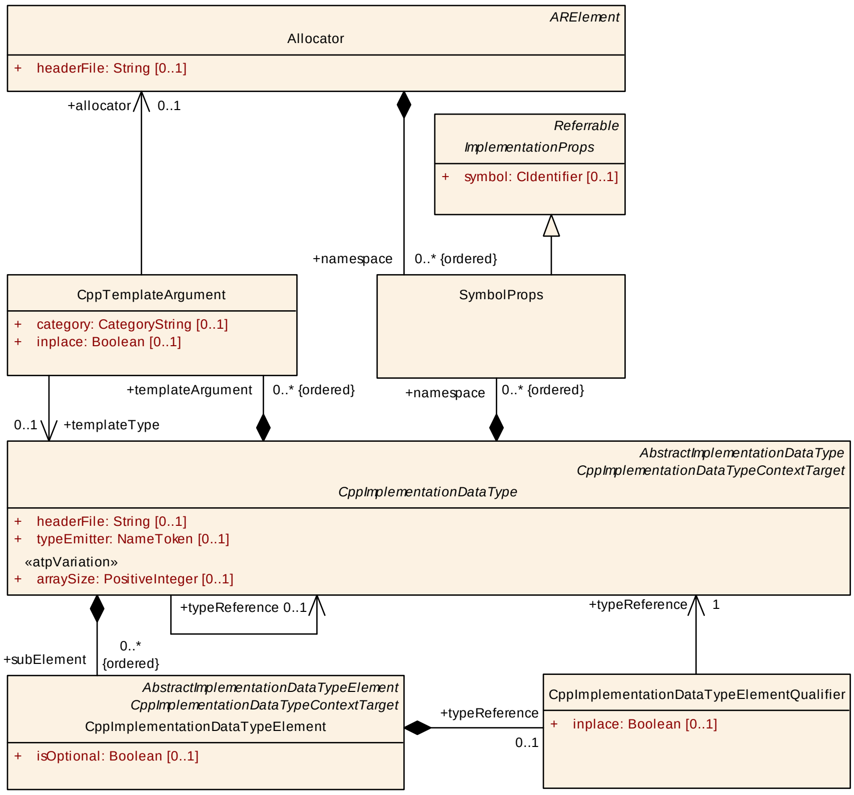
ARElement
Allocator
+ headerFile: String [0..1]
+allocator
0..1
Referrable
ImplementationProps
+ symbol: CIdentifier [0..1]
+namespace
0..* {ordered}
CppTemplateArgument
+ category: CategoryString [0..1]
+ inplace: Boolean [0..1]
SymbolProps
+templateArgument
0..* {ordered}
+namespace
0..* {ordered}
0..1
+templateType
AbstractImplementationDataType
CppImplementationDataTypeContextTarget
CppImplementationDataType
+ headerFile: String [0..1]
+ typeEmitter: NameToken [0..1]
«atpVariation»
+ arraySize: PositiveInteger [0..1]
+typeReference 0..1
+typeReference
1
+subElement
0..*
{ordered}
AbstractImplementationDataTypeElement
CppImplementationDataTypeContextTarget
CppImplementationDataTypeElement
+ isOptional: Boolean [0..1]
+typeReference
0..1
CppImplementationDataTypeElementQualifier
+ inplace: Boolean [0..1]

Figure 7.2: CppImplementationDataType

Further, [constr_1578] in [4] **must** be applied to all CppImplementationDataTypes in the following sub-chapters - this sets the necessary restriction of applicable category to CppImplementationDataType sub-element in the data type tree.

7.1.2.1 Sub-classes of CppImplementationDataType

Orthogonal to the category attribute, CppImplementationDataType is refined into two different sub-classes: StdCppImplementationDataType and CustomCppImplementationDataType (Figure 7.3)

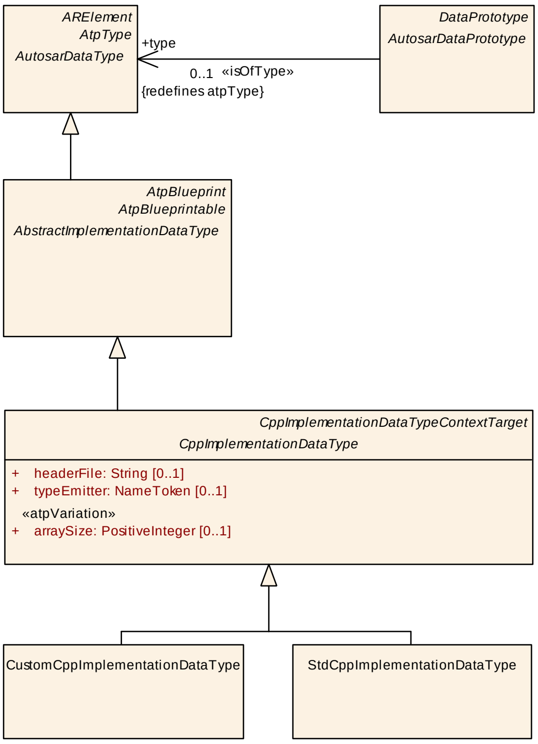
ARElement
AtpType
AutosarDataType
+type
0..1 «isOfType»
{redefines atpType}
DataPrototype
AutosarDataPrototype
AtpBlueprint
AtpBlueprintable
AbstractImplementationDataType
CppImplementationDataTypeContextTarget
CppImplementationDataType
+ headerFile: String [0..1]
+ typeEmitter: NameToken [0..1]
«atpVariation»
+ arraySize: PositiveInteger [0..1]
CustomCppImplementationDataType
StdCppImplementationDataType

Figure 7.3: Sub-classes of CppImplementationDataType

7.1.2.1.1 StdCppImplementationDataType

The `StdCppImplementationDataType` is the basis for `CppImplementationDataType`s where the exact C++ serialization shall be provided either:

- directly: by the C++ standard, e.g. 7.1.3,
- indirectly: by AUTOSAR which provides an implementation (wrapper) in `ara::core` [7], which is further based directly on the C++ standard, e.g. 7.1.6

7.1.2.1.2 CustomCppImplementationDataType

For data types modeled by `CustomCppImplementationDataType`, this sub-class facilitates the specification of data type definitions that are taken as the basis for a C++ `Language Binding` to custom implementations. In that case the declaration of the corresponding class shall be provided in the `headerFile` of the `CustomCppImplementationDataType`.

In case of a `CustomCppImplementationDataType` the model defines the following:

- `CustomCppImplementationDataType.shortName`: defines the C++ "class name" of the custom implementation

- CustomCppImplementationDataType.namespace: defines the C++ "namespace" of the custom implementation
- CustomCppImplementationDataType.headerFile: defines the C++ "header file" that contains the custom class declaration

Since the CustomCppImplementationDataType shall be capable of serving as a drop-in replacement for the StdCppImplementationDataType of the same category, it's public/protected access specifier needs to be compatible with the corresponding StdCppImplementationDataType.

This means that any existing AP application using the StdCppImplementationDataType shall be able to use the corresponding CustomCppImplementationDataType without requiring any modification of the code of the AP application. Only a re-compile of the AP application shall be required.

Thus the CustomCppImplementationDataType should conform to the following:

- The CustomCppImplementationDataType should provide the same properties (e.g. storage layout) as the corresponding StdCppImplementationDataType. e.g., a CustomCppImplementationDataType with category=VECTOR shall store its elements contiguously in memory (in the same way as ara::core::Vector emulates std::vector),
- The CustomCppImplementationDataType should provide the same public (and protected member declarations if the class has no final specifier), member functions, and operators as the corresponding StdCppImplementationDataType. The CustomCppImplementationDataType may, however, provide additional members, member functions, and operators,
- The CustomCppImplementationDataType should provide the same template<> arguments as the corresponding StdCppImplementationDataType. The CustomCppImplementationDataType may, however, provide additional template<> arguments in case these may be omitted due to default arguments,
- The method signatures of the member functions and operators of the CustomCppImplementationDataType should be compatible to the corresponding member functions and operators of the StdCppImplementationDataType, i.e., they shall exhibit the same return type and the same arguments (i.e. position and type). The member functions and operators of the CustomCppImplementationDataType may, however, provide additional arguments in case these may be omitted due to default arguments,
- The member functions and operators of the CustomCppImplementationDataType should provide the same template<> arguments (i.e. semantics and position) as the corresponding member functions and operators of the StdCppImplementationDataType. The member functions and operators of the CustomCppImplementationDataType may, however, provide additional template arguments in case these may be omitted due to default arguments,

- The member functions and operators of the CustomCppImplementationDataType should exhibit the same or a lower computational complexity as the corresponding member functions and operators of the corresponding StdCppImplementationDataType. e.g., the operator[]() of a CustomCppImplementationDataType with category=VECTOR shall exhibit a constant computational complexity (in the same way as the operator[]() of ara::core::Vector emulates the same operator[]() from std::vector),
- The serialization of CustomCppImplementationDataTypes of a specific category shall be identical to the serialization of a StdCppImplementationDataType of the same category,

7.1.3 Primitive Data Type

A Primitive CppImplementationDataType is classified by the category attribute of the CppImplementationDataType set to VALUE.

Models of Primitive CppImplementationDataType should conform to [TPS_MANI_03192] in [4].

[SWS_LBAP_00005]{DRAFT} **Standardized Primitive CppImplementationDataTypess** ⌈The StdCppImplementationDataType of category=VALUE is allowed to have one of the following shortNames:

- int8_t : see [SWS_APT_00001] in [8],
- int16_t : see [SWS_APT_00004] in [8],
- int32_t : see [SWS_APT_00007] in [8],
- int64_t : see [SWS_APT_00010] in [8],
- uint8_t : see [SWS_APT_00022] in [8],
- uint16_t: see [SWS_APT_00025] in [8],
- uint32_t: see [SWS_APT_00028] in [8],
- uint64_t: see [SWS_APT_00031] in [8],
- bool : see [SWS_APT_00049] in [8],
- float : see [SWS_APT_00043] in [8],
- double : see [SWS_APT_00046] in [8],

⌋*(RS_AP_00114, RS_AP_00122)*

7.1.3.1 Fixed Width Integer

Since only a defined set of StdCppImplementationDataTypes with category=VALUE are supported, the primitive C++ data types float, bool and double are supported in addition to chosen fixed width integer types defined in the C++ standard library header <cstdint>.

[SWS_LBAP_00006]{DRAFT} **Primitive CppImplementationDataType fixed width integers** ⌈If a StdCppImplementationDataType with the category=VALUE is referenced in a C++ Bound Interface, the C++ standard library header <cstdint> shall be included if the StdCppImplementationDataType has one of the following Cpp Implementation Data Type symbols:

- int8_t
- int16_t
- int32_t
- int64_t
- uint8_t
- uint16_t
- uint32_t
- uint64_t

⌋*(RS_AP_00114)*

7.1.4 String Data Type

A String CppImplementationDataType is classified by the category attribute of the CppImplementationDataType set to STRING.

There are two possible serializations depending on whether an Allocator is configured in a model:

- with no Allocator (7.1.4.1): defer to the default C++ std::allocator for (de-)allocating storage when the object shall grow/shrink in length,
- with Allocator (7.1.4.2): use a user provided Allocator for (de-)allocating storage when the object shall grow/shrink in length,

7.1.4.1 No Allocator

Models of CppImplementationDataType of category=STRING with no Allocator should conform to [TPS_MANI_03179] in [4].

If no Allocator is used in a model, the generated C++ Language Binding shall conform to [SWS_CORE_03001] and related items in chapter *"String data types"* in [7].

[SWS_LBAP_00015]{DRAFT} **StdCppImplementationDataType of category=STRING without Allocator** ⌈For each StdCppImplementationDataType of category=STRING there shall exist the corresponding type declaration as:

```
using <name> = ara::core::String;
```

where:

<name> is the Cpp Implementation Data Type symbol of the String CppImplementationDataType.

⌋*(RS_AP_00114, RS_AP_00127)*

7.1.4.2 Allocator

Models of CppImplementationDataType of category=STRING with Allocator should conform to [TPS_MANI_03188] in [4].

If an Allocator is used in a model, the generated C++ Language Binding shall conform to [SWS_CORE_03000] in [7].

[SWS_LBAP_00016]{DRAFT} **StdCppImplementationDataType of category=STRING with Allocator** ⌈If a StdCppImplementationDataType of category=STRING contains a templateArgument that points with the allocator reference to a custom Allocator the following type is declared:

```
    using <name> = ara::core::BasicString< <allocator> >;

```

where:

<name> is as per <name> in [SWS_LBAP_00015],

<allocator> is the <allocator namespace>::<allocator shortName> of the defined Allocator that is referenced by a CppTemplateArgument of String CppImplementationDataType with the allocator reference,

⌋*(RS_AP_00127, RS_AP_00114)*

7.1.5 Array Data Type

An Array CppImplementationDataType is classified by the category attribute of the CppImplementationDataType set to ARRAY.

Models of CppImplementationDataType of category=ARRAY should conform to: [TPS_MANI_03170], [TPS_MANI_03171], [constr_3433], [TPS_MANI_03172], [TPS_MANI_03173] in [4].

Array CppImplementationDataType serializations depend on the following information:

- the CppTemplateArgument.templateType: determines the referenced (underlying) data type of the array elements,
- the number of dimensions (one- or multi-dimensional): determined by whether the array contains a further nested array/vector.[2],
- arraySize: the number of elements for each dimension,
- inplace: determines whether the "raw" underlying data type shall be directly generated, or whether the "symbolic" name of the referenced type shall be used

Note: even if an Array CppImplementationDataType holds nested elements of types different from Array CppImplementationDataType which itself has array or vector elements, the term *one-dimensional* applies for the definition of the data type.

7.1.5.1 StdCppImplementationDataType

If the sub-class StdCppImplementationDataType is used in a model, the generated C++ Language Binding shall conform to [SWS_CORE_01201] and related items in chapter *"Array data type"* in [7].

7.1.5.1.1 One-dimensional

A *one-dimensional* StdCppImplementationDataType of category=ARRAY aggregates exactly one templateArgument that defines the type of elements that are contained in the array with the templateType reference, e.g. in case of a one-dimensional array of uint16 elements the templateType reference will point to a Primitive CppImplementationDataType that represents the uint16 element. The array size is defined with the arraySize attribute.

[SWS_LBAP_00007]{DRAFT} **StdCppImplementationDataType of category=ARRAY with one dimension** ⌈For each StdCppImplementationDataType of category=ARRAY with one dimension, there shall exist the corresponding type declaration as:

```
using <name> = ara::core::Array< <element>, <size> >;
```

where:

<name> is the Cpp Implementation Data Type symbol of the Array CppImplementationDataType,

[2] the term *dimension* is not related to the physical "size" in memory, but to the "length" semantics in the declaration of the data type

<element> is the array element specification. It is defined by the templateArgument that refers to a CppImplementationDataType with the templateType reference.

- If the CppTemplateArgument is marked with inplace=false, the shortName of the referenced CppImplementationDataType is used, and the declaration of the referenced CppImplementationDataType is generated **orthogonal** to the declaration of the ara::core::Array,
- If the CppTemplateArgument is marked with inplace=true, an anonymous CppImplementationDataType is generated as the value type of the array and the shortName of the referenced CppImplementationDataType is ignored,

<size> is the defined arraySize.

⌋*(RS_AP_00114, RS_AP_00127)*

In the case of a StdCppImplementationDataType with category=ARRAY and the shortName *MyArray* has a CppTemplateArgument that points with the templateType reference to a StdCppImplementationDataType with category=VALUE and *that* StdCppImplementationDataType.category=VALUE has a shortName=uint16_t and the CppTemplateArgument is marked with inplace=true this will result in the following code:

```
// example: inplace=true
using MyArray = ara::core::Array< std::uint16_t, 5> >;
```

If the CppTemplateArgument is marked with inplace=false, this will result in the following code:

```
// example: inplace=false
using MyInsideArray = ara::core::Array<std::uint16_t, 10>;
using MyArray = ara::core::Array<MyInsideArray, 5>;
```

7.1.5.1.2 Multi-dimensional

A *multi-dimensional* CppImplementationDataType of category=ARRAY contains nested CppImplementationDataTypes of category=ARRAY. This means, that the CppImplementationDataType of category=ARRAY will refer to a CppImplementationDataType of category=ARRAY via the aggregated templateArgument.

Such a definition describes a *two-dimensional* Array CppImplementationDataType; consequently a type with more dimensions is described by just nesting more CppImplementationDataTypes of category=ARRAY. The innermost CppImplementationDataType of category=ARRAY has the reference to the type of elements that are contained in the array.

[SWS_LBAP_00008]{DRAFT} **StdCppImplementationDataType of category=ARRAY with multiple dimensions** ⌈For each Array CppImplementationDataType having more than one dimension, there shall exist the corresponding type declaration according to [SWS_LBAP_00007] as base where <element> has a nested array for each additional dimension. The total number of dimensions is equal to the number of nested CppImplementationDataTypes with category=ARRAY plus one for the top level Array CppImplementationDataType. The array element itself is specified by the innermost CppImplementationDataType with category different from ARRAY.

```
using My2DimArray = ara::core::Array<ara::core::Array<std::uint16, 3>, 2>;
```

⌋*(RS_AP_00114)*

Please note that [SWS_LBAP_00008] and a StdCppImplementationDataType with category=ARRAY leads to an ara::core::Array type definition where the <size> definitions for each dimension are ordered from the leaf to the root ImplementationDataTypeElement, which is the same layout as the corresponding C-style array type definition where the <size> definitions for each dimension are ordered from the root to the leaf, like:

```
using My2DimArray = std::uint16_t[2][3];
```

7.1.5.2 CustomCppImplementationDataType

7.1.5.2.1 One-dimensional

If the sub-class CustomCppImplementationDataType is used, the array will be implemented as a custom array that is declared in the headerFile of the CustomCppImplementationDataType.

[SWS_LBAP_00009]{DRAFT} **CustomCppImplementationDataType of category=ARRAY** ⌈If a CustomCppImplementationDataType of category=ARRAY is used, that contains a single templateArgument that refers to a CppImplementationDataType with the templateType reference and has the arraySize attribute set to a value the following type declaration shall be available in the included headerFile of the CustomCppImplementationDataType:

```
<ClassName>< <element>, <size> >;
```

where:

<ClassName> is the Cpp Implementation Data Type symbol of the CustomCppImplementationDataType of category=ARRAY. Please note that the namespace that is defined with an ordered list of defined symbol is already handled by [SWS_LBAP_00035],

<element> is the array element specification. It is defined by the templateArgument that refers to the array element with the templateType reference.

<size> is the defined arraySize.

⌋(*RS_AP_00114*)

7.1.5.2.2 Multi-dimensional

Please note that multi-dimensional CustomCppImplementationDataTypes of category=ARRAY are handled in the same way as StdCppImplementationDataTypes of category=ARRAY. [SWS_LBAP_00008] is also valid for CustomCppImplementationDataTypes of category=ARRAY.

7.1.6 Vector Data Type

A Vector CppImplementationDataType is classified by the category attribute of the CppImplementationDataType set to VECTOR.

Models of CppImplementationDataType of category=VECTOR should conform to: [TPS_MANI_03175], [TPS_MANI_03176], [TPS_MANI_03186], [TPS_MANI_03177] in [4].

Vector CppImplementationDataType serializations depend on the following information:

- the CppTemplateArgument.templateType: determines the referenced (underlying) data type of the vector elements,
- the number of dimensions (one- or multi-dimensional) determined by whether the vector contains a further nested array/vector (see footnote in 7.1.5),
- an optional CppTemplateArgument.allocator that is used to acquire/release memory and to construct/destroy the elements in that memory,
- inplace: determines whether the "raw" underlying data type shall be directly generated, or whether the "symbolic" name of the referenced type shall be used,

The StdCppImplementationDataType of category=VECTOR is allowed to have one templateArgument that points with the templateType reference to the data type of elements that are contained in the vector.

A CppImplementationDataType of category=VECTOR aggregates one templateArgument that defines the type of elements that are contained in the vector with the templateType reference, e.g. in case of an one-dimensional vector of uint16 elements the templateType reference will point to a Primitive CppImplementationDataType that represents the uint16_t element.

Optionally the CppImplementationDataType of category=VECTOR may aggregate a second templateArgument that defines the used Allocator with the allocator reference. The type of the Allocator is the same as the data type the vector consists of.

If an Allocator is referenced then the attribute arraySize in the CppImplementationDataType of category=VECTOR can be used to define the maximal size of the vector.

[SWS_LBAP_00017]{DRAFT} **StdCppImplementationDataType of category=VECTOR with one dimension, without Allocator** ⌈For each StdCppImplementationDataType of category=VECTOR having only one dimension, there shall exist the corresponding type declaration as:

```
using <name> = ara::core::Vector< <element> >;
```

where:

<name> is the Cpp Implementation Data Type symbol of the Vector CppImplementationDataType.

<element> is the vector element specification. It is defined by the templateArgument that refers to a CppImplementationDataType with the templateType reference. The referenced CppImplementationDataType itself can be one of the data types allowed for the AP.

- If the CppTemplateArgument is marked with inplace=false, the shortName of the referenced CppImplementationDataType is used and the declaration of the referenced CppImplementationDataType is is generated **orthogonal** to the declaration of the ara::core::Vector,
- If the CppTemplateArgument is marked with inplace=true, an anonymous CppImplementationDataType is defined as the value type of the vector and the shortName of the referenced CppImplementationDataType is ignored,

⌋*(RS_AP_00114, RS_AP_00127)*

In case that a StdCppImplementationDataType with category=VECTOR and the shortName *MyVector* has a CppTemplateArgument that points with the templateType reference to a StdCppImplementationDataType with category=VECTOR and the CppTemplateArgument is marked with inplace=true this will result in the following code:

```
using MyVector = ara::core::Vector< ara::core::Vector<std::uint16_t> >;
```

If the CppTemplateArgument is marked with inplace=false this will result in the following code:

```
using MyVector = ara::core::Vector<MyInsideVector>;
using MyInsideVector = ara::core::Vector<std::uint16_t>;
```

7.1.6.1 StdCppImplementationDataType

If the sub-class StdCppImplementationDataType is used in a model, the generated C++ Language Binding shall conform to [SWS_CORE_01301] and related items in chapter *"Vector data type"* in [7].

7.1.6.1.1 One-dimensional

[SWS_LBAP_00018]{DRAFT} **StdCppImplementationDataType of category=VECTOR with one dimension, with Allocator** ⌈For each Vector CppImplementationDataType having only one dimension and a defined Allocator without a defined arraySize, there shall exist the corresponding type declaration as:

```
using <name> = ara::core::Vector< <element>, <allocator<element>> >.
```

If an arraySize is defined, the corresponding type declaration shall exist as:

```
using <name> = ara::core::Vector< <element>, <allocator<<element>,<maxSize
   >>> >;
```

where:

<name> is the Cpp Implementation Data Type symbol of the Vector CppImplementationDataType,

<element> is the vector element specification. It is defined by the templateArgument that refers to a CppImplementationDataType with the templateType reference. The referenced CppImplementationDataType itself can be one of the data types allowed for the AP.

- If the CppTemplateArgument is marked with inplace=false, the shortName of the referenced CppImplementationDataType is used and the declaration of the referenced CppImplementationDataType is defined **outside** of the vector,
- If the CppTemplateArgument is marked with inplace=true, an unnamed CppImplementationDataType is defined as value type of the vector and the shortName of the referenced CppImplementationDataType is ignored,

<allocator> is the <allocator namespace>::<allocator shortName> of the defined Allocator that is referenced by a CppTemplateArgument of Vector CppImplementationDataType with the allocator reference. The allocator receives as template arguments the element and the maxSize as number of elements of the vector. Attempts to resize the vector to a size greater than maxSize will lead to the allocator throwing an exception of type std::bad_array_new_length,

<maxSize> is the defined arraySize as number of elements of the StdCppImplementationDataType of category=VECTOR. The maxSize is a template parameter of the <allocator>,

⌋*(RS_AP_00114, RS_AP_00127)*

7.1.6.1.2 Multi-dimensional

A *multi-dimensional* CppImplementationDataType of category=VECTOR contains nested CppImplementationDataTypes of category=VECTOR. This means, that the CppImplementationDataType of category=VECTOR will refer to a CppImplementationDataType of category=VECTOR via the aggregated templateArgument.

Such a definition describes a *two-dimensional* Vector CppImplementationDataType; consequently a type with more dimensions is described by just nesting more CppImplementationDataTypes of category=VECTOR. The innermost CppImplementationDataType of category=VECTOR has the reference to the type of elements that are contained in the vector.

[SWS_LBAP_00019]{DRAFT} **StdCppImplementationDataType of category=VECTOR with multiple dimensions** ⌈For each Vector CppImplementationDataType having more than one dimension, there shall exist the corresponding type declaration according to [SWS_LBAP_00017] or [SWS_LBAP_00018] as base where <element> has a nested vector for each additional dimension. The total number of dimensions is equal to the number of nested CppImplementationDataTypes with category=VECTOR plus one for the top level Vector CppImplementationDataType. The vector element itself is specified by the innermost CppImplementationDataType with category different from VECTOR.⌋*()*

For a *two-dimensional* Vector CppImplementationDataType, as it is given as example for the definition of a *Rectangular Vector Data Type* in [4], the corresponding type declaration would look like this:

```
using DynamicDataArrayImplRectangular = ara::core::Vector< ara::core::
    Vector<std::uint16_t> >;
```

[SWS_LBAP_00020]{DRAFT} **CppImplementationDataType with category=VECTOR size semantics** ⌈The size of an CppImplementationDataType of category=VECTOR that is specified in CppImplementationDataType.arraySize will only be taken into account when the respective CppImplementationDataType defines an Allocator as defined in [SWS_LBAP_00018].⌋*()*

[SWS_LBAP_00021]{DRAFT} **Imposing memory limits with Allocator** ⌈CppImplementationDataTypes which support the CppTemplateArgument.Allocator according to [SWS_LBAP_00018], may in their respective implementations, restrict the maximum size of usable memory at the time of memory allocation in a C++ Language Binding.⌋*()*

7.1.6.2 CustomCppImplementationDataType

If the sub-class CustomCppImplementationDataType is used, the vector will be implemented as a custom vector that is declared in the headerFile of the CustomCppImplementationDataType.

7.1.6.2.1 One-dimensional

[SWS_LBAP_00022]{DRAFT} **CustomCppImplementationDataType of category=VECTOR** ⌈If a CustomCppImplementationDataType of category=VECTOR is used that contains a single templateArgument that refers to a CppImplementationDataType with the templateType reference, the following type declaration shall be available in the included headerFile of the CustomCppImplementationDataType:

```
<ClassName>< <element> >;
```

For each CustomCppImplementationDataType of category=VECTOR and a defined Allocator without a defined arraySize, there shall exist the corresponding type declaration as:

```
<ClassName>< <element>, <allocator<element>> >
```

If an arraySize is defined, the corresponding type declaration shall exist as:

```
<ClassName>< <element>, <allocator<element>,<maxSize>> >
```

where:

<ClassName> is the Cpp Implementation Data Type symbol of the CustomCppImplementationDataType of category=VECTOR. Please note that the namespace that is defined with an ordered list of defined symbol is already handled by [SWS_LBAP_00035],

<element> is the vector element specification. It is defined by the templateArgument that refers to the vector element with the templateType reference,

<allocator> is the <allocator namespace>::<allocator shortName> of the defined Allocator that is referenced by a CppTemplateArgument of Vector CppImplementationDataType with the allocator reference,

<maxSize> is the defined arraySize.

⌋*(RS_AP_00114)*

7.1.6.2.2 Multi-dimensional

Please note that multi-dimensional CustomCppImplementationDataTypes of category=VECTOR are handled in the same way as StdCppImplementationDataTypes of category=VECTOR. [SWS_LBAP_00019] is also valid for CustomCppImplementationDataTypes of category=VECTOR.

7.1.7 Structure Data Type

7.1.7.1 StdCppImplementationDataType

A Structure CppImplementationDataType is classified by the category attribute of the StdCppImplementationDataType set to STRUCTURE that has aggregated CppImplementationDataTypeElements in the role subElement.

Models of CppImplementationDataType of category=STRUCTURE should conform to [TPS_MANI_03181] in [4].

[SWS_LBAP_00010]{DRAFT} **StdCppImplementationDataType of category=STRUCTURE** ⌈For each Structure CppImplementationDataType, there shall exist the corresponding type declaration as:

```
struct <name> {<elements>};
```

where:

<name> is the Cpp Implementation Data Type symbol of the Structure CppImplementationDataType,

<elements> are record element specifications defined in Structure CppImplementationDataType by ordered CppImplementationDataTypeElements. For each record element defined by one CppImplementationDataTypeElement one record element specification <elements> is defined. The record element specifications shall be ordered according to the order of the related CppImplementationDataTypeElements in the input configuration. Sequential record elements are separated with a semi-colon.

⌋*(RS_AP_00114)*

[SWS_LBAP_00011]{DRAFT} **Structure element specification typed by CppImplementationDataType** ⌈Record element specifications <elements> of [SWS_LBAP_00010] shall exist as

```
<type> <name>;
```

where:

<type>

- is the Cpp Implementation Data Type symbol of the referred CppImplementationDataType if the typeReference is marked with inplace=false. In this case the type declaration of the referenced CppImplementationDataType is generated **outside** of the scope of the struct,
- is the type declaration of the referenced CppImplementationDataType if the typeReference is marked with inplace=true. In this case the type declaration is generated **inside** the scope of the struct,

<name> is the shortName of the ImplementationDataTypeElement.

⌋*(RS_AP_00114)*

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=ARRAY and inplace=false for the typeReference a using declaration shall exist **outside** the scope of the struct according to the rules defined in 7.1.5.

```
struct Foo {
    MyArray elementX;
};

using MyArray = ara::core::Array<std::uint8_t, 5>;
```

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=ARRAY and inplace=true for the typeReference an anonymous array shall be defined as a member type of the struct and the shortName of the referenced StdCppImplementationDataType is ignored.

```
struct Foo {
    ara::core::Array<std::uint8_t, 5> elementX;
};
```

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=VECTOR and inplace=false for the typeReference a using-declaration shall exist **outside** of the structure according to the rules defined in 7.1.6.

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=VECTOR and inplace=true for the typeReference an anonymous vector shall be defined as a member type of the struct and the shortName of the referenced StdCppImplementationDataType is ignored.

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=VARIANT and inplace=false for the typeReference a using-declaration shall exist **outside** of the structure according to the rules defined in 7.1.10.

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=VARIANT and inplace=true for the typeReference an anonymous variant shall be defined as a member type of the struct and the shortName of the referenced StdCppImplementationDataType is ignored.

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=ASSOCIATIVE_MAP and inplace=false for the typeReference a using-declaration shall exist **outside** of the structure according to the rules defined in 7.1.9.

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=ASSOCIATIVE_MAP and inplace=true for the typeReference an anonymous map shall be defined as a member type of the struct and the shortName of the referenced StdCppImplementationDataType is ignored.

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=STRUCTURE and inplace=false for the typeReference a struct-declaration shall exist **outside** of the structure according to the rule defined in [SWS_LBAP_00010].

```
struct Foo {
    Bar elementX;
};

struct Bar {
    ...
};
```

If the CppImplementationDataTypeElement points with the typeReference to a StdCppImplementationDataType with category=STRUCTURE and inplace=true for the typeReference an anonymous struct shall be defined as a member type of the struct and the shortName of the referenced StdCppImplementationDataType is ignored.

```
struct Foo {
    struct {
        ...
    } elementX;
};
```

7.1.7.2 Optional Elements

[SWS_LBAP_00012]{DRAFT} **Accessing optional record elements inside a Structure CppImplementationDataType that are serialized with the Tag-Length-Value principle.** ⌈

Optional record elements are modeled according to [TPS_MANI_01185]. For each CppImplementationDataTypeElement inside a Structure CppImplementationDataType which has CppImplementationDataTypeElement.isOptional=TRUE, there shall exist the corresponding type declaration as:

```
struct <struct_name> {
    ara::core::Optional<element_datatype> <element_name>;
}

// example with <element_datatype>=bool
struct MyStruct {
    ara::core::Optional<bool> myBool;
}
```

where:

<struct_name> is the Cpp Implementation Data Type symbol of the Structure CppImplementationDataType

<element_name> is the shortName of the optional CppImplementationDataTypeElement,

<element_datatype>

- is the shortName of the referred CppImplementationDataType if the typeReference is marked with inplace=false. In this case the type declaration of the referenced CppImplementationDataType is defined **outside** of the struct,
- is the type declaration of the referenced CppImplementationDataType if the typeReference is marked with inplace=true. In this case the type declaration is defined **inside** of the struct,

⌋*(RS_AP_00114, RS_AP_00127)*

If a CppImplementationDataTypeElement.isOptional is used in a model, the generated C++ Language Binding shall conform to [SWS_CORE_01033] and related items in chapter *"Optional data type"* in [7].

7.1.8 Enumeration Data Type

An Enumeration Data Type is classified by a Redefinition CppImplementationDataType that boils down to a Primitive CppImplementationDataType having a SwDataDefProps referencing a CompuMethod, where the CompuMethod has:

- the category attribute set to TEXTTABLE,
- and has a CompuScales container located in the compuInternalToPhys container,

- and the CompuScales container has CompuScales in role compuScale with point ranges only (i.e. lower and upper limit of a CompuScale are identical),

An Enumeration is not a plain primitive data type, but a structural description defined with a set of custom identifiers known as *enumerators* representing the possible values. In C++, an Enumeration is a first-class object and can take any of these enumerators as a value.

It is recommended that the underlying type of the enumeration should be explicitly defined to achieve both type safety and a fixed, well-defined size. Additionally, declaring enumerations as scoped enumeration classes avoids the need of unique enumerator names.

Therefore, enumerations being both typed and scoped are used instead of unscoped C++ enumerations; the underlying type is to be provided by the input configuration by defining an Enumeration Data Type.

Models of Enumeration Data Type should conform to [TPS_MANI_03187] in [4].

[SWS_LBAP_00027]{DRAFT} **Enumeration Data Type** ⌈For each Enumeration Data Type (transitively) referenced by a C++ Bound Interface, there shall exist the corresponding type declaration as:

```
enum class <name> : <type> {
  <enumerator-list>
};
```

where:

<name> is the Cpp Implementation Data Type symbol of the Redefinition CppImplementationDataType that boils down to a Primitive CppImplementationDataType.

<type> is the Primitive CppImplementationDataType that is referenced by the Redefinition CppImplementationDataType.

<enumerator-list> are the enumerators as defined by [SWS_LBAP_00028].

⌋*(RS_AP_00114)*

The enumerator names base on the CompuScale code symbolic name as defined in [TPS_SWCT_01569] in [3].

[SWS_LBAP_00028]{DRAFT} **Enumeration Data Type - enumerators** ⌈For each CompuScale with point range (i.e., lowerLimit equals upperLimit and both lowerLimit.intervalType and upperLimit.intervalType are either missing or set to CLOSED) in the Enumeration Data Type, there shall exist the corresponding enumeration nested in the declaration defined by [SWS_LBAP_00028] as:

```
  <enumeratorLiteral> = <initializer><suffix>,
```

where:

<enumeratorLiteral> is the name of the enumerator according to the following rule (lower values indicate higher priority):

1. the C++ compliant identifier specified by the symbol attribute of CompuScale if this attribute is available and not empty,
2. the string specified by the value of vt element of the CompuConst of the CompuScale if the value is a valid C++ identifier,
3. the string specified by the value of shortLabel attribute of CompuScale if the attribute is available and not empty.

<initializer> is the CompuScale's point range used as enumerator initializer,

<suffix> shall be "U" if <type> of [SWS_LBAP_00027] is an unsigned data type (i.e. if the Redefinition CppImplementationDataType boils down to a Primitive CppImplementationDataType where the Cpp Implementation Data Type symbol equals: uint8_t, uint16_t, uint32_t or uint64_t.

<suffix> shall be empty if it is a signed data type (i.e. if the Redefinition CppImplementationDataType boils down to a Primitive CppImplementationDataType where the Cpp Implementation Data Type symbol equals: int8_t, int16_t, int32_t or int64_t.

⌋*(RS_AP_00114)*

[SWS_LBAP_00029]{DRAFT} **Enumeration Data Type - skip CompuScales with non-point range** ⌈Any CompuScale with non-point range shall be simply skipped, i.e., no enumeration according to [SWS_LBAP_00028] shall be generated for those CompuScales.⌋*()*

[SWS_LBAP_00030]{DRAFT} **ARA generator rejection of incomplete Enumeration Data Types** ⌈If the input configuration contains an Enumeration Data Type and the name of an enumerator can not be determined according to [SWS_LBAP_00028], the ARA generator shall reject this input as an invalid configuration.⌋*()*

7.1.9 Associative Map Data Type

An Associative Map CppImplementationDataType is classified by the category attribute of the CppImplementationDataType set to ASSOCIATIVE_MAP.

Models of CppImplementationDataType of category=ASSOCIATIVE_MAP should conform to [TPS_MANI_03184] in [4].

7.1.9.1 StdCppImplementationDataType

If the sub-class StdCppImplementationDataType is used in a model, the generated C++ Language Binding shall conform to [SWS_CORE_01400] and related items in chapter *"Map data type"* in [7].

There are two possible serializations depending on whether an Allocator is configured in a model:

- with no Allocator (7.1.9.1.1): defer to the default C++ std::allocator for (de-)allocating storage when the object shall grow/shrink in length,
- with Allocator (7.1.9.1.2): use a user provided Allocator for (de-)allocating storage when the object shall grow/shrink in length,

7.1.9.1.1 No Allocator

[SWS_LBAP_00023]{DRAFT} **StdCppImplementationDataType with category=ASSOCIATIVE_MAP without an Allocator** ⌈For each StdCppImplementationDataType with category=ASSOCIATIVE_MAP, there shall exist the corresponding type declaration as:

```
using <name> = ara::core::Map< <key>, <value> >;
```

where:

<name> is the Cpp Implementation Data Type symbol of the Associative Map CppImplementationDataType,

<key> is the map key type specification. It is defined by the CppTemplateArgument with the category=ASSOC_MAP_KEY which is aggregated by the Associative Map CppImplementationDataType and points to a CppImplementationDataType with the templateType reference. The referenced CppImplementationDataType itself can be one of the data types allowed for the AP as long as the requirements on the key data type imposed by the ara::core::Map implementation (namely the applicability of std::less<key>) are met.

- If the CppTemplateArgument is marked with inplace=false, the shortName of the referenced CppImplementationDataType is used and the declaration of the referenced CppImplementationDataType is generated **orthogonal** to the declaration of the ara::core::Map,
- If the CppTemplateArgument is marked with inplace=true, an anonymous CppImplementationDataType is defined as key type and the shortName of the referenced CppImplementationDataType is ignored,

<value> is the mapped value type specification. It is defined by the CppTemplateArgument with the category=ASSOC_MAP_VALUE which is aggregated by the Associative Map CppImplementationDataType and points to a

CppImplementationDataType with the templateType reference. The CppImplementationDataType itself can be one of the data types allowed for the AP.

- If the CppTemplateArgument is marked with inplace=false, the shortName of the referenced CppImplementationDataType is used and the declaration of the referenced CppImplementationDataType is generated **orthogonal** to the declaration of the ara::core::Map,
- If the CppTemplateArgument is marked with inplace=true, an anonymous CppImplementationDataType is generated as the value type and the shortName of the referenced CppImplementationDataType is ignored,

⌋*(RS_AP_00114, RS_AP_00127)*

For an Associative Map CppImplementationDataType as it is given as example in chapter *Associative Map Data Type* of [4], the corresponding type declaration would look like this:

```
using MyMap = ara::core::Map<std::uint16_t, std::uint8_t>;
```

7.1.9.1.2 Allocator

[SWS_LBAP_00024]{DRAFT} **StdCppImplementationDataType with category=ASSOCIATIVE_MAP with an Allocator** ⌈For each StdCppImplementationDataType with category=ASSOCIATIVE_MAP with a defined Allocator, there shall exist the corresponding type declaration as:

```
using <name> = ara::core::Map< <key>, <value>, std::less<<key>>, <allocator
   > >;
```

where:

<name> is the Cpp Implementation Data Type symbol of the Associative Map CppImplementationDataType,

<key> is the map key type specification. It is defined by the CppTemplateArgument with the category=ASSOC_MAP_KEY which is aggregated by the Associative Map CppImplementationDataType and points to a CppImplementationDataType with the templateType reference. The referenced CppImplementationDataType itself can be one of the data types allowed for the AP as long as the requirements on the key data type imposed by the ara::core::Map implementation (namely the applicability of std::less<key>) are met.

- If the CppTemplateArgument is marked with inplace=false, the shortName of the referenced CppImplementationDataType is used and the declaration of the referenced CppImplementationDataType is defined **outside** of the map,

- If the CppTemplateArgument is marked with inplace=true, an unnamed CppImplementationDataType is defined as key type and the shortName of the referenced CppImplementationDataType is ignored,

<value> is the mapped value type specification. It is defined by the CppTemplateArgument with the category=ASSOC_MAP_VALUE which is aggregated by the Associative Map CppImplementationDataType and points to a CppImplementationDataType with the templateType reference. The CppImplementationDataType itself can be one of the data types allowed for the AP.

- If the CppTemplateArgument is marked with inplace=false, the shortName of the referenced CppImplementationDataType is used and the declaration of the referenced CppImplementationDataType is defined **outside** of the map,
- If the CppTemplateArgument is marked with inplace=true, an unnamed CppImplementationDataType is defined as value type and the shortName of the referenced CppImplementationDataType is ignored,

<allocator> is the defined Allocator that is referenced by the CppTemplateArgument of Associative Map CppImplementationDataType with the allocator reference.

⌋*(RS_AP_00114, RS_AP_00127)*

7.1.9.2 CustomCppImplementationDataType

If the sub-class CustomCppImplementationDataType is used, the map will be implemented as a custom map that is declared in the headerFile of the CustomCppImplementationDataType.

7.1.9.2.1 No Allocator

[SWS_LBAP_00025]{DRAFT} **CustomCppImplementationDataType of category=ASSOCIATIVE_MAP without Allocator** ⌈If a CustomCppImplementationDataType of category=ASSOCIATIVE_MAP is used that contains two templateArguments that both refer to a CppImplementationDataType with the templateType reference, the following type declaration shall be available in the included headerFile of the CustomCppImplementationDataType:

```
<ClassName>< <key>, <value> >;
```

where:

<ClassName> is the Cpp Implementation Data Type symbol of the CustomCppImplementationDataType of category=ASSOCIATIVE_MAP. Please note that the namespace that is defined with an ordered list of defined symbol is already handled by [SWS_LBAP_00035],

<key> is the map key type specification. It is defined by the CppTemplateArgument with the category=ASSOC_MAP_KEY which is aggregated by the Associative Map CppImplementationDataType and points to a CppImplementationDataType with the templateType reference. The referenced CppImplementationDataType itself can be one of the data types allowed for the AP,

<value> is the mapped value type specification. It is defined by the CppTemplateArgument with the category=ASSOC_MAP_VALUE which is aggregated by the Associative Map CppImplementationDataType and points to a CppImplementationDataType with the templateType reference. The CppImplementationDataType itself can be one of the data types allowed for the AP,

⌋*(RS_AP_00114)*

7.1.9.2.2 Allocator

[SWS_LBAP_00038]{DRAFT} **CustomCppImplementationDataType of category=ASSOCIATIVE_MAP with Allocator** ⌈A CustomCppImplementationDataType of category=ASSOCIATIVE_MAP with a defined Allocator shall have the following type declaration in the included headerFile of the CustomCppImplementationDataType:

```
<ClassName>< <key>, <value>, <comparator>, <allocator> >
```

where:

<ClassName> is as per <ClassName> in [SWS_LBAP_00025],

<key> is as per <key> in [SWS_LBAP_00025],

<value> is as per <value> in [SWS_LBAP_00025],

<comparator> is the comparison Functor used to sort the keys,

<allocator> is the Allocator that is referenced by the CppTemplateArgument of Associative Map CppImplementationDataType with the allocator reference.

⌋*(RS_AP_00114)*

7.1.10 Variant Data Type

A Variant CppImplementationDataType is classified by the category attribute of the CppImplementationDataType set to VARIANT.

A type alternative that is stored in a CppImplementationDataType of category=VARIANT is defined by an aggregated templateArgument that points with the templateType reference to the data type of the type alternative.

Models of CppImplementationDataType of category=VARIANT should conform to [TPS_MANI_03190], [TPS_MANI_03191] in [4].

7.1.10.1 StdCppImplementationDataType

If the sub-class StdCppImplementationDataType is used in a model, the generated C++ Language Binding shall conform to [SWS_CORE_01601] and related items in chapter *"Variant data type"* in [7].

[SWS_LBAP_00013]{DRAFT} **StdCppImplementationDataType of category=VARIANT** ⌈For each Variant CppImplementationDataType, there shall exist the corresponding type declaration as:

```
using <name> = ara::core::Variant< <elements> >;
```

where:

<name> is the Cpp Implementation Data Type symbol of the Variant CppImplementationDataType,

<elements> is the Variant element specification.

Each type alternative in a StdCppImplementationDataType of category=VARIANT is defined with a CppTemplateArgument that points with the templateType reference to the StdCppImplementationDataType that represents the alternative. For each CppTemplateArgument one element specification <elements> is defined. The Variant element specifications are ordered according the order of the related CppTemplateArguments in the input configuration. Sequential variant elements are separated with a semi-colon.

- If the CppTemplateArgument is marked with inplace=false, the shortName of the referenced CppImplementationDataType is used and the declaration of the referenced CppImplementationDataType is generated **orthogonal** to the declaration of the ara::core::Variant,
- If the CppTemplateArgument is marked with inplace=true, an anonymous CppImplementationDataType is generated as the type that may be stored in this variant and the shortName of the referenced CppImplementationDataType is ignored,

⌋*(RS_AP_00114, RS_AP_00127)*

A Variant data type describes a kind of structural overlay. Defining only one element in a VARIANT is therefore not reasonable.

7.1.10.2 CustomCppImplementationDataType

If the sub-class CustomCppImplementationDataType is used, the variant will be implemented as a custom variant that is declared in the headerFile of the CustomCppImplementationDataType.

[SWS_LBAP_00014]{DRAFT} **CustomCppImplementationDataType of category=VARIANT** ⌈If a CustomCppImplementationDataType of category=VARIANT is used, the following type declaration shall be available in the included headerFile:

```
<ClassName>< <elements> >;
```

where:

<ClassName> is the Cpp Implementation Data Type symbol of the CustomCppImplementationDataType of category=VARIANT. Please note that the namespace that is defined with an ordered list of defined symbol is already handled by [SWS_LBAP_00035],

<elements> is the variant element specification. Each type alternative in a CustomCppImplementationDataType of category=VARIANT is defined with a CppTemplateArgument that points with the templateType reference to the CustomCppImplementationDataType that represents the alternative. For each CppTemplateArgument one element specification <elements> is defined. The Variant element specifications are ordered according the order of the related CppTemplateArguments in the input configuration. Sequential variant elements are separated with a semi-colon.

⌋*(RS_AP_00114)*

7.1.11 Redefinition of Implementation Data Type

A Redefinition CppImplementationDataType is classified by the category attribute of the referring StdCppImplementationDataType set to TYPE_REFERENCE.

The StdCppImplementationDataType of category=TYPE_REFERENCE points to an another CppImplementationDataType with the typeReference and defines a type alias in this way.

Models of Redefinition CppImplementationDataType should conform to [TPS_MANI_03193] in [4].

[SWS_LBAP_00026]{DRAFT} **StdCppImplementationDataType of category=TYPE_REFERENCE** ⌈For each Redefinition CppImplementationDataType which is typed by an StdCppImplementationDataType, there shall exist the corresponding type declaration as:

```
using <name> = <type>;
```

```

// example:
using MyTypeAlias = SomeOtherTypeDefinedElsewhere;
```

where:

<name> is the Cpp Implementation Data Type symbol of the Redefinition CppImplementationDataType,

<type> is the Cpp Implementation Data Type symbol of the referred StdCppImplementationDataType.

⌋(*RS_AP_00114*)

7.1.12 Scale Linear And Texttable Data Type

A Scale Linear And Texttable Data Type is classified by a Redefinition CppImplementationDataType that boils down to a Primitive CppImplementationDataType having a SwDataDefProps referencing a CompuMethod, where the CompuMethod has:

- the category=SCALE_LINEAR_AND_TEXTTABLE,
- and has a CompuScales container located in the compuInternalToPhys container,
- and the CompuScales container has CompuScales in role compuScale with point ranges (i.e. lower and upper limit of a CompuScale are identical) and non-point ranges where the CompuRationalCoeffs define a linear function,

A Scale Linear And Texttable Data Type is not a plain primitive data type, but a structural description defined with an Enumeration Data Type. The Scale Linear And Texttable Data Type can hold the values of the enumerators and also the values of the underlying type of the Enumeration Data Type it was defined with.

If a Scale Linear And Texttable Data Type is used in a model, the generated C++ Language Binding shall conform to [SWS_CORE_08101] and related items in chapter *"ScaleLinearAndTexttable data type"* in [7].

[SWS_LBAP_00031]{DRAFT} **Scale Linear And Texttable Data Type** ⌈For each Scale Linear And Texttable Data Type there shall exist the corresponding type declaration as:

```
using <name> = ara::core::ScaleLinearAndTexttable<enum_type>;
```

where:

<name> is the Cpp Implementation Data Type symbol of the Scale Linear And Texttable Data Type,

<enum_type> is the generated Enumeration Data Type used to specify the Scale Linear And Texttable Data Type.

⌋*(RS_AP_00114)*

8 API specification

The LBAP has no dedicated API specification.

A Mentioned Manifest Elements

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

Chapter is generated.

Class	***AbstractImplementationDataType*** (abstract)			
Package	M2::AUTOSARTemplates::CommonStructure::ImplementationDataTypes			
Note	This meta-class represents an abstract base class for different flavors of ImplementationDataType.			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *AutosarDataType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Subclasses	*CppImplementationDataType*, ImplementationDataType			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
–	–	–	–	–

Table A.1: AbstractImplementationDataType

Class	**Allocator**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::CppImplementationDataType			

Note	This meta-class represents the ability to take influence on the way objects are allocated in memory, for example it can be controlled whether an objects is allocated on the heap or on the stack.
	Tags:
	atp.Status=draft
	atp.recommendedPackage=Allocators			

Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
headerFile	String	0..1	attr	Configuration of the Header File with the custom class declaration **Tags:**atp.Status=draft
namespace (ordered)	SymbolProps	*	aggr	This aggregation allows for the definition of a namespace of an Allocator. **Tags:**atp.Status=draft

Table A.2: Allocator

Class	***ApplicationDataType*** (abstract)
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes
Note	ApplicationDataType defines a data type from the application point of view. Especially it should be used whenever something "physical" is at stake. An ApplicationDataType represents a set of values as seen in the application model, such as measurement units. It does not consider implementation details such as bit-size, endianess, etc. It should be possible to model the application level aspects of a VFB system by using ApplicationDataTypes only.
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *AutosarDataType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*
Subclasses	*ApplicationCompositeDataType*, ApplicationPrimitiveDataType

▽

△

Class	***ApplicationDataType*** (abstract)			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
–	–	–	–	–

Table A.3: ApplicationDataType

Class	***AutosarDataPrototype*** (abstract)			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::DataPrototypes			
Note	Base class for prototypical roles of an AutosarDataType.			
Base	*ARObject*, *AtpFeature*, *AtpPrototype*, *DataPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Subclasses	ArgumentDataPrototype, Field, ParameterDataPrototype, PersistencyDataElement, VariableDataPrototype			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
type	AutosarDataType	0..1	tref	This represents the corresponding data type. **Stereotypes:** isOfType

Table A.4: AutosarDataPrototype

Class	***AutosarDataType*** (abstract)			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Note	Abstract base class for user defined AUTOSAR data types for software.			
Base	*ARElement*, *ARObject*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Subclasses	*AbstractImplementationDataType*, *ApplicationDataType*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
swDataDefProps	SwDataDefProps	0..1	aggr	The properties of this AutosarDataType.

Table A.5: AutosarDataType

Class	**CompuConst**			
Package	M2::MSR::AsamHdo::ComputationMethod			
Note	This meta-class represents the fact that the value of a computation method scale is constant.			
Base	*ARObject*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***

compuConstContentType	CompuConstContent	0..1	aggr	This is the actual content of the constant compu method scale.
				Tags:
				xml.roleElement=false
				xml.roleWrapperElement=false
				xml.sequenceOffset=10
				xml.typeElement=false
				xml.typeWrapperElement=false

Table A.6: CompuConst

Class	**CompuConstTextContent**			
Package	M2::MSR::AsamHdo::ComputationMethod			
Note	This meta-class represents the textual content of a scale.			
Base	*ARObject*, *CompuConstContent*			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
vt	VerbatimString	0..1	attr	This represents a textual constant in the computation method.

Table A.7: CompuConstTextContent

Class	**CompuMethod**			
Package	M2::MSR::AsamHdo::ComputationMethod			
Note	This meta-class represents the ability to express the relationship between a physical value and the mathematical representation. Note that this is still independent of the technical implementation in data types. It only specifies the formula how the internal value corresponds to its physical pendant. **Tags:**atp.recommendedPackage=CompuMethods			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *CollectableElement*, *Identifiable*, *Multilanguage Referrable*, *PackageableElement*, *Referrable*			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
compuInternal ToPhys	Compu	0..1	aggr	This specifies the computation from internal values to physical values. **Tags:**xml.sequenceOffset=80
compuPhysTo Internal	Compu	0..1	aggr	This represents the computation from physical values to the internal values. **Tags:**xml.sequenceOffset=90
displayFormat	DisplayFormatString	0..1	attr	This property specifies, how the physical value shall be displayed e.g. in documents or measurement and calibration tools. **Tags:**xml.sequenceOffset=20
unit	Unit	0..1	ref	This is the physical unit of the Physical values for which the CompuMethod applies. **Tags:**xml.sequenceOffset=30

Table A.8: CompuMethod

Class	**CompuRationalCoeffs**			
Package	M2::MSR::AsamHdo::ComputationMethod			
Note	This meta-class represents the ability to express a rational function by specifying the coefficients of nominator and denominator.			
Base	*ARObject*			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
compu Denominator	CompuNominator Denominator	0..1	aggr	This is the denominator of the expression. **Tags:**xml.sequenceOffset=30
compu Numerator	CompuNominator Denominator	0..1	aggr	This is the numerator of the rational expression. **Tags:**xml.sequenceOffset=20

Table A.9: CompuRationalCoeffs

Class	**CompuScale**			
Package	M2::MSR::AsamHdo::ComputationMethod			
Note	This meta-class represents the ability to specify one segment of a segmented computation method.			
Base	*ARObject*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
compuInverseValue	CompuConst	0..1	aggr	This is the inverse value of the constraint. This supports the case that the scale is not reversible per se. **Tags:**xml.sequenceOffset=60

compuScaleContents	CompuScaleContents	0..1	aggr	This represents the computation details of the scale.
				Tags:
				xml.roleElement=false
				xml.roleWrapperElement=false
				xml.sequenceOffset=70
				xml.typeElement=false
				xml.typeWrapperElement=false

desc	MultiLanguageOverviewParagraph	0..1	aggr	<desc> represents a general but brief description of the object in question. **Tags:**xml.sequenceOffset=30
lowerLimit	Limit	0..1	attr	This specifies the lower limit of the scale. **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=preCompileTime xml.sequenceOffset=40
mask	PositiveInteger	0..1	attr	In difference to all the other computational methods every COMPU-SCALE will be applied including the bit MASK. Therefore it is allowed for this type of COMPU-METHOD, that COMPU-SCALES overlap. To calculate the string reverse to a value, the string has to be split and the according value for each substring has to be summed up. The sum is finally transmitted. The processing has to be done in order of the COMPU-SCALE elements. **Tags:**xml.sequenceOffset=35
shortLabel	Identifier	0..1	attr	This element specifies a short name for the particular scale. The name can for example be used to derive a programming language identifier. **Tags:**xml.sequenceOffset=20
symbol	CIdentifier	0..1	attr	The symbol, if provided, is used by code generators to get a C identifier for the CompuScale. The name will be used as is for the code generation, therefore it needs to be unique within the generation context. **Tags:**xml.sequenceOffset=25
upperLimit	Limit	0..1	attr	This specifies the upper limit of a of the scale. **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=preCompileTime xml.sequenceOffset=50

Table A.10: CompuScale

Class	**CompuScales**			
Package	M2::MSR::AsamHdo::ComputationMethod			
Note	This meta-class represents the ability to stepwise express a computation method.			
Base	*ARObject*, *CompuContent*			
Attribute	**Type**	**Mult.**	**Kind**	**Note**

compuScale (ordered)	CompuScale	*	aggr	This represents one scale within the compu method. Note that it contains a Variationpoint in order to support blueprints of enumerations.
				Stereotypes: atpVariation
				Tags:
				vh.latestBindingTime=blueprintDerivationTime
				xml.roleElement=true
				xml.roleWrapperElement=true
				xml.sequenceOffset=40
				xml.typeElement=false
				xml.typeWrapperElement=false

Table A.11: CompuScales

Class	***CppImplementationDataType*** (abstract)			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::CppImplementationDataType			
Note	This meta-class represents the way to specify a reusable data type definition taken as a the basis for a C++ language binding **Tags:**atp.Status=draft			
Base	*ARElement*, *ARObject*, *AbstractImplementationDataType*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *AutosarDataType*, *CollectableElement*, *CppImplementationDataTypeContextTarget*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Subclasses	CustomCppImplementationDataType, StdCppImplementationDataType			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
arraySize	PositiveInteger	0..1	attr	This attribute can be used to specify the array size if the enclosing CppImplementationDataType has array semantics. **Stereotypes:** atpVariation **Tags:** atp.Status=draft vh.latestBindingTime=preCompileTime
headerFile	String	0..1	attr	Configuration of the Header File with the custom class declaration. **Tags:**atp.Status=draft
namespace (ordered)	SymbolProps	*	aggr	This aggregation allows for the definition an own namespace for the enclosing CppImplementationDataType. **Tags:**atp.Status=draft
subElement (ordered)	CppImplementationDataTypeElement	*	aggr	This represents the collection of sub-elements of the enclosing CppImplementationDataType **Tags:**atp.Status=draft
templateArgument (ordered)	CppTemplateArgument	*	aggr	This aggreation allows for the specification of properties of template arguments **Tags:**atp.Status=draft
typeEmitter	NameToken	0..1	attr	This attribute can be taken to control how the respective CppImplementationDataType is contributed to the language binding. **Tags:**atp.Status=draft

▽

△

Class	*CppImplementationDataType* (abstract)			
typeReference	CppImplementation DataType	0..1	ref	This reference shall be defined to define a type reference (a.k.a. typedef). **Tags:**atp.Status=draft

Table A.12: CppImplementationDataType

Class	**CppImplementationDataTypeElement**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::CppImplementationDataType			
Note	Declares a data object which is locally aggregated. Such an element can only be used within the scope where it is aggregated. A CppImplementationDataTypeElement is used to represent an element of a structure, defining its type. **Tags:**atp.Status=draft			
Base	*ARObject*, *AbstractImplementationDataTypeElement*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *CppImplementationDataTypeContextTarget*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
isOptional	Boolean	0..1	attr	This attribute represents the ability to declare the enclosing CppImplementationDataTypeElement as optional. This means the that, at runtime, the CppImplementationDataTypeElement may or may not have a valid value and shall therefore be ignored. The underlying runtime software provides means to set the CppImplementationDataTypeElement as not valid at the sending end of a communication and determine its validity at the receiving end. **Tags:**atp.Status=draft
typeReference	CppImplementation DataTypeElement Qualifier	0..1	aggr	This aggregation defines the type of the CppImplementationDataTypeElement and determines whether in C++ the CppImplementationDataTypeElement is defined inside or outside of the enclosing CppImplementationDataType. **Tags:**atp.Status=draft

Table A.13: CppImplementationDataTypeElement

Class	**CppImplementationDataTypeElementQualifier**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::CppImplementationDataType			
Note	This element qualifies the typeReference of the CppImplementationDataTypeElement to the CppImplementationDataType. **Tags:**atp.Status=draft			
Base	*ARObject*			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
inplace	Boolean	0..1	attr	This attribute defines whether the member type of the CppImplementationDataTypeElement in C++ is an embedded type element inside of the enclosing struct (true) or whether the type declaration is defined outside of the struct. **Tags:**atp.Status=draft
typeReference	CppImplementation DataType	1	ref	This reference defines a type reference. **Tags:**atp.Status=draft

Table A.14: CppImplementationDataTypeElementQualifier

Class	**CppTemplateArgument**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::CppImplementationDataType			
Note	This meta-class has the ability to define properties for template arguments. **Tags:**atp.Status=draft			
Base	*ARObject*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
allocator	Allocator	0..1	ref	This reference identifies the applicable allocator. **Tags:**atp.Status=draft
category	CategoryString	0..1	attr	This attribute shall be used to contribute further clarification regarding the semantics of the enclosing CppTemplateArgument. **Tags:**atp.Status=draft
inplace	Boolean	0..1	attr	This attribute specifies whether the shortName of the referenced templateType is used in the code generation and the type declaration is defined outside of the enclosing CppImplementationDataType (true) or whether the type definition is embedded inside of the enclosing CppImplementationDataType and the shortName is ignored (false). **Tags:**atp.Status=draft
templateType	CppImplementationDataType	0..1	ref	This reference identifies the data type of the specific template argument required for the language binding. **Tags:**atp.Status=draft

Table A.15: CppTemplateArgument

Class	**CustomCppImplementationDataType**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::CppImplementationDataType			

Note	This meta-class represents the way to specify a data type definition that is taken as the basis for a C++ language binding to a custom implementation that is declared in the configured header file. The ShortName of this CustomCppImplementationDataType defines the Class-Name of the custom implementation.
	Tags:
	atp.Status=draft
	atp.recommendedPackage=CppImplementationDataTypes			

Base	*ARElement*, *ARObject*, *AbstractImplementationDataType*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *AutosarDataType*, *CollectableElement*, *CppImplementationDataType*, *CppImplementationDataTypeContextTarget*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
–	–	–	–	–

Table A.16: CustomCppImplementationDataType

Class	***DataPrototype*** (abstract)			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::DataPrototypes			
Note	Base class for prototypical roles of any data type.			
Base	*ARObject*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Subclasses	*ApplicationCompositeElementDataPrototype*, *AutosarDataPrototype*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***

▽

△

Class	***DataPrototype*** (abstract)			
swDataDef Props	SwDataDefProps	0..1	aggr	This property allows to specify data definition properties which apply on data prototype level.

Table A.17: DataPrototype

Class	**DataTypeMap**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Note	This class represents the relationship between ApplicationDataType and its implementing Abstract ImplementationDataType.			
Base	*ARObject*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
applicationData Type	ApplicationDataType	0..1	ref	This is the corresponding ApplicationDataType
implementation DataType	AbstractImplementation DataType	0..1	ref	This is the corresponding AbstractImplementationData Type.

Table A.18: DataTypeMap

Class	**DataTypeMappingSet**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Datatype::Datatypes			
Note	This class represents a list of mappings between ApplicationDataTypes and ImplementationDataTypes. In addition, it can contain mappings between ImplementationDataTypes and ModeDeclarationGroups. **Tags:**atp.recommendedPackage=DataTypeMappingSets			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *CollectableElement*, *Identifiable*, *Multilanguage Referrable*, *PackageableElement*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
dataTypeMap	DataTypeMap	*	aggr	This is one particular association between an Application DataType and its AbstractImplementationDataType.
modeRequest TypeMap	ModeRequestTypeMap	*	aggr	This is one particular association between an Mode DeclarationGroup and its AbstractImplementationData Type.

Table A.19: DataTypeMappingSet

Class	***Identifiable*** (abstract)
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::Identifiable
Note	Instances of this class can be referred to by their identifier (within the namespace borders). In addition to this, Identifiables are objects which contribute significantly to the overall structure of an AUTOSAR description. In particular, Identifiables might contain Identifiables.
Base	*ARObject*, *MultilanguageReferrable*, *Referrable*
Subclasses	ARPackage, *AbstractDoIpLogicAddressProps*, *AbstractEvent*, *AbstractImplementationDataTypeElement*, *AbstractSecurityEventFilter*, *AbstractSecurityIdsmInstanceFilter*, *AbstractServiceInstance*, *Abstract SignalBasedToISignalTriggeringMapping*, *AdaptiveModuleInstantiation*, AdaptiveSwcInternalBehavior, ApApplicationEndpoint, ApplicationEndpoint, ApplicationError, ArtifactChecksum, *AtpBlueprint*, *Atp Blueprintable*, *AtpClassifier*, *AtpFeature*, AutosarOperationArgumentInstance, AutosarVariableInstance, *BuildActionEntity*, BuildActionEnvironment, Chapter, CheckpointTransition, ClassContentConditional, ClientIdDefinition, ClientServerOperation, Code, *CollectableElement*, ComManagementMapping, *Comm ConnectorPort*, *CommunicationConnector*, *CommunicationController*, Compiler, ConsistencyNeeds, ConsumedEventGroup, CouplingPort, *CouplingPortStructuralElement*, CryptoCertificate, CryptoKeySlot, CryptoProvider, *CryptoServiceMapping*, DataPrototypeGroup, DataTransformation, DdsDomainRange, ▽

▽

△

Class	***Identifiable*** (abstract)			
	△ DependencyOnArtifact, DeterministicClientResourceNeeds, *DiagEventDebounceAlgorithm*, Diagnostic ConnectedIndicator, DiagnosticDataElement, DiagnosticDebounceAlgorithmProps, DiagnosticFunction InhibitSource, *DiagnosticRoutineSubfunction*, DltApplication, DltArgument, DltMessage, DoIpInterface, DoIpLogicAddress, DoIpRoutingActivation, E2EProfileConfiguration, End2EndEventProtectionProps, EndToEndProtection, EthernetWakeupSleepOnDatalineConfig, EventHandler, EventMapping, Exclusive Area, *ExecutableEntity*, *ExecutionTime*, FMAttributeDef, FMFeatureMapAssertion, FMFeatureMap Condition, FMFeatureMapElement, FMFeatureRelation, FMFeatureRestriction, FMFeatureSelection, FieldMapping, FireAndForgetMapping, FlexrayArTpNode, FlexrayTpPduPool, *FrameTriggering*, General Parameter, GlobalSupervision, GlobalTimeGateway, *GlobalTimeMaster*, *GlobalTimeSlave*, *Health Channel*, *HeapUsage*, HwAttributeDef, HwAttributeLiteralDef, HwPin, HwPinGroup, IPSecRule, IPv6Ext HeaderFilterList, ISignalToIPduMapping, ISignalTriggering, *IdentCaption*, InternalTriggeringPoint, Keyword, LifeCycleState, Linker, MacMulticastGroup, McDataInstance, MemorySection, Method Mapping, ModeDeclaration, ModeDeclarationMapping, ModeSwitchPoint, NetworkEndpoint, *NmCluster*, *NmNode*, *PackageableElement*, ParameterAccess, PduActivationRoutingGroup, PduToFrameMapping, PduTriggering, PerInstanceMemory, *PersistencyDeploymentElement*, *PersistencyInterfaceElement*, *Phm Supervision*, *PhysicalChannel*, PortGroup, *PortInterfaceMapping*, PossibleErrorReaction, ProcessTo MachineMapping, Processor, ProcessorCore, PskIdentityToKeySlotMapping, RecoveryNotification, ResourceConsumption, ResourceGroup, RootSwClusterDesignComponentPrototype, RootSw ComponentPrototype, RootSwCompositionPrototype, RptComponent, RptContainer, RptExecutable Entity, RptExecutableEntityEvent, RptExecutionContext, RptProfile, RptServicePoint, RunnableEntity Group, *SdgAttribute*, SdgClass, SecOcJobMapping, SecOcJobRequirement, SecureCommunication AuthenticationProps, *SecureCommunicationDeployment*, SecureCommunicationFreshnessProps, SecurityEventContextProps, *ServiceEventDeployment*, *ServiceFieldDeployment*, ServiceInterface ElementSecureComConfig, *ServiceMethodDeployment*, *ServiceNeeds*, SignalServiceTranslationEvent Props, SignalServiceTranslationProps, SocketAddress, SoftwarePackageStep, SomeipEventGroup, SomeipProvidedEventGroup, SomeipTpChannel, *SpecElementReference*, *StackUsage*, StaticSocket Connection, StructuredReq, SupervisionCheckpoint, SupervisionMode, SupervisionModeCondition, Sw GenericAxisParamType, SwServiceArg, SwcServiceDependency, SystemMapping, SystemMemory Usage, *TimeBaseResource*, TimingCondition, *TimingConstraint*, *TimingDescription*, TimingExtension Resource, TimingModeInstance, TlsCryptoCipherSuite, TlsCryptoCipherSuiteProps, TlsJobMapping, Topic1, TpAddress, TraceableTable, TraceableText, *TracedFailure*, *TransformationProps*, Transformation Technology, Trigger, UcmDescription, UcmStep, VariableAccess, VariationPointProxy, VehicleRollout Step, ViewMap, VlanConfig, WaitPoint			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
adminData	AdminData	0..1	aggr	This represents the administrative data for the identifiable object. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=adminData xml.sequenceOffset=-40
annotation	Annotation	*	aggr	Possibility to provide additional notes while defining a model element (e.g. the ECU Configuration Parameter Values). These are not intended as documentation but are mere design notes. **Tags:**xml.sequenceOffset=-25
category	CategoryString	0..1	attr	The category is a keyword that specializes the semantics of the Identifiable. It affects the expected existence of attributes and the applicability of constraints. **Tags:**xml.sequenceOffset=-50
desc	MultiLanguageOverview Paragraph	0..1	aggr	This represents a general but brief (one paragraph) description what the object in question is about. It is only one paragraph! Desc is intended to be collected into overview tables. This property helps a human reader to identify the object in question. More elaborate documentation, (in particular how the object is built or used) should go to "introduction". **Tags:**xml.sequenceOffset=-60

▽

△

Class	***Identifiable*** (abstract)			
introduction	DocumentationBlock	0..1	aggr	This represents more information about how the object in question is built or is used. Therefore it is a DocumentationBlock. **Tags:**xml.sequenceOffset=-30
uuid	String	0..1	attr	The purpose of this attribute is to provide a globally unique identifier for an instance of a meta-class. The values of this attribute should be globally unique strings prefixed by the type of identifier. For example, to include a DCE UUID as defined by The Open Group, the UUID would be preceded by "DCE:". The values of this attribute may be used to support merging of different AUTOSAR models. The form of the UUID (Universally Unique Identifier) is taken from a standard defined by the Open Group (was Open Software Foundation). This standard is widely used, including by Microsoft for COM (GUIDs) and by many companies for DCE, which is based on CORBA. The method for generating these 128-bit IDs is published in the standard and the effectiveness and uniqueness of the IDs is not in practice disputed. If the id namespace is omitted, DCE is assumed. An example is "DCE:2fac1234-31f8-11b4-a222-08002b34c003". The uuid attribute has no semantic meaning for an AUTOSAR model and there is no requirement for AUTOSAR tools to manage the timestamp. **Tags:**xml.attribute=true

Table A.20: Identifiable

Class	**ImplementationDataType**			
Package	M2::AUTOSARTemplates::CommonStructure::ImplementationDataTypes			
Note	Describes a reusable data type on the implementation level. This will typically correspond to a typedef in C-code. **Tags:**atp.recommendedPackage=ImplementationDataTypes			
Base	*ARElement*, *ARObject*, *AbstractImplementationDataType*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *AutosarDataType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
dynamicArray SizeProfile	String	0..1	attr	Specifies the profile which the array will follow in case this data type is a variable size array.
isStructWith Optional Element	Boolean	0..1	attr	This attribute is only valid if the attribute category is set to STRUCTURE. If set to True, this attribute indicates that the ImplementationDataType has been created with the intention to define at least one element of the structure as optional.
subElement (ordered)	ImplementationData TypeElement	*	aggr	Specifies an element of an array, struct, or union data type. The aggregation of ImplementionDataTypeElement is subject to variability with the purpose to support the conditional existence of elements inside a Implementation DataType representing a structure. **Stereotypes:** atpVariation **Tags:**vh.latestBindingTime=preCompileTime

▽

△

Class	**ImplementationDataType**			
symbolProps	SymbolProps	0..1	aggr	This represents the SymbolProps for the Implementation DataType. **Stereotypes:** atpSplitable **Tags:**atp.Splitkey=symbolProps.shortName
typeEmitter	NameToken	0..1	attr	This attribute is used to control which part of the AUTOSAR toolchain is supposed to trigger data type definitions.

Table A.21: ImplementationDataType

Class	**ImplementationDataTypeElement**			
Package	M2::AUTOSARTemplates::CommonStructure::ImplementationDataTypes			
Note	Declares a data object which is locally aggregated. Such an element can only be used within the scope where it is aggregated. This element either consists of further subElements or it is further defined via its swDataDefProps. There are several use cases within the system of ImplementationDataTypes fur such a local declaration: • It can represent the elements of an array, defining the element type and array size • It can represent an element of a struct, defining its type • It can be the local declaration of a debug element.			
Base	*ARObject*, *AbstractImplementationDataTypeElement*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
arrayImplPolicy	ArrayImplPolicyEnum	0..1	attr	This attribute controls the implementation of the payload of an array. It shall only be used if the enclosing ImplementationDataType constitutes an array.
arraySize	PositiveInteger	0..1	attr	The existence of this attributes (if bigger than 0) defines the size of an array and declares that this Implementation DataTypeElement represents the type of each single array element. **Stereotypes:** atpVariation **Tags:**vh.latestBindingTime=preCompileTime
arraySize Handling	ArraySizeHandling Enum	0..1	attr	The way how the size of the array is handled in case of a variable size array.
arraySize Semantics	ArraySizeSemantics Enum	0..1	attr	This attribute controls the meaning of the value of the array size.
isOptional	Boolean	0..1	attr	This attribute represents the ability to declare the enclosing ImplementationDataTypeElement as optional. This means that, at runtime, the ImplementationDataType Element may or may not have a valid value and shall therefore be ignored. The underlying runtime software provides means to set the CppImplementationDataTypeElement as not valid at the sending end of a communication and determine its validity at the receiving end.
subElement (ordered)	ImplementationData TypeElement	*	aggr	Element of an array, struct, or union in case of a nested declaration (i.e. without using "typedefs"). The aggregation of ImplementionDataTypeElement is subject to variability with the purpose to support the conditional existence of elements inside a Implementation DataType representing a structure. **Stereotypes:** atpVariation **Tags:**vh.latestBindingTime=preCompileTime

▽

△

Class	**ImplementationDataTypeElement**			
swDataDefProps	SwDataDefProps	0..1	aggr	The properties of this ImplementationDataTypeElement.

Table A.22: ImplementationDataTypeElement

Class	***ImplementationProps*** (abstract)			
Package	M2::AUTOSARTemplates::CommonStructure::Implementation			
Note	Defines a symbol to be used as (depending on the concrete case) either a complete replacement or a prefix when generating code artifacts.			
Base	*ARObject*, *Referrable*			
Subclasses	BswSchedulerNamePrefix, ExecutableEntityActivationReason, SectionNamePrefix, SymbolProps, SymbolicNameProps			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
symbol	CIdentifier	0..1	attr	The symbol to be used as (depending on the concrete case) either a complete replacement or a prefix.

Table A.23: ImplementationProps

<table>
<tr><td><i>Primitive</i></td><td colspan="4"><b>Limit</b></td></tr>
<tr><td><i>Package</i></td><td colspan="4">M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::PrimitiveTypes</td></tr>
<tr><td><i>Note</i></td><td colspan="4">This class represents the ability to express a numerical limit. Note that this is in fact a NumericalVariationPoint but has the additional attribute intervalType.
<b>Tags:</b>
xml.xsd.customType=LIMIT-VALUE
xml.xsd.pattern=(0[xX][0-9a-fA-F]+)|(0[0-7]+)|(0[bB][0-1]+)|(([+\-]?[1-9][0-9]+(\.[0-9]+)?|[+\-]?[0-9](\.[0-9]+)?)([eE]([+\-]?)[0-9]+)?)|\.0|INF|-INF|NaN
xml.xsd.type=string</td></tr>
<tr><td><i>Attribute</i></td><td><i>Type</i></td><td><i>Mult.</i></td><td><i>Kind</i></td><td><i>Note</i></td></tr>
<tr><td>intervalType</td><td>IntervalTypeEnum</td><td>0..1</td><td>attr</td><td>This specifies the type of the interval. If the attribute is missing the interval shall be considered as "CLOSED".
<b>Tags:</b>xml.attribute=true</td></tr>
</table>

Table A.24: Limit

Class	**PersistencyKeyValueStorageInterface**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::PortInterface			

Note	This meta-class provides the ability to implement a PortInterface for supporting persistency use cases for data.
	Tags:
	atp.Status=draft
	atp.recommendedPackage=PersistencyKeyValueStorageInterfaces			

Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *PersistencyInterface*, *PortInterface*, *Referrable*			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
dataElement	PersistencyDataElement	*	aggr	This aggregation represents the collection of PersistencyDataElements in the context of the enclosing PersistencyKeyValueStorageInterface. **Tags:**atp.Status=draft

▽

△

Class	**PersistencyKeyValueStorageInterface**			
dataTypeFor Serialization	AbstractImplementation DataType	*	ref	This reference identifies the AbstractImplementationData Types that shall be supported for storing in a key-value storage in addition to the types already determined from tha aggregation of PersistencyDataElement. **Tags:**atp.Status=draft

Table A.25: PersistencyKeyValueStorageInterface

Class	***PortInterface*** (abstract)			
Package	M2::AUTOSARTemplates::SWComponentTemplate::PortInterface			
Note	Abstract base class for an interface that is either provided or required by a port of a software component.			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Subclasses	*AbstractRawDataStreamInterface*, *AbstractSynchronizedTimeBaseInterface*, ClientServerInterface, *CryptoInterface*, *DataInterface*, *DiagnosticPortInterface*, LogAndTraceInterface, ModeSwitchInterface, *PersistencyInterface*, *PlatformHealthManagementInterface*, SecurityEventReportInterface, Service Interface, TriggerInterface			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
namespace (ordered)	SymbolProps	*	aggr	This represents the SymbolProps used for the definition of a hierarchical namespace applicable for the generation of code artifacts out of the definition of a ServiceInterface. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=namespace.shortName atp.Status=draft

Table A.26: PortInterface

Class	**PortInterfaceToDataTypeMapping**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::PortInterface			
Note	This meta-class represents the ability to associate a PortInterface with a DataTypeMappingSet. This association is needed for the generation of header files in the scope of a single PortInterface. The association is intentionally made outside the scope of the PortInterface itself because the designers of a PortInterface most likely will not want to add details about the level of ImplementationDataType. **Tags:** atp.Status=draft atp.recommendedPackage=PortInterfaceToDataTypeMappings			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
dataType MappingSet	DataTypeMappingSet	1..*	ref	This represents the reference to the applicable data TypemappingSet **Tags:** atp.Status=draft atp.StatusComment=Reserved for adaptive platform
portInterface	PortInterface	1	ref	This represents the reference to the applicable Port Interface **Tags:** atp.Status=draft atp.StatusComment=Reserved for adaptive platform

Table A.27: PortInterfaceToDataTypeMapping

Class	***Referrable*** (abstract)			
Package	M2::AUTOSARTemplates::GenericStructure::GeneralTemplateClasses::Identifiable			
Note	Instances of this class can be referred to by their identifier (while adhering to namespace borders).			
Base	*ARObject*			
Subclasses	*AtpDefinition*, BswDistinguishedPartition, *BswModuleCallPoint*, BswModuleClientServerEntry, BswVariableAccess, CouplingPortTrafficClassAssignment, *CppImplementationDataTypeContextTarget*, *DiagnosticEnvModeElement*, EthernetPriorityRegeneration, ExclusiveAreaNestingOrder, *HwDescriptionEntity*, *ImplementationProps*, ModeTransition, *MultilanguageReferrable*, NmNetworkHandle, PncMappingIdent, *SingleLanguageReferrable*, SoConIPduIdentifier, SocketConnectionBundle, SomeipRequiredEventGroup, TimeSyncServerConfiguration, TpConnectionIdent			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
shortName	Identifier	1	attr	This specifies an identifying shortName for the object. It needs to be unique within its context and is intended for humans but even more for technical reference. **Stereotypes:** atpIdentityContributor **Tags:** xml.enforceMinMultiplicity=true xml.sequenceOffset=-100
shortNameFragment	ShortNameFragment	*	aggr	This specifies how the Referrable.shortName is composed of several shortNameFragments. **Tags:**xml.sequenceOffset=-90

Table A.28: Referrable

Class	**ServiceInterface**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::PortInterface			

Note	This represents the ability to define a PortInterface that consists of a heterogeneous collection of methods, events and fields.
	Tags:
	atp.Status=draft
	atp.recommendedPackage=ServiceInterfaces			

Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *PortInterface*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***

event	VariableDataPrototype	*	aggr	This represents the collection of events defined in the context of a ServiceInterface.
				Stereotypes: atpVariation
				Tags:
				atp.Status=draft
				vh.latestBindingTime=blueprintDerivationTime
				xml.sequenceOffset=30

field	Field	*	aggr	This represents the collection of fields defined in the context of a ServiceInterface.
				Stereotypes: atpVariation
				Tags:
				atp.Status=draft
				vh.latestBindingTime=blueprintDerivationTime
				xml.sequenceOffset=40

majorVersion	PositiveInteger	0..1	attr	Major version of the service contract.
				Tags:
				atp.Status=draft
				xml.sequenceOffset=10

▽

△

Class	**ServiceInterface**			

method	ClientServerOperation	*	aggr	This represents the collection of methods defined in the context of a ServiceInterface.
				Stereotypes: atpVariation
				Tags:
				atp.Status=draft
				vh.latestBindingTime=blueprintDerivationTime
				xml.sequenceOffset=50

minorVersion	PositiveInteger	0..1	attr	Minor version of the service contract.
				Tags:
				atp.Status=draft
				xml.sequenceOffset=20

trigger	Trigger	*	aggr	This represents the collection of triggers defined in the context of a ServiceInterface.
				Stereotypes: atpVariation
				Tags:
				atp.Status=draft
				vh.latestBindingTime=blueprintDerivationTime
				xml.sequenceOffset=60

Table A.29: ServiceInterface

Class	**StdCppImplementationDataType**			
Package	M2::AUTOSARTemplates::AdaptivePlatform::ApplicationDesign::CppImplementationDataType			

Note	This meta-class represents the way to specify a data type definition that is taken as the basis for a C++ language binding to a C++ Standard Library feature.
	Tags:
	atp.Status=draft
	atp.recommendedPackage=CppImplementationDataTypes			

Base	*ARElement*, *ARObject*, *AbstractImplementationDataType*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *AutosarDataType*, *CollectableElement*, *CppImplementationDataType*, *CppImplementationDataTypeContextTarget*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
–	–	–	–	–

Table A.30: StdCppImplementationDataType

Class	<<atpVariation>> **SwDataDefProps**
Package	M2::MSR::DataDictionary::DataDefProperties
Note	This class is a collection of properties relevant for data objects under various aspects. One could consider this class as a "pattern of inheritance by aggregation". The properties can be applied to all objects of all classes in which SwDataDefProps is aggregated. Note that not all of the attributes or associated elements are useful all of the time. Hence, the process definition (e.g. expressed with an OCL or a Document Control Instance MSR-DCI) has the task of implementing limitations. SwDataDefProps covers various aspects: • Structure of the data element for calibration use cases: is it a single value, a curve, or a map, but also the recordLayouts which specify how such elements are mapped/converted to the DataTypes in the programming language (or in AUTOSAR). This is mainly expressed by properties like swRecordLayout and swCalprmAxisSet • Implementation aspects, mainly expressed by swImplPolicy, swVariableAccessImplPolicy, swAddrMethod, swPointerTagetProps, baseType, implementationDataType and additionalNativeTypeQualifier ▽

▽

△

Class	<<atpVariation>> **SwDataDefProps**			
	△ • Access policy for the MCD system, mainly expressed by swCalibrationAccess • Semantics of the data element, mainly expressed by compuMethod and/or unit, dataConstr, invalidValue • Code generation policy provided by swRecordLayout **Tags:**vh.latestBindingTime=codeGenerationTime			
Base	*ARObject*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
additionalNativeTypeQualifier	NativeDeclarationString	0..1	attr	This attribute is used to declare native qualifiers of the programming language which can neither be deduced from the baseType (e.g. because the data object describes a pointer) nor from other more abstract attributes. Examples are qualifiers like "volatile", "strict" or "enum" of the C-language. All such declarations have to be put into one string. **Tags:**xml.sequenceOffset=235

annotation	Annotation	*	aggr	This aggregation allows to add annotations (yellow pads ...) related to the current data object.
				Tags:
				xml.roleElement=true
				xml.roleWrapperElement=true
				xml.sequenceOffset=20
				xml.typeElement=false
				xml.typeWrapperElement=false

baseType	SwBaseType	0..1	ref	Base type associated with the containing data object. **Tags:**xml.sequenceOffset=50
compuMethod	CompuMethod	0..1	ref	Computation method associated with the semantics of this data object. **Tags:**xml.sequenceOffset=180
dataConstr	DataConstr	0..1	ref	Data constraint for this data object. **Tags:**xml.sequenceOffset=190
displayFormat	DisplayFormatString	0..1	attr	This property describes how a number is to be rendered e.g. in documents or in a measurement and calibration system. **Tags:**xml.sequenceOffset=210
display Presentation	DisplayPresentation Enum	0..1	attr	This attribute controls the presentation of the related data for measurement and calibration tools.
implementation DataType	AbstractImplementation DataType	0..1	ref	This association denotes the ImplementationDataType of a data declaration via its aggregated SwDataDefProps. It is used whenever a data declaration is not directly referring to a base type. Especially • redefinition of an ImplementationDataType via a "typedef" to another ImplementationDatatype • the target type of a pointer (see SwPointerTargetProps), if it does not refer to a base type directly • the data type of an array or record element within an ImplementationDataType, if it does not refer to a base type directly • the data type of an SwServiceArg, if it does not refer to a base type directly **Tags:**xml.sequenceOffset=215

▽

△

Class	<<atpVariation>> **SwDataDefProps**			
invalidValue	ValueSpecification	0..1	aggr	Optional value to express invalidity of the actual data element. **Tags:**xml.sequenceOffset=255
stepSize	Float	0..1	attr	This attribute can be used to define a value which is added to or subtracted from the value of a DataPrototype when using up/down keys while calibrating.
swAddrMethod	SwAddrMethod	0..1	ref	Addressing method related to this data object. Via an association to the same SwAddrMethod it can be specified that several DataPrototypes shall be located in the same memory without already specifying the memory section itself. **Tags:**xml.sequenceOffset=30
swAlignment	AlignmentType	0..1	attr	The attribute describes the intended typical alignment of the DataPrototype. If the attribute is not defined the alignment is determined by the swBaseType size and the memoryAllocationKeywordPolicy of the referenced Sw AddrMethod. **Tags:**xml.sequenceOffset=33
swBit Representation	SwBitRepresentation	0..1	aggr	Description of the binary representation in case of a bit variable. **Tags:**xml.sequenceOffset=60
swCalibration Access	SwCalibrationAccess Enum	0..1	attr	Specifies the read or write access by MCD tools for this data object. **Tags:**xml.sequenceOffset=70
swCalprmAxis Set	SwCalprmAxisSet	0..1	aggr	This specifies the properties of the axes in case of a curve or map etc. This is mainly applicable to calibration parameters. **Tags:**xml.sequenceOffset=90

swComparison Variable	SwVariableRefProxy	*	aggr	Variables used for comparison in an MCD process.
				Tags:
				xml.sequenceOffset=170
				xml.typeElement=false

swData Dependency	SwDataDependency	0..1	aggr	Describes how the value of the data object has to be calculated from the value of another data object (by the MCD system). **Tags:**xml.sequenceOffset=200

swHostVariable	SwVariableRefProxy	0..1	aggr	Contains a reference to a variable which serves as a host-variable for a bit variable. Only applicable to bit objects.
				Tags:
				xml.sequenceOffset=220
				xml.typeElement=false

swImplPolicy	SwImplPolicyEnum	0..1	attr	Implementation policy for this data object. **Tags:**xml.sequenceOffset=230
swIntended Resolution	Numerical	0..1	attr	The purpose of this element is to describe the requested quantization of data objects early on in the design process. The resolution ultimately occurs via the conversion formula present (compuMethod), which specifies the transition from the physical world to the standardized world (and vice-versa) (here, "the slope per bit" is present implicitly in the conversion formula). ▽

▽

△

Class	<<atpVariation>> **SwDataDefProps**			
				△ In the case of a development phase without a fixed conversion formula, a pre-specification can occur through swIntendedResolution. The resolution is specified in the physical domain according to the property "unit". **Tags:**xml.sequenceOffset=240
swInterpolation Method	Identifier	0..1	attr	This is a keyword identifying the mathematical method to be applied for interpolation. The keyword needs to be related to the interpolation routine which needs to be invoked. **Tags:**xml.sequenceOffset=250
swIsVirtual	Boolean	0..1	attr	This element distinguishes virtual objects. Virtual objects do not appear in the memory, their derivation is much more dependent on other objects and hence they shall have a swDataDependency . **Tags:**xml.sequenceOffset=260
swPointerTarget Props	SwPointerTargetProps	0..1	aggr	Specifies that the containing data object is a pointer to another data object. **Tags:**xml.sequenceOffset=280
swRecord Layout	SwRecordLayout	0..1	ref	Record layout for this data object. **Tags:**xml.sequenceOffset=290
swRefresh Timing	MultidimensionalTime	0..1	aggr	This element specifies the frequency in which the object involved shall be or is called or calculated. This timing can be collected from the task in which write access processes to the variable run. But this cannot be done by the MCD system. So this attribute can be used in an early phase to express the desired refresh timing and later on to specify the real refresh timing. **Tags:**xml.sequenceOffset=300
swTextProps	SwTextProps	0..1	aggr	the specific properties if the data object is a text object. **Tags:**xml.sequenceOffset=120
swValueBlock Size	Numerical	0..1	attr	This represents the size of a Value Block **Stereotypes:** atpVariation **Tags:** vh.latestBindingTime=preCompileTime xml.sequenceOffset=80
swValueBlock SizeMult (ordered)	Numerical	*	attr	This attribute is used to specify the dimensions of a value block (VAL_BLK) for the case that that value block has more than one dimension. The dimensions given in this attribute are ordered such that the first entry represents the first dimension, the second entry represents the second dimension, and so on. For one-dimensional value blocks the attribute swValue BlockSize shall be used and this attribute shall not exist. **Stereotypes:** atpVariation **Tags:**vh.latestBindingTime=preCompileTime

▽

△

Class	<<atpVariation>> **SwDataDefProps**			
unit	Unit	0..1	ref	Physical unit associated with the semantics of this data object. This attribute applies if no compuMethod is specified. If both units (this as well as via compuMethod) are specified the units shall be compatible. **Tags:**xml.sequenceOffset=350
valueAxisDataType	ApplicationPrimitiveDataType	0..1	ref	The referenced ApplicationPrimitiveDataType represents the primitive data type of the value axis within a compound primitive (e.g. curve, map). It supersedes CompuMethod, Unit, and BaseType. **Tags:**xml.sequenceOffset=355

Table A.31: SwDataDefProps

Class	**SymbolProps**			
Package	M2::AUTOSARTemplates::SWComponentTemplate::Components			
Note	This meta-class represents the ability to contribute a part of a namespace.			
Base	*ARObject*, *ImplementationProps*, *Referrable*			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
–	–	–	–	–

Table A.32: SymbolProps

B History of Specification Items

B.1 Specification Item evolution compared to AUTOSAR R20-11

In previous AUTOSAR releases, the content of this specification was incorporated in [1] chapter *"Communication Payload Data Types"*. In AUTOSAR release R21-11, AUTOSAR has decided that the serialization rules of transforming AP modeled data types to implementation language bound data types are not cardinal to Communication scenarios, i.e. usage within a ServiceInterface, rather, they should be available to **any** sub-class of PortInterface used in the AP.

This section therefore defines the mapping of those Specification Item identifiers previously present in [1] in AUTOSAR release R20-11, to the corresponding newly introduced Specification Item identifiers in this document in AUTOSAR release R21-11 and thereafter.

It is paramount that i) specifications referring to, and ii) code bases implementing those Specification Item identifiers in [1] chapter *"Communication Payload Data Types"* in AUTOSAR release R20-11 can trace these to the *new* Specification Item identifiers in this document.

Specification Item identifier (current)	**Specification Item identifier (R20-11)**
[SWS_LBAP_00001]	[SWS_CM_00423]
[SWS_LBAP_00002]	[SWS_CM_00421]
[SWS_LBAP_00003]	[SWS_CM_00411]
[SWS_LBAP_00004]	[SWS_CM_00400]
[SWS_LBAP_00005]	[SWS_CM_00504]
[SWS_LBAP_00006]	[SWS_CM_00402]
[SWS_LBAP_00007]	[SWS_CM_00403]
[SWS_LBAP_00008]	[SWS_CM_00404]
[SWS_LBAP_00009]	[SWS_CM_00502]
[SWS_LBAP_00010]	[SWS_CM_00405]
[SWS_LBAP_00011]	[SWS_CM_00414]
[SWS_LBAP_00012]	[SWS_CM_01032]
[SWS_LBAP_00013]	[SWS_CM_00449]
[SWS_LBAP_00014]	[SWS_CM_00508]
[SWS_LBAP_00015]	[SWS_CM_00406]
[SWS_LBAP_00016]	[SWS_CM_00509]
[SWS_LBAP_00017]	[SWS_CM_00407]
[SWS_LBAP_00018]	[SWS_CM_00503]
[SWS_LBAP_00019]	[SWS_CM_00408]
[SWS_LBAP_00020]	[SWS_CM_00452]
[SWS_LBAP_00021]	[SWS_CM_00450]

▽

△

Specification Item identifier (current)	Specification Item identifier (R20-11)
[SWS_LBAP_00022]	[SWS_CM_00507]
[SWS_LBAP_00023]	[SWS_CM_00409]
[SWS_LBAP_00024]	[SWS_CM_00505]
[SWS_LBAP_00025]	[SWS_CM_00506]
[SWS_LBAP_00026]	[SWS_CM_00410]
[SWS_LBAP_00027]	[SWS_CM_00424]
[SWS_LBAP_00028]	[SWS_CM_00425]
[SWS_LBAP_00029]	[SWS_CM_10376]
[SWS_LBAP_00030]	[SWS_CM_00426]
[SWS_LBAP_00031]	[SWS_CM_10409]
[SWS_LBAP_00033]	[SWS_CM_10373]
[SWS_LBAP_00034]	[SWS_CM_01020], ([SWS_CM_12000][1])
[SWS_LBAP_00035]	[SWS_CM_10375]
[SWS_LBAP_00038]	[SWS_CM_00506]

Table B.1: Specification Item evolution table

B.2 Specification Item history of this document according to AUTOSAR R21-11

Please note that the lists in this chapter also include specification items that have been removed from the specification in a later version. These specification items do not appear as hyperlinks in the document.

B.2.1 Added Traceables in R21-11

Number	Heading
[SWS_LBAP_00001]	ARA generator rejection of unmapped data types
[SWS_LBAP_00002]	`ARA Language Binding Generator` usage of `typeEmitter`
[SWS_LBAP_00003]	ARA generator rejection of symbol clashes
[SWS_LBAP_00004]	Naming of data types by `shortName`
[SWS_LBAP_00005]	Standardized `Primitive CppImplementationDataTypes`s
[SWS_LBAP_00006]	`Primitive CppImplementationDataType` fixed width integers

▽

[1] Newly added in R21-11

△

Number	Heading
[SWS_LBAP_00007]	StdCppImplementationDataType of category=ARRAY with one dimension
[SWS_LBAP_00008]	StdCppImplementationDataType of category=ARRAY with multiple dimensions
[SWS_LBAP_00009]	CustomCppImplementationDataType of category=ARRAY
[SWS_LBAP_00010]	StdCppImplementationDataType of category=STRUCTURE
[SWS_LBAP_00011]	Structure element specification typed by CppImplementationDataType
[SWS_LBAP_00012]	Accessing optional record elements inside a Structure CppImplementationDataType that are serialized with the Tag-Length-Value principle.
[SWS_LBAP_00013]	StdCppImplementationDataType of category=VARIANT
[SWS_LBAP_00014]	CustomCppImplementationDataType of category=VARIANT
[SWS_LBAP_00015]	StdCppImplementationDataType of category=STRING without Allocator
[SWS_LBAP_00016]	StdCppImplementationDataType of category=STRING with Allocator
[SWS_LBAP_00017]	StdCppImplementationDataType of category=VECTOR with one dimension, without Allocator
[SWS_LBAP_00018]	StdCppImplementationDataType of category=VECTOR with one dimension, with Allocator
[SWS_LBAP_00019]	StdCppImplementationDataType of category=VECTOR with multiple dimensions
[SWS_LBAP_00020]	CppImplementationDataType with category=VECTOR size semantics
[SWS_LBAP_00021]	Imposing memory limits with Allocator
[SWS_LBAP_00022]	CustomCppImplementationDataType of category=VECTOR
[SWS_LBAP_00023]	StdCppImplementationDataType with category=ASSOCIATIVE_MAP without an Allocator
[SWS_LBAP_00024]	StdCppImplementationDataType with category=ASSOCIATIVE_MAP with an Allocator
[SWS_LBAP_00025]	CustomCppImplementationDataType of category=ASSOCIATIVE_MAP without Allocator
[SWS_LBAP_00026]	StdCppImplementationDataType of category=TYPE_REFERENCE
[SWS_LBAP_00027]	Enumeration Data Type
[SWS_LBAP_00028]	Enumeration Data Type - enumerators
[SWS_LBAP_00029]	Enumeration Data Type - skip CompuScales with non-point range
[SWS_LBAP_00030]	ARA generator rejection of incomplete Enumeration Data Types
[SWS_LBAP_00031]	Scale Linear And Texttable Data Type
[SWS_LBAP_00032]	CppImplementationTypes Header Files artifact generation
[SWS_LBAP_00033]	CppImplementationTypes Header Files file names
[SWS_LBAP_00034]	CppImplementationTypes Header Files directory names
[SWS_LBAP_00035]	CppImplementationTypes Header Files namespace hierarchy

▽

△

Number	Heading
[SWS_LBAP_00036]	CppImplementationTypes Header Files multiple inclusion guard
[SWS_LBAP_00037]	Principle of an ARA Language Binding Generator
[SWS_LBAP_00038]	CustomCppImplementationDataType of category=ASSOCIATIVE_MAP with Allocator

Table B.2: Added Traceables in R21-11

B.2.2 Changed Traceables in R21-11

none

B.2.3 Deleted Traceables in R21-11

none