Document Title	Specification of ECU Configuration
Document Owner	AUTOSAR GbR
Document Responsibility	AUTOSAR GbR
Document Identification No	087
Document Classification	Standard

Document Version	2.2.0
Document Status	Final
Part of Release	3.0
Revision	0006

Document Change History

Date	Version	Changed by	Description
14.10.2009	2.2.0	AUTOSAR Administration	Updated definition how symbolic names are generated from the EcuC.
15.09.2008	2.1.0	AUTOSAR Administration	Fixed foreign reference to PduToFrameMapping
01.02.2008	2.0.1	AUTOSAR Administration	Added reference from Container to ContainerDef. Removed reference from Container to ParamConfContainerDef.
06.12.2007	2.0.0	AUTOSAR Administration	• Changed representation of a ChoiceContainerDef in an ECU Configuration Description • Moved sections from ”ECU Configuration Parameter Definition” into the ”Specification of ECU Configuration” (COM-Stack Configuration Patterns) • Updated interaction of ECU Configuration with BSW Module Description • Added specification items which define what is allowed when creating a Vendor Specific Module Definition (VSMD)

			• Correction of ”InstanceParamRef” definition in ECU Configuration Specification • Refined the available character set of calculationFormula • Added clarification about the usage of ADMIN-DATA to track version information • Document meta information extended • Small layout adaptations made
31.01.2007	1.1.1	AUTOSAR Administration	• “Advice for users” revised • Legal disclaimer revised
06.12.2006	1.1.0	AUTOSAR Administration	• Methodology chapter revised (incl. introduction of support for AUTOSAR Services) • Added EcucElement, EcuSwComposition, configuration class affection, LinkerSymbolDef and LinkerSymbolValue to the metamodel • Support for multiple configuration sets added • Legal disclaimer revised
28.06.2006	1.0.1	AUTOSAR Administration	Layout Adaptations
09.05.2006	1.0.0	AUTOSAR Administration	Initial Release

Table of Contents

Bibliography

[1] Methodology
AUTOSAR_Methodology.pdf

[2] Glossary
AUTOSAR_Glossary.pdf

[3] Requirements on ECU Configuration
AUTOSAR_RS_ECU_Configuration.pdf

[4] General Requirements on Basic Software Modules
AUTOSAR_SRS_General.pdf

[5] Requirements on Basic Software Module Description Template
AUTOSAR_RS_BSW_ModuleDescription.pdf

[6] Specification of ECU Configuration Parameters
AUTOSAR_ECU_ConfigationParameters.pdf

[7] Software Component Template
AUTOSAR_SoftwareComponentTemplate.pdf

[8] Model Persistence Rules for XML
AUTOSAR_ModelPersistenceRulesXML.pdf

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

[10] System Template
AUTOSAR_SystemTemplate.pdf

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

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

[13] Specification of ECU Configuration Parameters (XML)
AUTOSAR_EcucParamDef.arxml

[14] IEEE standard for radix-independent floating-point arithmetic
(ANSI/IEEE Std 854-1987)

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

1 Introduction

According to AUTOSAR Methodology the configuration process contains 4 steps that will be discussed in chapter 2 in more detail:

- `Configure System`
- `Extract ECU-Specific Information`
- `Configure ECU`
- `Generate Executable`

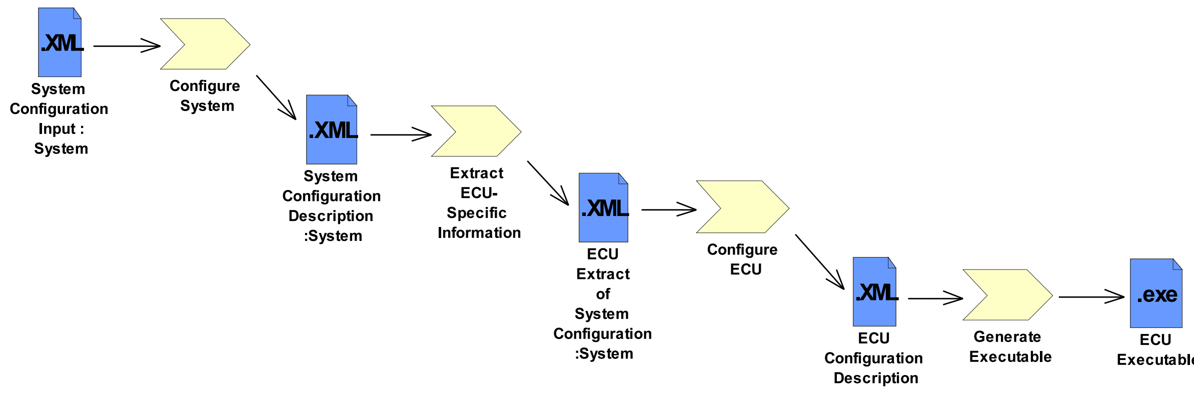
.XML
System Configuration Input : System
Configure System
.XML
System Configuration Description :System
Extract ECU-Specific Information
.XML
ECU Extract of System Configuration :System
Configure ECU
.XML
ECU Configuration Description
Generate Executable
.exe
ECU Executable

Figure 1.1: AUTOSAR Methodology Overview (from [1])

The configuration process of an ECU starts with the splitting of the System Description into several descriptions, whereas each contains all information about one single ECU. This ECU extract is the basis for the ECU Configuration step.

Within the ECU Configuration process each single module of the AUTOSAR Architecture can be configured for the special needs of this ECU. Because of a quite complex AUTOSAR Architecture, modules and interdependencies between the modules, tool-support is required: AUTOSAR ECU Configuration Editor(s).

The tool strategy and tooling details for the ECU Configuration are out of scope of this specification. Nevertheless tools need the knowledge about ECU Configuration Parameters and their constraints such as configuration class, value range, multiplicities etc. This description is the input for the tools. The description of configuration parameters is called ECU Configuration Parameter Definition and described in detail in this specification (chapter 3.3).

To make sure, that all tools are using the same output-format within the configured values of the parameters, the ECU Configuration Description is also part of this specification and described in detail later on (chapter 3.4). The ECU Configuration Description may be on one hand the input format for other configuration tools (within a tool-chain of several configuration editors) and on the other hand it is the basis of generators. The configured parameters are generated into ECU executables. This is the last step of the configuration process and again out of scope of this specification.

1.1 Abbreviations

This section describes abbreviations that are specific to the ECU Configuration Specification and that are not part of the official AUTOSAR Glossary [2].

Following abbreviations are mentioned that are specifically used in this specification:

ECUC	ECU Configuration
ECUC Description	ECU Configuration Description
ECUC ParamDef	ECU Configuration Parameter Definition
ECUC Value	ECU Configuration Value
StMD	Standardized Module Definition
VSMD	Vendor Specific Module Definition

1.2 Requirements Traceability

Following table references the requirements specified in AUTOSAR ECU Configuration Requirements [3] and in General Requirements on Basic Software Modules [4] and links to the ”ecuc_sws” fulfillments of these.

Requirement	Description	Satisfied by
[BSW00344]	Reference to link time configuration	not applicable (BSW implementation issue)
[BSW00345]	Pre-compile time configuration	not applicable (BSW implementation issue)
[BSW00380]	Separate C-File for configuration parameters	not applicable (BSW implementation issue)
[BSW00381]	Separate configuration header file for pre-compile time parameters	not applicable (BSW implementation issue)
[BSW00382]	Not-used configuration elements need to be listed	not applicable (requirement on the BSW Module Description [5])
[BSW00383]	List dependencies of configuration files	not applicable (requirement on the BSW Module Description [5])
[BSW00384]	List dependencies to other modules	not applicable (requirement on the BSW Module Description [5])
[BSW00385]	List possible error notifications	not applicable (requirement on the BSW Module Description [5])
[BSW00386]	Configuration for detecting an error	not applicable (requirement on ECU Configuration Parameters [6] and BSW Module Description [5])
[BSW00387]	Specify the configuration class of callback function	[ecuc_sws_2016]
[BSW00388]	Introduce containers	[ecuc_sws_2006]
[BSW00389]	Containers shall have names	[ecuc_sws_2043]
[BSW00390]	Parameter content shall be unique within the module	not applicable (requirement on the BSW SWS)

Requirement	Description	Satisfied by
[BSW00391]	Parameters shall have unique names	[ecuc_sws_2043] [ecuc_sws_2014]
[BSW00392]	Parameters shall have a type	[ecuc_sws_2014]
[BSW00393]	Parameters shall have a range	[ecuc_sws_2027] [ecuc_sws_2028]
[BSW00394]	Specify the scope of the parameters	not applicable (requirement on the BSW SWS)
[BSW00395]	List the required parameters (per parameter)	[ecuc_sws_2039]
[BSW00396]	Configuration classes	[ecuc_sws_2016]
[BSW00397]	Pre-compile time parameters	[ecuc_sws_2017] [ecuc_sws_1031]
[BSW00398]	Link time parameters	[ecuc_sws_2018] [ecuc_sws_1032]
[BSW00399]	Loadable Post-build time parameters	[ecuc_sws_4006] [ecuc_sws_4000] [ecuc_sws_4005]
[BSW00400]	Selectable Post-build time parameters	[ecuc_sws_4007]
[BSW00401]	Documentation of multiple instances of configuration parameters	not applicable (requirement on the BSW SWS)
[BSW00402]	Published information	not applicable (requirement on the BSW Module Description [5])
[BSW00404]	Reference to post-build time configuration	not applicable (BSW implementation issue)
[BSW00405]	Reference to multiple configuration sets	not applicable (BSW implementation issue)
[BSW00408]	Configuration parameter naming convention	requirement on ECU Configuration Parameters [6]
[BSW00410]	Compiler switches shall have defined values	not applicable (BSW implementation issue)
[BSW00411]	Get version info keyword	not applicable (BSW implementation issue)
[BSW00412]	Separate H-File for configuration parameters	not applicable (BSW implementation issue)
[BSW00413]	Accessing instances of BSW modules	requirement on ECU Configuration Parameters [6]
[BSW00414]	Parameter of init function	not applicable (BSW implementation issue)
[BSW00415]	User dependent include files	not applicable (BSW implementation issue)
[BSW00416]	Sequence of Initialization	requirement on the BSW SWS
[BSW00417]	Reporting of Error Events by Non-Basic Software	not applicable (non-BSW implementation issue)
[BSW00419]	Separate C-Files for pre-compile time configuration parameters	not applicable (BSW implementation issue)
[BSW00420]	Production relevant error event rate detection	not applicable (requirement on the BSW SWS)
[BSW00421]	Reporting of production relevant error events	not applicable (requirement on the BSW SWS and BSW Module Description [5])
[BSW00422]	Debouncing of production relevant error status	not applicable (requirement on the BSW SWS)
[BSW00423]	Usage of SW-C template to describe BSW modules with AUTOSAR Interfaces	not applicable (requirement on BSW Module Description [5])
[BSW00424]	BSW main processing function task allocation	not applicable (requirement on BSW scheduler module)

Requirement	**Description**	**Satisfied by**
[BSW00425]	Trigger conditions for schedulable objects	not applicable (requirement on BSW Module Description [5])
[BSW00426]	Exclusive areas in BSW modules	not applicable (requirement on BSW Module Description [5] and BSW implementation issue)
[BSW00427]	ISR description for BSW modules	not applicable (requirement on BSW Module Description [5])
[BSW00428]	Execution order dependencies of main processing functions	not applicable (requirement on BSW Module Description [5])
[BSW00429]	Restricted BSW OS functionality access	not applicable (BSW implementation issue)
[BSW00431]	The BSW Scheduler module implements task bodies	not applicable (requirement on BSW scheduler module)
[BSW00432]	Modules should have separate main processing functions for read/receive and write/transmit data path	not applicable (BSW implementation issue)
[BSW00433]	Calling of main processing functions	not applicable (BSW implementation issue)
[BSW00434]	The Schedule Module shall provide an API for exclusive areas	not applicable (BSW implementation issue)
[BSW159]	Tool-based configuration	[ecuc_sws_1030] [ecuc_sws_1000]
[BSW167]	Static configuration checking	[ecuc_sws_1000]
[BSW170]	Data for reconfiguration of AUTOSAR SW-Components	not applicable (requirement on the Software-Component Template [7])
[BSW171]	Configurability of optional functionality	[ecuc_sws_2009] [ecuc_sws_1000] [ecuc_sws_1002] [ecuc_sws_1010] [ecuc_sws_1012]
[ECUC0002]	Support of vendor-specific ECU Configuration Parameters	[ecuc_sws_1029] [ecuc_sws_1001] [ecuc_sws_1002] [ecuc_sws_1011] [ecuc_sws_1013] [ecuc_sws_1014] [ecuc_sws_1015] [ecuc_sws_5001] [ecuc_sws_5002] [ecuc_sws_5003]
[ECUC0008]	Definition of post-build time changeable configuration of BSW	[ecuc_sws_2019] [ecuc_sws_4006] [ecuc_sws_4000]
[ECUC0012]	One description mechanism for different configuration classes	[ecuc_sws_2016]
[ECUC0015]	Configuration of multiple instances of BSW modules	[ecuc_sws_2059] [ecuc_sws_2008]
[ECUC0016]	Execution order of runnable entities	not applicable (requirement on ECU Configuration Parameters [6])
[ECUC0018]	Extension handling	fulfilled by the Model Persistence Rules for XML [8]
[ECUC0021]	Select Application SW Component and BSW module implementation	[ecuc_sws_1036] [ecuc_sws_1029]
[ECUC0025]	Compatible with iterative design	[ecuc_sws_1027] [ecuc_sws_4000]

Requirement	**Description**	**Satisfied by**
[ECUC0029]	Identify mechanisms not criteria	proceeding was followed in the development of the ECU Configuration Specification
[ECUC0030]	Clarify configuration terminology	terms defined in AUTOSAR Glossary [2]
[ECUC0032]	ECU Configuration Description shall be the root for the whole configuration information of an ECU	[ecuc_sws_2003] [ecuc_sws_1029]
[ECUC0039]	Support configuration of BSW	requirement on ECU Configuration Parameters [6] [ecuc_sws_1029]
[ECUC0040]	Support configuration of RTE	requirement on ECU Configuration Parameters [6] [ecuc_sws_1029]
[ECUC0041]	Support AUTOSAR SW Component Integration	requirement on ECU Configuration Parameters [6]
[ECUC0043]	Duplication free description	[ecuc_sws_1027] [ecuc_sws_1028] [ecuc_sws_1031]
[ECUC0046]	Support definition of configuration class	[ecuc_sws_2016]
[ECUC0047]	Pre-compile time configuration of BSW	[ecuc_sws_2017] [ecuc_sws_1031]
[ECUC0048]	Link time configuration of BSW	[ecuc_sws_2018] [ecuc_sws_1032]
[ECUC0049]	ECU Configuration description shall be tool processable	[ecuc_sws_2001] [ecuc_sws_1030]
[ECUC0050]	Specify ECU Configuration Parameter Definition	[ecuc_sws_2045]
[ECUC0053]	Support for multiple configuration sets	not applicable (feature 128 canceled)
[ECUC0055]	Support mandatory and optional configuration parameters	[ecuc_sws_3011] [ecuc_sws_3010] [ecuc_sws_3030] [ecuc_sws_2009] [ecuc_sws_1002]
[ECUC0065]	Development according to the AUTOSAR Metamodeling Guide	[ecuc_sws_2000]
[ECUC0066]	Transformation of ECUC modeling according to the AUTOSAR Model Persistence Rules for XML	[ecuc_sws_2001]
[ECUC0070]	Support mandatory and optional containers	[ecuc_sws_2009] [ecuc_sws_1003]
[ECUC0071]	Support for Generic Configuration Editor	[ecuc_sws_1031]
[ECUC0072]	Support for referencing from dependent containers	[ecuc_sws_3027] [ecuc_sws_3033] [ecuc_sws_2039]
[ECUC0073]	Support Service Configuration of AUTOSAR SW Components	postponed; no concept available yet
[ECUC0074]	Support Sequential ECU Configuration	[ecuc_sws_1031]

2 ECU Configuration Methodology

ECU Configuration is one step in the overall AUTOSAR methodology, which is described in [1]. Figure 1.1 already introduced in chapter 1 is taken from that document and shows the most abstract view of the methodology. In this document, the activities regarding configuring an ECU and generate the configuration data will be defined in more detail than provided in [1]. To understand this chapter, the reader should be familiar with [1].

2.1 Notation used

Figure 1.1 and all other figures taken from the AUTOSAR Methodology [1] use a formal notation called SPEM (Software Process Engineering Meta-Model), explained in detail in [1]. The SPEM elements used in this document are

- *Work products* (blue document shaped elements),
- *Activities* (block arrows) and
- *Guidances* (set square) to depict tools, attached to the activity they support by a dashed line.

The flow of work products is depicted by solid lines with arrow heads, pointing in the direction of the work flow. Dependencies are depicted by dashed lines with arrow heads, pointing from the dependent element to the element it depends on. Compositions are depicted by a solid line with a solid diamond on the end of the aggregating element.

2.2 Inputs to ECU Configuration

[ecuc_sws_1036] ECU Configuration has two input sources. First of all, all configuration that must be agreed across ECUs is defined in the System Configuration, which results in a `System Configuration Description` (and the resulting `ECU Extract of the System Configuration` for the individual ECUs). Secondly, the ECU BSW is built using BSW modules. The specifics of these module implementation are defined in the `BSW Module Descriptions` (not shown in figure 1.1, see figure 2.7). The latter is described in [9] in more detail. The concept of the ECU extract is depicted below:

ECU Extract of System Description

ECU Configuration can only be started once a plausible `System Configuration Description` and the corresponding ECU extract has been generated (see figure 1.1). Details on the `System Configuration Description` can be found in [10]. The `System Configuration Description` contains all relevant system-wide configuration, such as

- ECUs present in the system
- Communication systems interconnecting those ECUs and their configuration
- Communication matrices (frames sent and received) for those communication systems
- Definition of Software Components with their ports and interfaces and connections (defined in the SWC Description and referenced in the `System Configuration Description`).
- Mapping of SWCs to ECUs

The `ECU Extract of the System Configuration` is a description in the same format as the `System Configuration Description`, but with only those elements included that are relevant for the configuration of one specific ECU.

2.3 ECU Configuration

ECU Configuration can be broken down into three activities, as shown in figure 2.1:

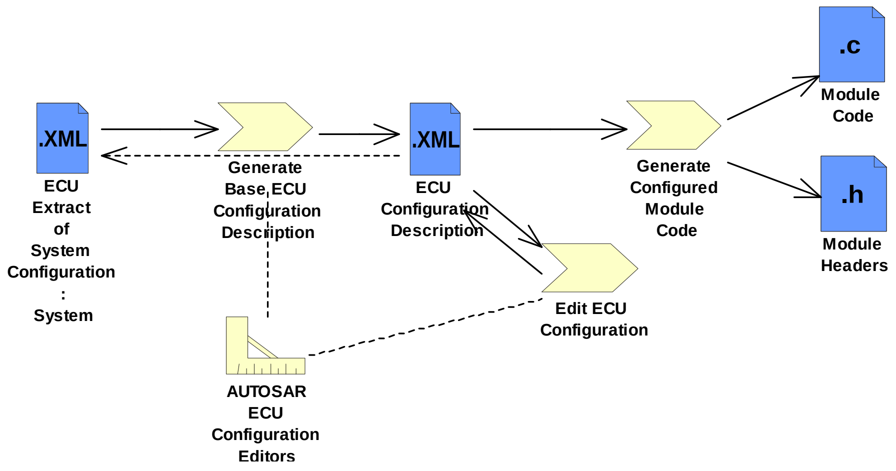
.XML
ECU Extract of System Configuration : System
Generate Base ECU Configuration Description
.XML
ECU Configuration Description
AUTOSAR ECU Configuration Editors
Edit ECU Configuration
Generate Configured Module Code
.c
Module Code
.h
Module Headers

Figure 2.1: ECU Configuration broken down into three sub-activities

- `Generate Base ECU Configuration Description`
- `Edit ECU Configuration`
- `Generate Configured Module Code`

[ecuc_sws_1027] All three activities use a single work product, the `ECU Configuration Description`, which contains (i.e. references) all the configuration information for all BSW modules on the ECU. In order to better understand the three different activities an introduction to configuration classes is essential. In a real implementation of a BSW module all configuration parameters are most likely not the same configuration class. I.e it will be a mix of parameters with different configuration classes within a BSW module.

These three activities are introduced in detail in later sections, but first is an introduction to ECU configuration description and configuration classes.

2.3.1 ECU Configuration Description

The `ECU Extract of System Configuration` only defines the configuration elements that must be agreed between ECUs. In order to generate a working executable that runs on the ECU, much more configuration information must be provided.

The remaining part of the configuration is about configuring all BSW modules within the ECU. Typical BSW modules within an ECU can be: RTE, Com, Can, OS, NVRAM etc. There are also dependencies between BSW modules to consider when configuring the ECU. When the configuration is done, the generation of configuration data takes place. I.e. there are both configuration editors and configuration generators involved in the process. In order to obtain consistency within the overall configuration of the ECU, AUTOSAR has defined a single format, the `ECU Configuration Description` to be used for all BSW modules within an ECU. Both configuration editors and configuration generators are working toward `ECU Configuration Descriptions`.

[ecuc_sws_1028] This one description (`ECU Configuration Description`) collects the complete configuration of BSW modules in a single ECU. Each module generator may then extract the subset of configuration data it needs from that single format.

2.3.2 Introduction To Configuration Classes

The development of BSW modules involve the following development cycles: compiling, linking and downloading of the executable to ECU memory. Configuration of parameters can be done in any of these process-steps: pre-compile time, link time or even post-build time. According to the process-step that does the configuration of parameters, the configuration classes are categorized as below

- pre-compile time
- link time
- post-build time

The configuration in different process-steps has some consequences for the handling of ECU configuration parameters. If a configuration parameter is defined as pre-compile time, after compilation this configuration parameter can not be changed any more. Or if a configuration parameter is defined at post-build time the configuration parameter has to be stored at a known memory location. Also, the format in which the BSW module is delivered determines in what way parameters are changeable. A source code delivery or an object code delivery of a BSW module has different degrees of freedom regarding the configuration.

The configuration class of a parameter is typically not fixed in the standardized parameter definition since several variants are possible. However once the module is implemented the configuration class for each of the parameters is fixed in that implementation. Choosing the right configuration class from the available variants is depending on the type of application and the design decisions taken by the module implementer. Different configuration classes can be combined within one module. For example, for post-build time configurable BSW implementations only a subset of the parameters might be configurable post-build time. Some parameters might be configured as pre-compile time or link time.

Output file formats used for describing the configuration classes:

- `.xml` (An xml file standardized by AUTOSAR.)
- `.exe` (An executable that can be downloaded to an ECU.)
- `.hex` (A binary file that can be downloaded to an ECU , but it can not execute by its own.)
- `.c` (A C-source file containing either source code or configuration data.)
- `.h` (A header file for either source code or configuration data.)
- `.obj` (A object file for either source code or configuration data.)

2.3.2.1 Configuration Class pre-compile time

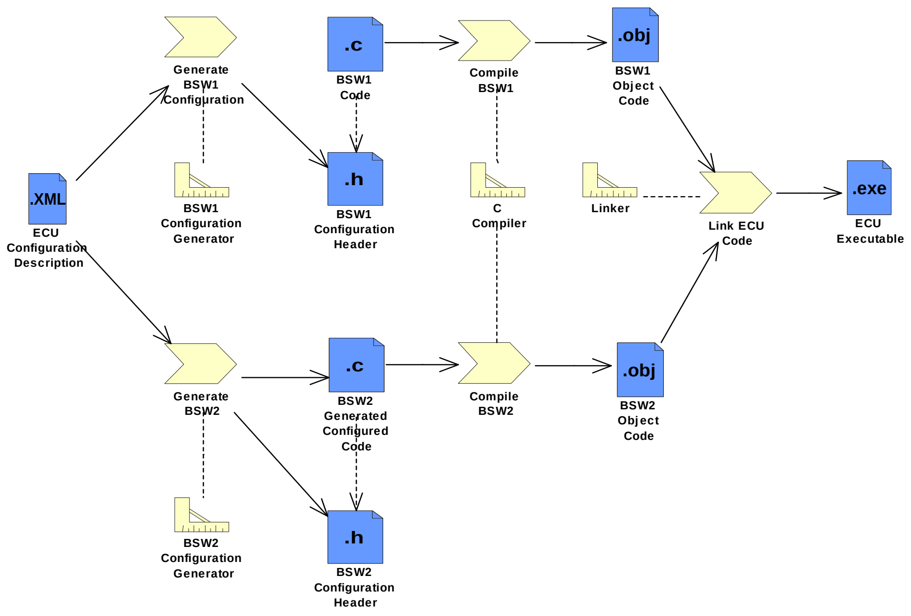
Generate BSW1 Configuration
.c
BSW1 Code
Compile BSW1
.obj
BSW1 Object Code
.XML
ECU Configuration Description
BSW1 Configuration Generator
.h
BSW1 Configuration Header
C Compiler
Linker
Link ECU Code
.exe
ECU Executable
Generate BSW2
.c
BSW2 Generated Configured Code
Compile BSW2
.obj
BSW2 Object Code
BSW2 Configuration Generator
.h
BSW2 Configuration Header

Figure 2.2: Pre-compile time configuration chain

[ecuc_sws_1031] This type of configuration is a standalone configuration done before compiling the source code. That means parameter values for those configurable elements are selected before compiling and will be effective after compilation time. The value of the configurable parameter is decided in earlier stage of software development process and any changes in the parameter value calls for a re-compilation. The contents of pre-compile time parameters can not be changed at the subsequent development steps like link time or post-build time.

Example BSW1 in figure 2.2 shows one possible approach to implement pre-compile time parameters. Configurable parameter values will be kept in a configuration header file and compiled along with the module source file, which is not touched by the `BSW1 Configuration Generator`. Example BSW2 in figure 2.2 shows an alternative approach, in which the `BSW2 Configuration Generator` generates the complete, configuration-specific code. Both approaches are equally valid.

Whenever the decision of parameter value must be taken before the selection of other dependable parameters, pre-compile time configuration is the right choice. For example, the algorithm choice for CRC initial checksum parameter is based on the selection of CRC type (CRC16 or CRC32). When CRC16 is selected, there will be increase in processing time but reduction in memory usage. Whereas when CRC32 is selected, there will be decrease in processing time but increase in memory usage. The correct choice should be made by the implementer before compilation of source code based on the requirement and resource availability.

Sample cases where pre-compile time configuration can be adopted are listed below:

- Configure the number of memory tables and block descriptor table of NVRAM manager.
- Enable the macro for the development error tracing of the software modules.

2.3.2.2 Configuration Class link time

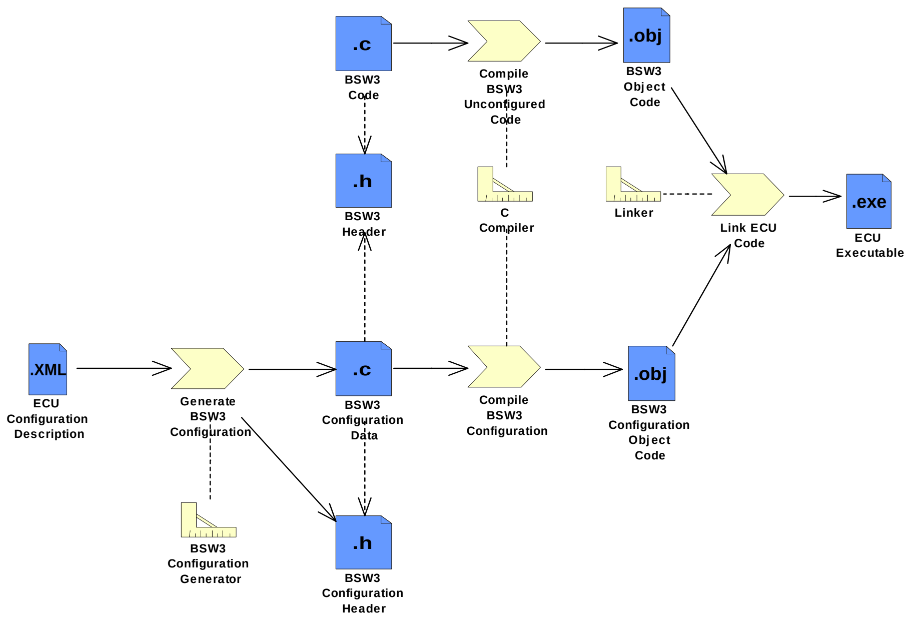
.c
BSW3 Code
Compile BSW3 Unconfigured Code
.obj
BSW3 Object Code
.h
BSW3 Header
C Compiler
Linker
Link ECU Code
.exe
ECU Executable
.XML
ECU Configuration Description
Generate BSW3 Configuration
.c
BSW3 Configuration Data
Compile BSW3 Configuration
.obj
BSW3 Configuration Object Code
BSW3 Configuration Generator
.h
BSW3 Configuration Header

Figure 2.3: Link time configuration chain

[ecuc_sws_1032] This type of configuration is done for the BSW module during link time. That means the object code of the BSW module receives parts of its configuration from another object code file or it is defined by linker options. Link time parameters are typically used when delivering object code to the integrator.

This configuration class provides a modular approach to the configuration process. A separate module will handle the configuration details and those parameter values will be made available to the other modules during the linking process.

In figure 2.3 the configuration parameter data is defined in a common header file (`BSW3 Header`) and included by both module source file (`BSW3 Code`) and module configuration source file (`BSW3 Configuration Data`). The module source file needs this header file to resolve the references and module configuration source file will need it in order to cross check the declaration of data type against the definition. Both module source file and module configuration source file are compiled separately and generates module object file and module configuration object file respectively. During the linking process, the configuration data will be available to module object file by resolving the external references. When the value of configuration parameters is to be changed the module configuration object file needs to be replaced by the one containing the new parameters.

Sample cases where Link time configuration can be adopted are listed below:

- Initial value and invalid value of signal.
- Unique channel identifier configured for the respective instance of the Network Management.

- Logical handle of CAN network.
- Identifier and type of Hardware Reception Handle and Hardware Transmission Handle for CAN interface.
- Definition of ComFilterAlgorithm.
- COM callback function to indicate RTE about the reception of an invalidated signal.

2.3.2.3 Post-build Configuration

There are two kinds of post-build configuration defined. They are:

- Post-build time loadable.
- Post-build time selectable.

For the methodology it is essential to distinguish between `PostBuildLoadable` and `PostBuildSelectable`.

For the ECU Configuration Parameter Definition the configuration classes `PostBuildLoadable` and `PostBuildSelectable` are no longer required because the difference between these two configuration classes is only relevant for the memory mapping of the configuration data during linking.

2.3.2.3.1 Configuration Class post-build time loadable

[ecuc_sws_4006] This type of configuration is possible after building the BSW module or the ECU software. The BSW module gets the parameters of its configuration by downloading a separate file to the ECU memory separately, avoiding a re-compilation and re-build of the BSW module.

In figure 2.4 one approach of post-build time loadable is described.

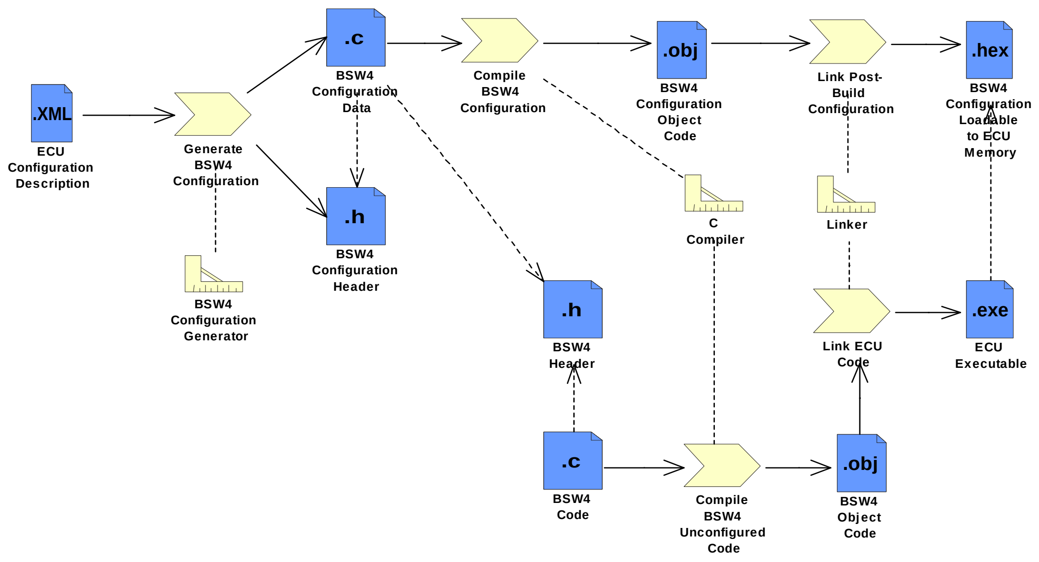
.XML
ECU Configuration Description
Generate BSW4 Configuration
BSW4 Configuration Generator
.c
BSW4 Configuration Data
.h
BSW4 Configuration Header
Compile BSW4 Configuration
.obj
BSW4 Configuration Object Code
C Compiler
Link Post-Build Configuration
Linker
.hex
BSW4 Configuration Loadable to ECU Memory
.h
BSW4 Header
.c
BSW4 Code
Compile BSW4 Unconfigured Code
.obj
BSW4 Object Code
Link ECU Code
.exe
ECU Executable

Figure 2.4: Post-build time loadable configuration chain

In order to make the post-build time loadable re-configuration possible, the re-configurable parameters shall be stored at a known memory location of the ECU memory. An example is shown in figure 2.4. The BSW4 source code (`BSW4 Code`) is compiled and linked independently of its configuration data. The `BSW4 Configuration Generator` generates the configuration data as normal C source code (`BSW4 Configuration Data`) that is compiled and linked independently of the source code. The configuration data, `BSW4 Configuration Loadable to ECU Memory`, is stored at a known memory location and it is possible to exchange the configuration data without replacing the `ECU Executable`.

Another approach of post-build time loadable is shown in figure 2.5.

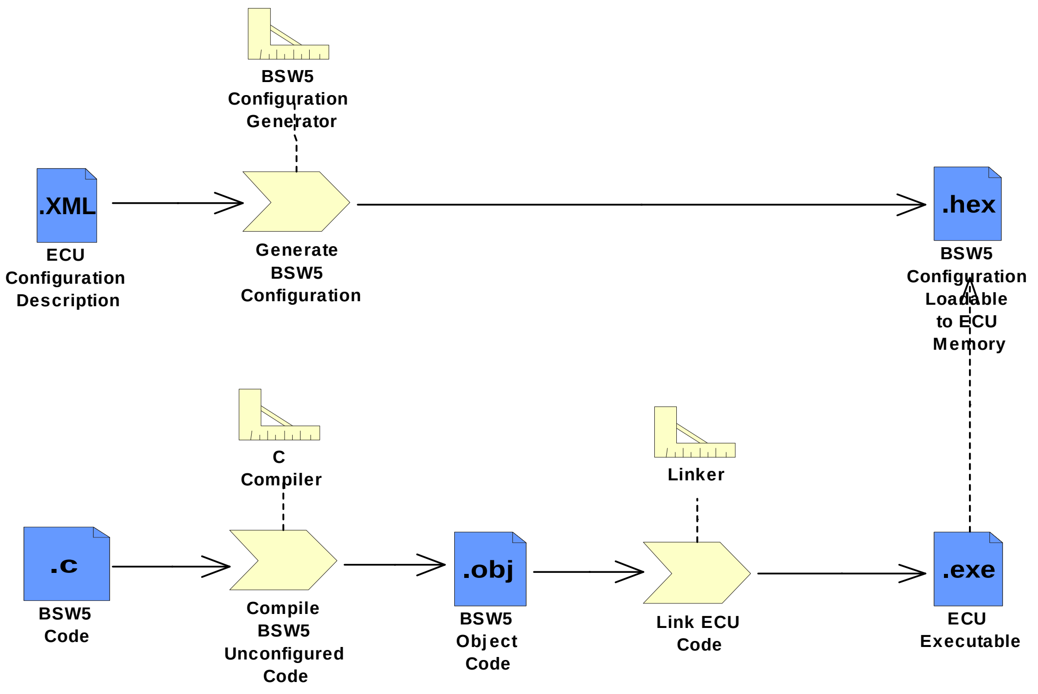
BSW5 Configuration Generator
.XML
ECU Configuration Description
Generate BSW5 Configuration
.hex
BSW5 Configuration Loadable to ECU Memory
C Compiler
Linker
.c
BSW5 Code
Compile BSW5 Unconfigured Code
.obj
BSW5 Object Code
Link ECU Code
.exe
ECU Executable

Figure 2.5: Post-build time loadable configuration chain

The difference compared to the other approach is that the `BSW5 Configuration Generator` does perform the tasks performed by the compiler and linker in the prior approach. I.e the `BSW5 Configuration Loadable to ECU Memory` is generated directly from the generator. The configuration data and the executable is still independently exchangeable.

Sample cases where post-build time loadable configuration can be adopted are listed below.

- Identifiers of the CAN frames
- CAN driver baudrate and propagation delay
- COM transmission mode, transmission mode time offset and time period

2.3.2.3.2 Configuration Class post-build time selectable

[ecuc_sws_4007]Post-build time selectable makes it possible to define multiple configuration sets. Which set that will become active is choosen during boot-time. A description of post-build time selectable is shown in figure 2.6.

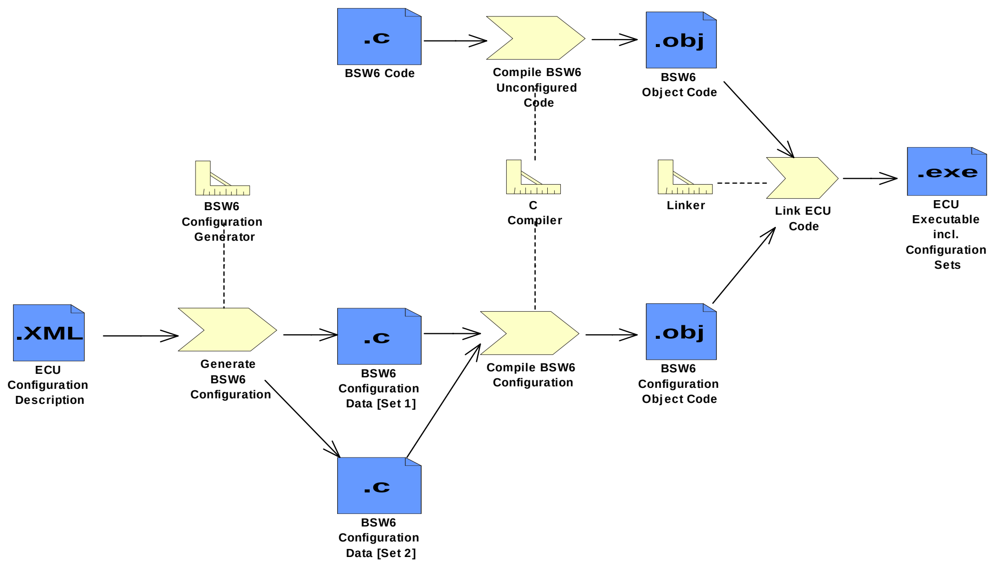
.c
BSW6 Code
Compile BSW6 Unconfigured Code
.obj
BSW6 Object Code
BSW6 Configuration Generator
C Compiler
Linker
Link ECU Code
.exe
ECU Executable incl. Configuration Sets
.XML
ECU Configuration Description
Generate BSW6 Configuration
.c
BSW6 Configuration Data [Set 1]
Compile BSW6 Configuration
.obj
BSW6 Configuration Object Code
.c
BSW6 Configuration Data [Set 2]

Figure 2.6: Post-build time selectable configuration chain

In the example the `BSW6 Configuration Generator` generates two sets of configuration parameters. The configuration data is compiled and linked together with the source code of the BSW module (`BSW6 Code`). The resulting executable, `ECU Executable incl.Configuration Sets`, includes all configuration sets as well as the source code of the BSW module. I.e. it is not possible to exchange the configuration data without re-building the entire executable.

2.3.3 Generate Base ECU Configuration Description

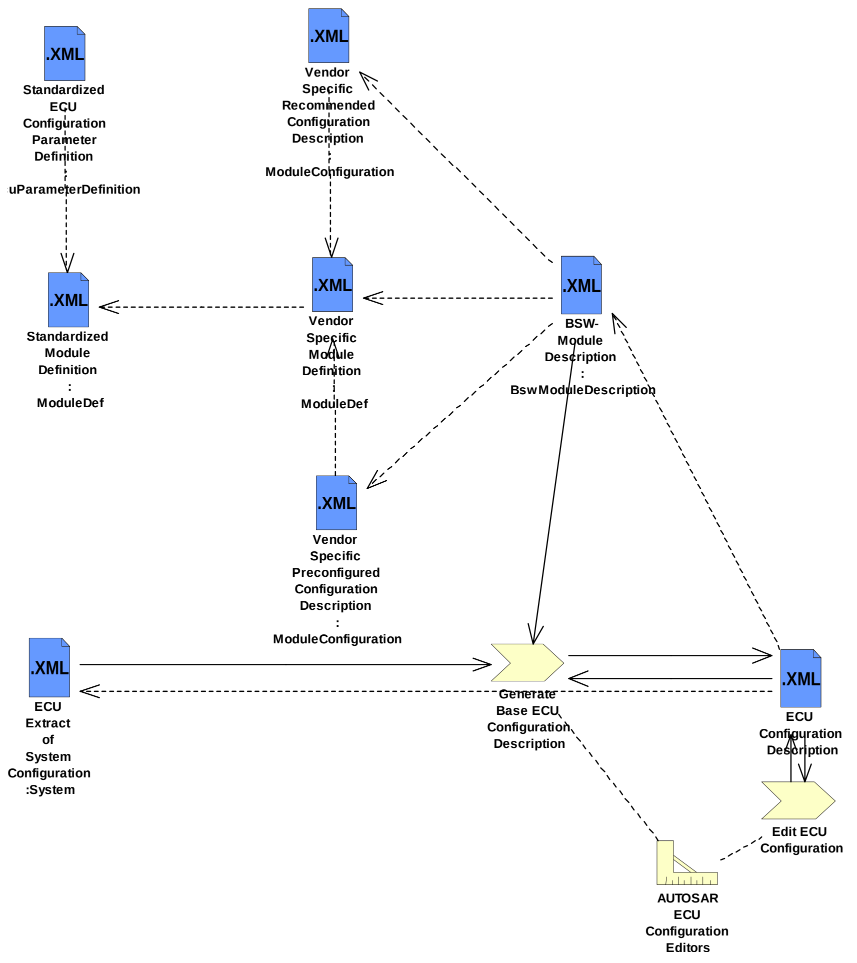
.XML
Standardized ECU Configuration Parameter Definition : uParameterDefinition
.XML
Vendor Specific Recommended Configuration Description : ModuleConfiguration
.XML
Standardized Module Definition : ModuleDef
.XML
Vendor Specific Module Definition : ModuleDef
.XML
BSW-Module Description : BswModuleDescription
.XML
Vendor Specific Preconfigured Configuration Description : ModuleConfiguration
.XML
ECU Extract of System Configuration :System
Generate Base ECU Configuration Description
.XML
ECU Configuration Description
Edit ECU Configuration
AUTOSAR ECU Configuration Editors

Figure 2.7: Generation of the base ECU Configuration Description

The first step in the process of ECU configuration is to generate the base `ECU Configuration Description`. This step involves the generation of the `ECU Composition`. See chapter 2.3.3.1

The `ECU Configuration Description` contains (i.e. references) the configuration of all BSW modules present on the ECU. The configuration of the different BSW modules is done in different sections of the overall description. The section for a specific BSW module in the base `ECU Configuration Description` can be generated using the `Vendor Specific ECU Configuration Parameter Definition` (referenced via the `BSW Module Description` BSWMD for that module) and the `ECU Extract of the System Configuration`, as input, see figure 2.7. This generation is a semi-automatic process.

[ecuc_sws_1029] For each BSW module that shall be present in the ECU, the implementation must be chosen. This is done by referencing the BSWMD delivered with the BSW module. The BSWMD defines all configuration parameters, and their structuring in containers, relevant for this specific implementation of the module. This is done in the `Vendor Specific Module Definition`. The rules that must be followed when building the base ECU Configuration Description are available in chapter 5.2.

2.3.3.1 Generating the ECU Composition

In the ECU Extract of the System Configuration only the application Software Components are considered. The RTE and all BSW modules are not taken into account in the System Configuration. In ECU Configuration all aspects of the ECU software need to be considered, therefore means to support the addition of the BSW and RTE need to be provided.

In the ECU Configuration Description an additional hierarchical level is introduced which defines the Application SW-Component instances (*EcuTopLevelCompositionPrototype*) and the AUTOSAR BSW Service instances (see figure 2.8 *EcuSwComposition*). AUTOSAR BSW Services are modules like the NvRam Manager, the Watchdog Manager, the ECU State Manager, etc.

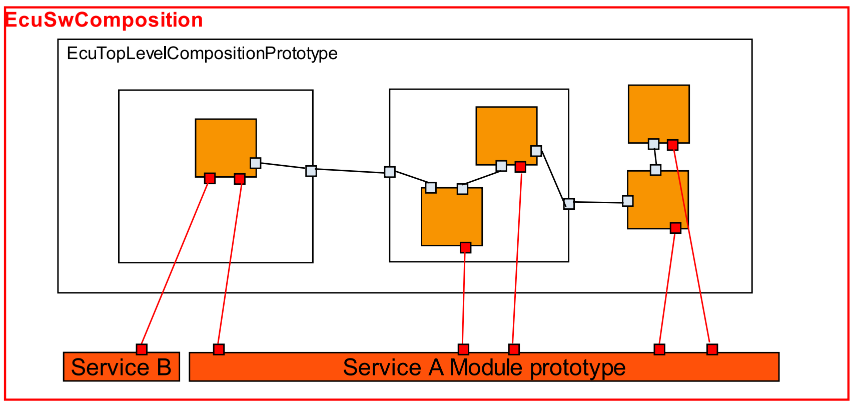

Figure 2.8: Structure of the EcuComposition introduced in the ECU Configuration

[ecuc_sws_2085] When generating the Base ECU Configuration the EcuSwComposition is introduced, which defines one additional level of hierarchy for SW-Components.

[ecuc_sws_2087] The EcuSwComposition contains the SW-Component descriptions of the application SW-Component prototypes, the AUTOSAR BSW Service modules prototypes and the connections between the Application SW-Component ports and the BSW Service modules' ports.

[ecuc_sws_2086] The TopLevelComposition is *instantiated* in the EcuSwComposition with the ComponentPrototype name EcuTopLevelCompositionPrototype.

The TopLevelComposition is defined in the System Description ([10]) using the means of the SW-Component template [7]. The *instantiation* is done by defining a ComponentPrototype using the mechanisms defined in the SW-Component template.

When generating the AUTOSAR BSW Services SW-Components the actual *needs*[1] from the Application SW-Components are collected and an appropriate number of ports is created at each BSW Service SW-Component.

The connections of the Application SW-Component ports to the BSW Service ports are special, because it is allowed to connect ports which are on different levels of hierarchy (this is not allowed in the plain SW-Component descriptions).

2.3.4 Edit ECU Configuration

The second step in the process of ECU configuration is to edit the configuration parameters for all BSW modules.

[ecuc_sws_1030] Once the section for a specific BSW module has been generated in the base `ECU Configuration Description`, it can be edited with AUTOSAR `ECU Configuration Editors`. Those editors may operate with user interaction, semi automatically or automatically, depending on BSW module and implementation. A straightforward approach editing the `ECU Configuration Description` is described in figure 2.1.

2.3.4.1 Details in Edit ECU Configuration

Editing the ECU Configuration is a process that has some aspects which put specific requirements on tools and workprocedures. One aspect is the iterative process when editing ECU configuration parameters and another aspect is support for configuration management.

2.3.4.1.1 Iterations within ECU Configuration

What appears clear is that there are likely to be both optimizations and trade-offs to be made between parameters, both within and between BSW modules. The configuration deals with, for example, detailed scheduling information or the configuration data for the needed BSW modules. Hence this is a non-trivial design step and requires complex design algorithms and/or engineering knowledge. ECU Configuration is thus likely to be an iterative process. This iteration will initially be between editors and then, when a plausible ECU Configuration is achieved, code generation may highlight additional changes that require further iteration. It is hoped that the majority of generator-editor iterations will be limited by ensuring that the editor tools are capable of spotting/predicting potential generator errors and ensuring that the engineer corrects them prior to entering generation.

[1]The *needs* of the Application SW-Components are defined in the SW-Component description in the `ServiceNeeds` section.

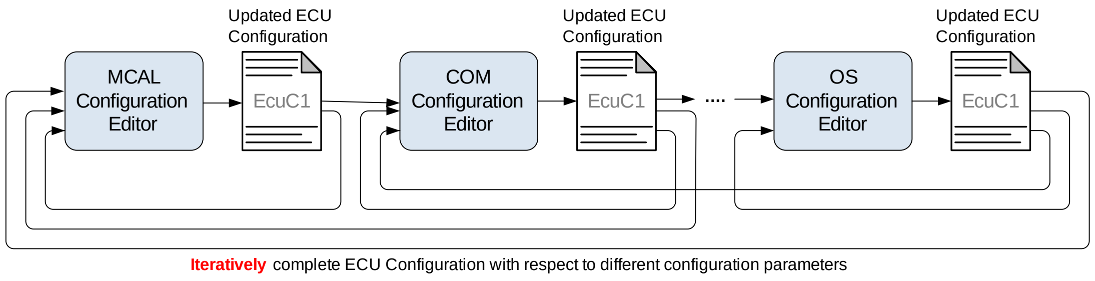
Updated ECU Configuration
MCAL Configuration Editor
EcuC1
Updated ECU Configuration
COM Configuration Editor
EcuC1
....
Updated ECU Configuration
OS Configuration Editor
EcuC1
Iteratively complete ECU Configuration with respect to different configuration parameters

Figure 2.9: Sequential Application of tools

Figure 2.9 shows how a set of custom tools might be used in a chain with iteration in order to achieve a successful ECU Configuration. Tools are sequentially called within a tool chain to create an `ECU Configuration Description`. Iteration cycles must be implemented by repeated activation of different configuration tools for specific aspects of the BSW. Dependencies between tools, as well as the configuration work flow, might need to be expressed explicitly. Configuration tools are required only to support a single standardized interface, the `ECU Configuration Description` Template.

Tools supporting the methodology and the iterations needed for ECU configuration can be designed based on different strategies. Chapter A.1.1 gives som information about this topic.

Iterations, as described in figure 2.9, will be divided between several organisations due to the fact that parameters within a BSW module are either configured pre-compile time, link time or post-build time. Typically pre-compile time parameters are configured by a Tier2 supplier and post-build time parameters are configured by the OEM. Link time parameters can either be configured by a Tier1 or Tier2 supplier, depending on the delivery format of the BSW module.

A description of editing ECU configuration parameters in an iterative manner with several organisations involved is described in figure 2.10.

Configuring of pre-compile time and post-build time parameters occur at different organisations. The methodology supports parallel and iterative configuration activities. A pre-compile time parameter can affect i.e. a post-build time parameter. The methodology supports description of dependencies between parameters.

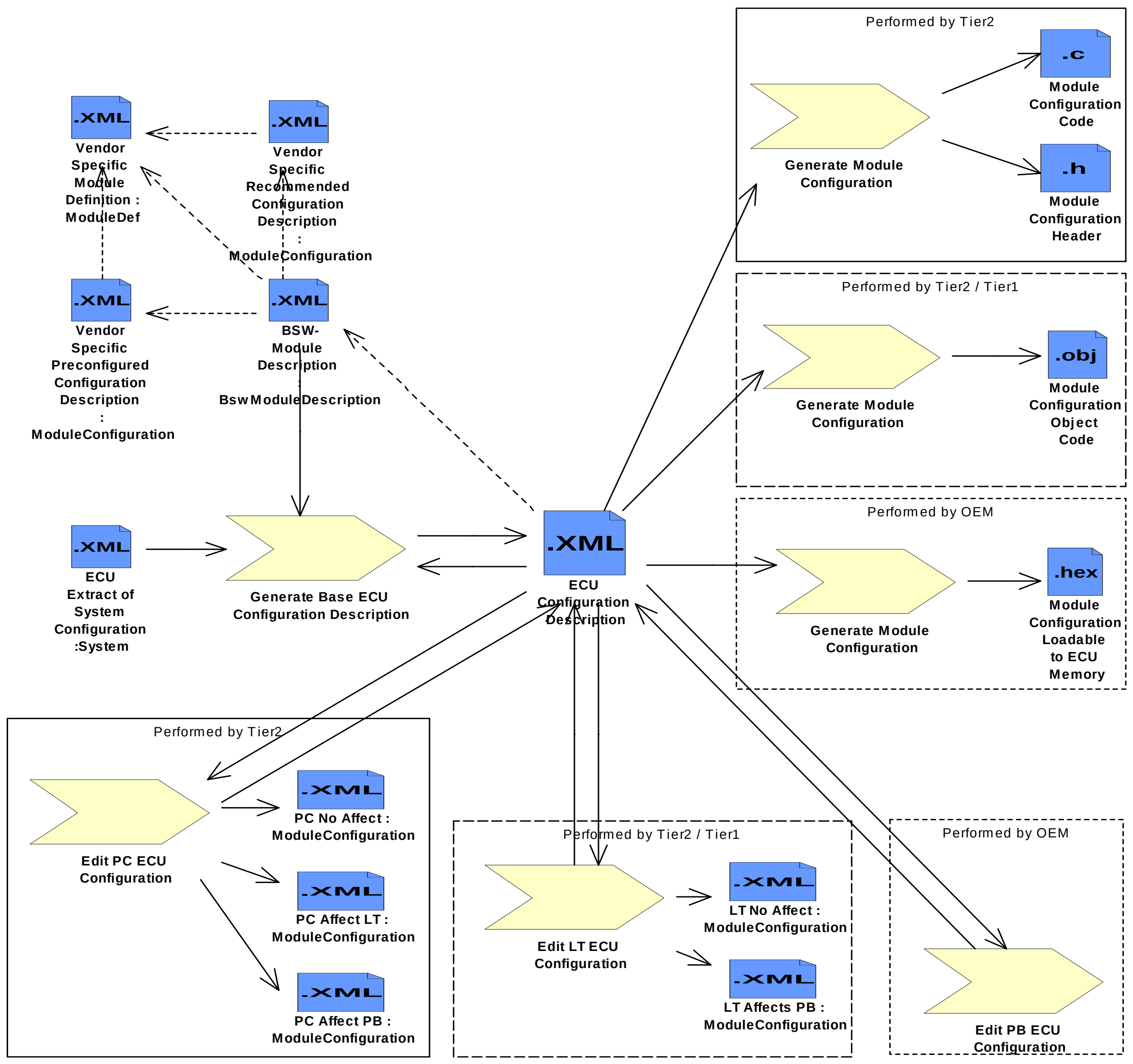
Performed by Tier2
.c
Module Configuration Code
Generate Module Configuration
.h
Module Configuration Header
.XML
Vendor Specific Module Definition : ModuleDef
.XML
Vendor Specific Recommended Configuration Description : ModuleConfiguration
Performed by Tier2 / Tier1
Generate Module Configuration
.obj
Module Configuration Object Code
.XML
Vendor Specific Preconfigured Configuration Description : ModuleConfiguration
.XML
BSW-Module Description
BswModuleDescription
Performed by OEM
.XML
ECU Extract of System Configuration :System
Generate Base ECU Configuration Description
.XML
ECU Configuration Description
Generate Module Configuration
.hex
Module Configuration Loadable to ECU Memory
Performed by Tier2
Edit PC ECU Configuration
.XML
PC No Affect : ModuleConfiguration
.XML
PC Affect LT : ModuleConfiguration
.XML
PC Affect PB : ModuleConfiguration
Performed by Tier2 / Tier1
Edit LT ECU Configuration
.XML
LT No Affect : ModuleConfiguration
.XML
LT Affects PB : ModuleConfiguration
Performed by OEM
Edit PB ECU Configuration

Figure 2.10: Detailed description of ECU configuration

The BSWMD (delivered by a Tier2 supplier) contains a description of the entire BSW module, including dependencies between parameters. Each parameter has an attribute that can be assigned one of these values:

- NO-AFFECT (The parameter has no affect on any other parameter)
- PC-AFFECTS-LT (A pre-compile time parameter affecting one or several link time parameter(s))
- PC-AFFECTS-PB (A pre-compile time parameter affecting one or several post-build time parameter(s))
- PC-AFFECTS-LT-AND-PB (A pre-compile time parameter affecting one or several link time and post-build time parameter(s))
- LT-AFFECTS-PB (A link time parameter affecting one or several post-build time parameter(s))

In addition it is also possible to list the affected parameters in the BSWMD. The description of dependencies makes it possible to inform about changes that will affect other organisations taking part in the ECU configuration.

[ecuc_sws_4000] In the figure 2.10 there are three activities defined: "‘Edit PC ECU Configuration"’, "‘Edit LT ECU Configuration"’ and "‘Edit PB ECU Configuration"’. These activities are performed by different organisations. In order to transfer information from one step of the configuration to another, it is possible to generate a file containing parameters that affect another step of the configuration. This is done by i.e. generating the xml-file: "‘PC Affect PB"’. The xml-file is a `ModuleConfiguration` and contains the pre-compile time parameters that has an affect on post-build time parameter(s) for a specific BSW module or cluster of BSW modules. Since the editor is the same in all steps of the configuration of a BSW module or a cluster the "‘PC Affect PB"’ and all other files can be generated by any person or organisation using the editor.

After the first generation of the base `ECU Configuration Description` the different organisations can start their part of the parameter configuration by editing the `ECU Configuration Description`. The different organisations are most likely working with a local copy of the `ECU Configuration Description`. Eventually there is a need to combine the local copies into one `ECU Configuration Description`. This is done with a simple merge tool, that is a part of the `ECU Configuration Editor`. The merging is easy since there should be no redundant sections. To ensure editing only pre-compile time, link time or post-build time parameters, the `ECU Configuration Editor` shall be able to define a subset of parameters that are allowed to be edited. The subset can be any combination of pre-compile time, link time and post-build time parameters.

Requirements for `ECU Configuration Editors` supporting the methodology can be found in chapter 5.3.

2.3.4.1.2 Configuration Management and Post-build Time Loadable

Post-build time loadable permits the change of configuration parameter values after building the rest of the ECU-SW (BSW modules and SW-Cs) by downloading a new configuration loadable to the ECU memory at a specific address. This implies that there are at least two SW articles with unique part numbers for an ECU if using the post-build loadable strategy. (There can be more than two since every BSW module can theoretically be configured post-build time loadable). Since there are several SW articles with unique part numbers there is a must to keep track of each SW article from a Configuration Management perspective. In order to do this for each post-build time loadable, `ModuleConfigurations` describing different aspects (E.g the post-build aspect) of a BSW module needs to be put under Configuration Management as a separate file.

In figure 2.11 the relationships between `ModuleConfigurations` describing different aspects and the SW articles with their unique part numbers are shown. Note that the relationships are Configuration Management(CM) relations. This means if a parameter in the `ModuleConfiguration` describing the post-build aspect is changed it is only needed to re-build the configuration data, the rest of the SW articles in the ECU remain untouched. If a parameter in a `ModuleConfiguration` describing the pre-compile aspect is changed the BSW module needs to be re-build. The configuration data must also be re-build if the `ModuleConfiguration` contains e.g. a pre-compile time parameter with the attribute "'PC affects PB"'. See chapter 2.3.4.1.1. Each `ModuleConfiguration` describing a certain aspect, e.g the post-build aspect, must be put under Configuration Management as a Configuration Item(CI).

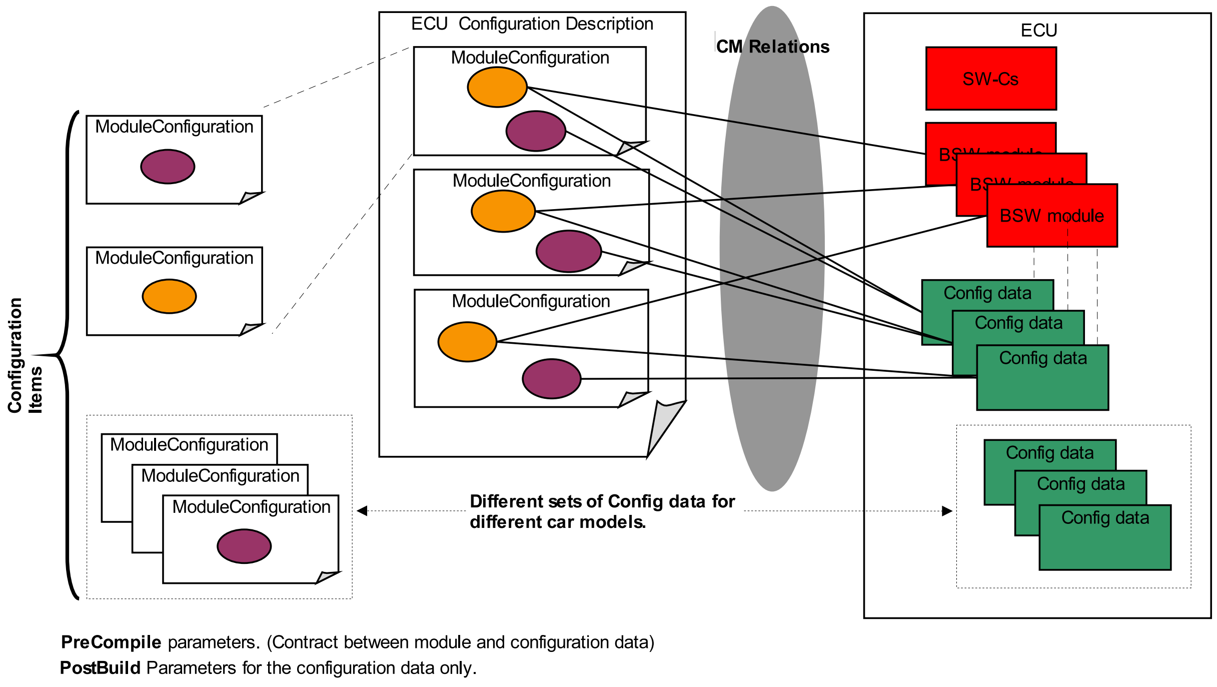

Figure 2.11: Configuration Items(CIs) and post-build time loadable configuration.

Another use case where ModuleConfigurations describing different aspects must be Configuration Items, is for different car models which can have different set of configuration data, but the rest of the SW in the ECU is the same for all car models. See figure 2.11.

Requirements for the ECU Configuration Editors supporting post-build time loadable strategy can be found in chapter 5.3.

2.3.5 Generate Configured Module Code

The third and last step of the AUTOSAR ECU Configuration methodology has already been referenced in the preceding sections and so comes as no surprise. Generation of configured module code for the different BSW modules. Generation is the process of applying the tailored `ECU Configuration Description` to the software modules. This can be performed in different ways, and is dependent on the configuration classes chosen for the different modules (see chapter 2.3.2), and on implementers choices.

For each BSW module, a generator reads the relevant parameters from the `ECU Configuration Description` and creates code that implements the specified configuration, as shown on the right hand side of figures A.1 and A.2. In this generation step, the abstract parameters of the `ECU Configuration Description` are translated to hardware and implementation-specific data structures that fit to the implementation of the corresponding software module. This specification does not specify the generator tools in detail. It is assumed however that generators perform error, consistency and completeness checks on the part of the configuration they require for generation.

There are some alternative approaches when it comes to generation of configuration data. See chapter A.1.2 for more details.

3 Configuration Metamodel

3.1 Introduction

AUTOSAR exchange formats are specified using a metamodel based approach (see also *Specification of Interoperability of Authoring Tools* [11]). The metamodel for the configuration of ECU artifacts uses an universal description language so that it is possible to specify different kinds of configuration aspects. This is important as it is possible to describe AUTOSAR-standardized and vendor-specific ECU Configuration Parameters with the same set of language elements. This eases the development of tools and introduces the possibility to standardize vendor-specific ECU Configuration Parameters at a later point in time.

In general the configuration language uses containers and actual parameters. Containers are used to group corresponding parameters. Parameters hold the relevant values that configure the specific parts of an ECU. Due to the flexibility that has to be achieved by the configuration language the configuration description is divided into two parts:

- ECU Configuration Parameter Definition
- ECU Configuration Description

A detailed description of these two parts and their relationships are presented in the following sections.

3.2 ECU Configuration Template Structure

In this section the relationships between the different AUTOSAR templates involved in the ECU Configuration are introduced. A template is defining the structure and possible content of an actual description. The concept is open to be implemented in several possible ways, in AUTOSAR XML files have been chosen to be used for the exchange formats. If XML files are used there is no conceptual limit in the number of files making up the description. All the contributing files are virtually merged to build the actual description[1].

The goal of the ECU Configuration Description template is to specify an exchange format for the ECU Configuration Values of one ECU. The actual output of ECU Configuration editors is stored in the ECU Configuration Description, which might be one or several XML files. But the ECU Configuration editors need to know how the content of an ECU Configuration Description should be structured (which parameters are available in which container) and what kind of restrictions are to be respected (e.g. the ECU Configuration Parameter is an integer value in the range between 0 and 255). This is specified in the ECU Configuration Parameter Definition which is also an XML file. The relationship between the two file types is shown in figure 3.1.

[1]The rules are defined in the Specification of Interoperability of Authoring Tools document [11].

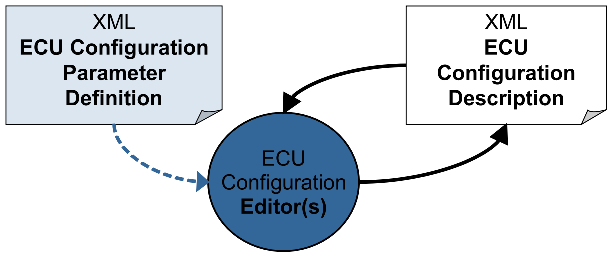
XML
ECU Configuration Parameter Definition
XML
ECU Configuration Description
ECU Configuration Editor(s)

Figure 3.1: Parameter Definition and ECU Configuration Description files

For the ECU Configuration editors there are basically two possible approaches how to implement these definitions. Either the ECU Configuration Parameter Definition is read and interpreted directly from the XML file or the defined structures are hard-coded into the tool[2].

For the development of the ECU Configuration Parameter Definition and the ECU Configuration Description a model-based approach has been chosen which already has been used during the development of other AUTOSAR template formats.

The main approach is to use a subset of UML to graphically model the desired entities and their relationships. Then, in a generation step, the actual XML formats are automatically generated out of the model.

[ecuc_sws_2000] The modeling of the ECU Configuration Description and ECU Configuration Parameter Definition metamodels is done according to the Template UML Profile and Modeling Guide [12].

[ecuc_sws_2001] The transformation of the ECU Configuration Description and ECU Configuration Parameter Definition metamodels to schema definitions is done according to the Model Persistence Rules for XML [8].

Because of these transformation rules there is a given discrepancy between the UML model and the generated XML-Schema names. This also affects this document. The major descriptions will be based on the UML model notations (figures and tables), although the corresponding XML notation might be given for reference purposes.

In this section the application of the modeling approach for the ECU Configuration is described.

AUTOSAR uses the UML metamodel (M2-level) to describe the classes and objects that may be used in an AUTOSAR-compliant system. These metamodel elements may be used in an application model (M1-level) to describe the content of a real vehicle. ECU Configuration is a part of the AUTOSAR standard so the elements of ECU Configuration Description must be described in the UML metamodel at M2-level. The (M2) metamodel has therefore been populated with UML descriptions from which ECU Configuration Parameter models may be built.

[2]The advantage of using the interpreter is that changes on the ECU Configuration Parameter Definition are directly available in the tool. But the hard-coded approach allows for more custom user support in the tool

With M2 definitions in place, it is possible to create AUTOSAR-conforming models of real application ECU Configuration Parameters (an ECU Configuration Parameter Definition Model) at M1-level. Certain aspects of real application configurations are already defined: BSW Modules have standard interfaces and configuration requirements. These 'real' configuration parameters have therefore already been modeled at M1-level for each defined BSW Module. These are described in detail in the SWS documents.

XML has been chosen as the technology that will be used by AUTOSAR-compliant tools in order to define and share information during an AUTOSAR-compliant system development. It must therefore be possible to transform the UML Configuration Parameter Definition Model (M1-level) into an XML Configuration Parameter Definition so that it may be used by ECU Configuration tools. This is the way that the tool gets a definition of exactly which ECU Configuration Parameters are available and how they may be configured. The Model Persistence Rules for XML [8] describes how the UML metamodel (M2-level) may be transformed into a schema that describes the format of XML to contain model elements.

This same formalization is also true for the ECU Configuration Parameter Definition Metamodel elements on M2-level: the Model Persistence Rules for XML dictates how ECU Configuration Parameter Definition elements will generate a schema to hold ECU Configuration Parameter Model (M1-level) elements in an XML ECU Configuration Parameter Definition, that can then be interpreted by ECU Configuration tools.

ECU Configuration editors allow a system designer to set ECU Configuration Parameter Values for their particular application. The actual values are then stored in an ECU Configuration Description that conforms to the template described in the UML.

An ECU Configuration Description is an XML file that conforms to an AUTOSAR schema called an ECU Configuration Description Template. The template in turn is an AUTOSAR standard defined by placing ECU Configuration Template elements into the UML Meta-Model (M2-level) such that the schema (the ECU Configuration Description Template) can be generated (using the Formalization Guide rules).

There are three different parts involved in the development of the ECU Configuration: UML models, Schema and XML content files. The overview is shown in figure 3.2.

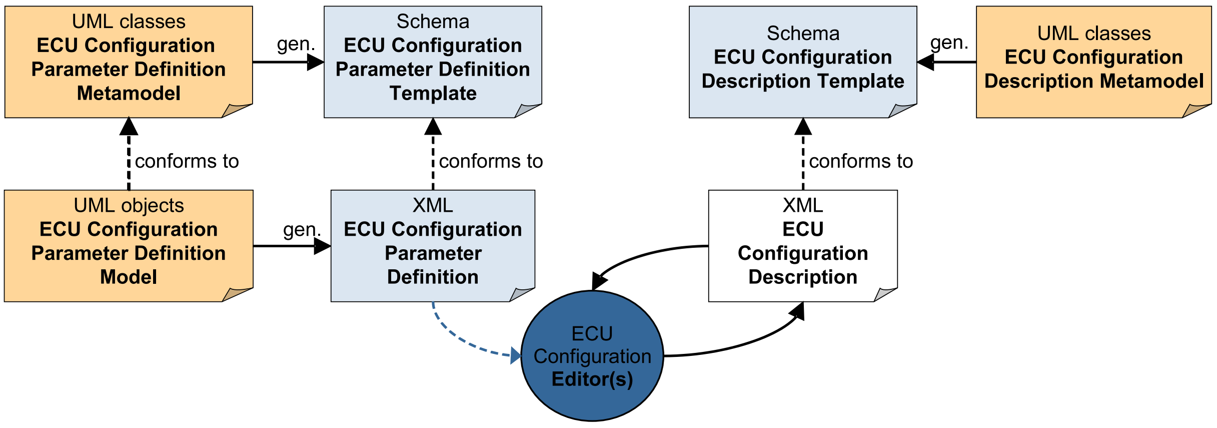
UML classes ECU Configuration Parameter Definition Metamodel
gen.
Schema ECU Configuration Parameter Definition Template
Schema ECU Configuration Description Template
gen.
UML classes ECU Configuration Description Metamodel
conforms to
conforms to
conforms to
UML objects ECU Configuration Parameter Definition Model
gen.
XML ECU Configuration Parameter Definition
XML ECU Configuration Description
ECU Configuration Editor(s)

Figure 3.2: Relationship between UML models and XML files

[ecuc_sws_2045] The ECU Configuration Parameter Definition Model is used to specify the ECU Configuration Parameter Definition. This is done using object diagrams (this is the M1 level of metamodeling) with special semantics defined in section 3.3. What kind of UML elements are allowed in the ECU Configuration Parameter Definition Model is defined in the ECU Configuration Parameter Definition Metamodel which is conforming to the Template UML Profile and Modeling Guide [12]. The definition is done using UML class diagrams (which is done on M2 level of metamodeling).

Out of the ECU Configuration Parameter Definition Metamodel a schema [3] is generated and the generated ECU Configuration Parameter Definition XML file has to conform to this schema. Vendor-specific ECU Configuration Parameter Definitions need to conform to this schema as well.

The ECU Configuration Description XML file needs to conform to the ECU Configuration Description Template schema which itself is generated out of the ECU Configuration Description Metamodel specified in UML class diagrams as well.

In the next section the ECU Configuration Parameter Definition Metamodel and its application toward the ECU Configuration Parameter Definition Model is described.

3.3 ECU Configuration Parameter Definition Metamodel

The two major building blocks for the specification of ECU Configuration Parameter Definitions are containers and parameters/references. With the ability to establish relationships between containers and parameters and the means to specify references, the definition of parameters has enough power for the needs of the ECU Configuration.

[3]Whether a DTD or an XML-Schema is used is not relevant for this explanation and is left to the formalization strategy defined in [8].

3.3.1 ECU Configuration Parameter Definition top-level structure

The definition of each Software Module's[4] configuration has at the top level the same structure which is shown in figure 3.3.

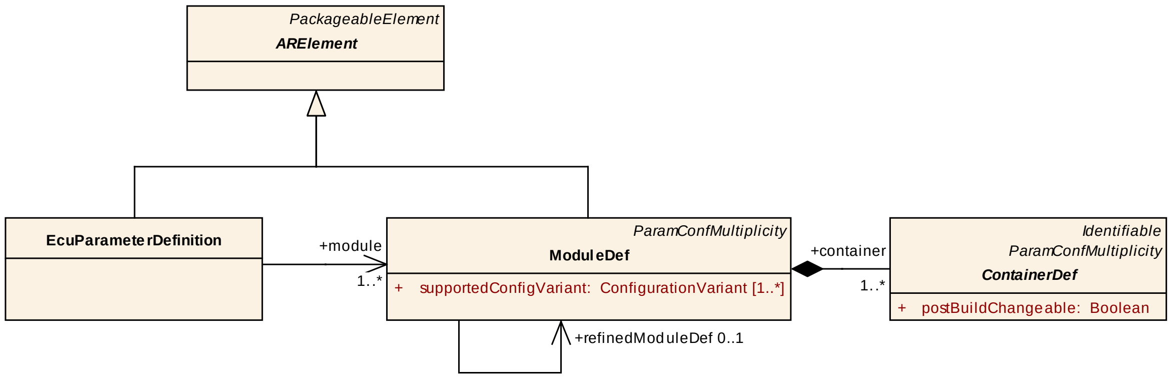

Figure 3.3: ECU Configuration Parameter Definition top-level structure

[ecuc_sws_2002] The generic structure of all AUTOSAR templates is described in detail in the Template UML Profile and Modeling Guide [12].

[ecuc_sws_2003] First ECU Configuration specific class is the `EcuParameterDefinition` which inherits from `ARElement`. Through this inheritance the `EcuParameterDefinition` can be part of the AUTOSAR `topLevelPackage` and thus part of an AUTOSAR description.

[ecuc_sws_2065] The Configuration Parameter Definition of one module is called `ModuleDef` and is an `ARElement`.

`ARElement` itself inherits form `Identifiable` which has two consequences: First, each `Identifiable` has to have a machine readable `shortName`. Second, the `Identifiable` introduces the concept of a namespace for the contained `Identifiable` objects, so those objects need to have unique `shortNames` in the scope of that namespace. For additional information about the consequences of being an `Identifiable` and the additional attributes please refer to [12].

In the following figures and tables the names from the UML model are shown. In the generated XML-Schema the names may differ based on the Model Persistence Rules for XML [8]. For instance, the attribute `shortName` will become `SHORT-NAME` in the XML-Schema.

[4]A Software Module might be Basic Software, Application Software Component or the RTE, see AUTOSAR Glossary [2]

Class	**Identifiable (abstract)**			
Package	M2::AUTOSARTemplates::GenericStructure::Infrastructure::Identifiable			
Class Desc.	Instances of this class can be referred to by their identifier (while adhering to namespace borders).			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
shortName	Identifier	1	aggregation	Use <shortName> to generate a short name for the context element, which enables it to be ** .
longName	MlData4	0..1	aggregation	Use <longName> to create a comprehensive name for the context element.
desc	MlData2	0..1	aggregation	<desc> represents a general but brief description of the object in question.
category	String	0..1	aggregation	This element assigns a category to the parent element. The category can be used by a semantic checker in post-processes to ensure that the parent object is defined correctly i.e. has the right number of elements for example.
adminData	AdminData	0..1	aggregation	<adminData> can be used to set administrative information for an element. This administration information is to be treated as metadata such as revision id or state of the file. There are basically four kinds of metadata * The language and/or used laguages. * Revision information covering e.g. revision number, state, release date, changes. Note that this information can be given in general as well as related to a particular company. * Document metadata specific for a company * Formatting controls that can affect layouts for example. * Revision information for the element.

uuid	String	1	aggregation	The purpose of this attribute is to provide a globally unique identifier for an instance of a metaclass. 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".

Table 3.1: Identifiable

[ecuc_sws_2004] The usage-case of the `EcuParameterDefinition` class is to collect all references to individual module configuration definitions of the AUTOSAR ECU Configuration. Therefore the `EcuParameterDefinition` defines a reference relationship to the definition of several Software Modules in the `module` attribute.

Class	**EcuParameterDefinition**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	This represents the anchor point of an ECU Configuration Parameter Definition within the AUTOSAR templates structure.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
module	ModuleDef	1..*	reference	References to the module definitions of individual software modules.

Table 3.2: EcuParameterDefinition

3.3.1.1 Usage of the Admin Data

`AdminData` can be used to set administrative information for an element (e.g. version information). Such administrative information can be set for the whole ECU Configuration Parameter Definition XML file and for each module definition.

[ecuc_sws_6004] An `AdminData` field is required at the beginning of every ECU Configuration Parameter Definition XML file (regardless whether it is the StMD or the VSMD file) to allow the setting of `AdminData` for the whole XML File. The usage of this field is optional.

Example 3.1 shows how `AdminData` can be used for the whole ECU Configuration Parameter Definition XML file.

Example 3.1

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/2.1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://autosar.org/2.1.0 autosar.xsd">
  <ADMIN-DATA>
    <DOC-REVISIONS>
      <DOC-REVISION>
        <REVISION-LABEL>
          <L-10>revision2.1.1</L-10>
        </REVISION-LABEL>
        <ISSUED-BY>
          <L-10>AUTOSAR GbR</L-10>
        </ISSUED-BY>
      </DOC-REVISION>
    </DOC-REVISIONS>
  </ADMIN-DATA>
  <TOP-LEVEL-PACKAGES>
    <!-- AR-Package: AUTOSAR -->
    <AR-PACKAGE UUID="ECUC:AUTOSAR">
      <SHORT-NAME>AUTOSAR</SHORT-NAME>
      <ELEMENTS>
        <ECU-PARAMETER-DEFINITION>
          <SHORT-NAME>AUTOSARParameterDefinition</SHORT-NAME>
```

[ecuc_sws_6005] For each module definition there needs to be provided which revision the StMD is. For the VSMD the AUTOSAR release version and the vendor's own version information must be provided. The usage of `AdminData` on `ModuleDef` is mandatory.

Example 3.2 shows that there are possibilities to specify several elements for the `AdminData`. The initial one would be provided by AUTOSAR, the additional one is the vendor's information which is based on the AUTOSAR one.

Example 3.2

```
 ...
        <MODULE-DEF UUID="ECUC:c03229fe-4dca-445e-a47c-1ae11e6c1832">
          <SHORT-NAME>Adc</SHORT-NAME>
          <DESC>
            <L-2 L="EN">Configuration of the Adc
(Analog Digital Conversion) module.</L-2>
          </DESC>
          <ADMIN-DATA>
            <DOC-REVISIONS>
              <DOC-REVISION>
                <REVISION-LABEL>
                  <L-10>revision2.1.1</L-10>
                </REVISION-LABEL>
                <ISSUED-BY>
                  <L-10>AUTOSAR GbR</L-10>
                </ISSUED-BY>
                <DATE>
                  <L-10>09.05.2007</L-10>
                </DATE>
              </DOC-REVISION>
              <DOC-REVISION>
                <REVISION-LABEL>
                  <L-10>version15.3</L-10>
                </REVISION-LABEL>
                <!--predecessor -->
                <REVISION-LABEL-P-1>revision2.1.1</REVISION-LABEL-P-1>
                <ISSUED-BY>
                  <L-10>VendorX</L-10>
                </ISSUED-BY>
                <DATE>
                  <L-10>19.05.2007</L-10>
                </DATE>
              </DOC-REVISION>
            </DOC-REVISIONS>
          </ADMIN-DATA>
          <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
          <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
          <CONTAINERS>
...
```

3.3.2 ECU Configuration Module Definition

[ecuc_sws_2005] The class `ModuleDef` is defining the ECU Configuration Parameters of one Software Module[5]. It is inheriting form `ARElement`, so each individual `ModuleDef` needs to have an unique name within its enclosing `ARPackage`.

[ecuc_sws_2059] The `ModuleDef` is using the `ParamConfMultiplicity` to specify how many instances of that specific module are allowed in the ECU Configuration Description (see section 3.3.4.1).

Class	**ModuleDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Used as the top-level element for configuration definition for Software Modules, including BSW and RTE as well as ECU Infrastructure.			
Base Class(es)	ARElement , ParamConfMultiplicity			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
container	Container Def	1..*	aggregation	Aggregates the top-level container definitions of this specific module definition.
refined ModuleDef	ModuleDef	0..1	reference	Optional reference from the Vendor Specific Module Definition to the Standardized Module Definition it refines.
supported Config Variant	Configuration Variant	1..*	aggregation	Specifies which ConfigurationVariants are supported by this software module.

Table 3.3: ModuleDef

[ecuc_sws_2094] The `ModuleDef` aggregates container definitions (`ContainerDef`) with the role name `container` which may hold other container definitions, parameter definitions and reference definitions.

[ecuc_sws_2095] The reference `refinedModuleDef` to another `ModuleDef` is used to specify that this `ModuleDef` is the *Vendor Specific Module Definition* for the referenced `ModuleDef` (which is then the *Standardized Module Definition*).

[ecuc_sws_2096] The `ModuleDef` specifies which configuration variants are supported by this software modules configuration using the element `supportedConfig-Variant`. For a detailed description how the configuration variants are related to the configuration classes please refer to section 3.3.4.2.2. For each configuration variant that is supported one entry shall be provided.

In figure 3.4 an actual example of the top-level structure is provided and in the example 3.3 the corresponding ECU Configuration Parameter Definition XML file extract is shown. In the example XML also the overall XML structure of AUTOSAR descriptions is shown.

[5]A Software Module is not restricted to the BSW Modules but also includes the RTE, Application Software Components and generic ECU Configuration.

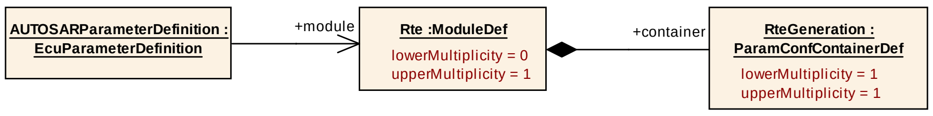

Figure 3.4: ECU Configuration Definition example

Example 3.3

```
<AUTOSAR>
  <TOP-LEVEL-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>AUTOSAR</SHORT-NAME>
      <ELEMENTS>
        <ECU-PARAMETER-DEFINITION>
          <SHORT-NAME>ParamDef</SHORT-NAME>
          <MODULE-REFS>
            <MODULE-REF DEST="MODULE-DEF">/AUTOSAR/Rte</MODULE-REF>
            <!-- Further references to module definitions -->
          </MODULE-REFS>
        </ECU-PARAMETER-DEFINITION>
        <MODULE-DEF>
          <SHORT-NAME>Rte</SHORT-NAME>
          <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
          <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
          <SUPPORTED-CONFIG-VARIANTS>
  <SUPPORTED-CONFIG-VARIANT>VARIANT-PRE-COMPILE</SUPPORTED-CONFIG-VARIANT>
          </SUPPORTED-CONFIG-VARIANTS>
          <CONTAINERS>
            <!-- ... -->
          </CONTAINERS>
        </MODULE-DEF>
      </ELEMENTS>
    </AR-PACKAGE>
  </TOP-LEVEL-PACKAGES>
</AUTOSAR>
```

In the next sections the structure of containers, individual parameters and references is introduced.

3.3.3 Container Definition

[ecuc_sws_2006] The container definition is used to group other parameter container definitions, parameter definitions and reference definitions.

There are two specializations of a container definition. The abstract class `ContainerDef` is used to gather the common features (see figure 3.5).

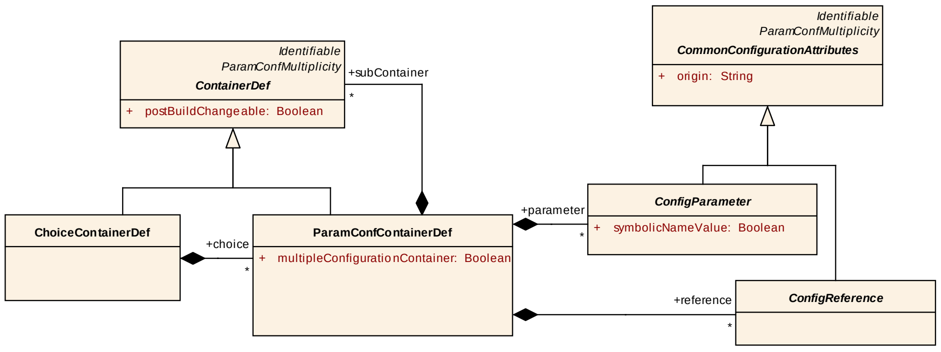

Figure 3.5: Class diagram for parameter container definition

Class	**ContainerDef (abstract)**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Base class used to gather common attributes of configuration container definitions.			
Base Class(es)	ParamConfMultiplicity , Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
postBuild Change-able	Boolean	1	aggregation	Specifies if the number of instances of this container may be changed post-build time. This parameter may only be set to true if all of the following conditions hold: - the container's upperMultiplicity > lowerMultiplicity - all parameters within the container and subContainers are post-build time changeable. If any of the aggregated parameters is either pre-compile time or link time this attribute is ignored and may be omitted.

Table 3.4: ContainerDef

[ecuc_sws_2043] Each `ContainerDef` is an `Identifiable`.

[ecuc_sws_2044] Each `ContainerDef` also has the features of `ParamConfMultiplicity` which enables to specify for each `ContainerDef` how often it is allowed to occur in the ECU Configuration Description later on (see section 3.3.4.1).

[ecuc_sws_2064] The attribute `postBuildChangeable` specifies if the number of containers can be changed `PostBuild time` in the ECU Configuration Description.

[ecuc_sws_2007] A `ParamConfContainerDef` is the main container class definition and can contain other containers, configuration parameters and references.

Class	**ParamConfContainerDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Used to define configuration containers that can hierarchically contain other containers and/or parameter definitions.			
Base Class(es)	ContainerDef			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
multiple Config-uration Container	Boolean	1	aggregation	Specifies whether this container is used to define multiple configuration sets. Only one container in the whole ModuleDef shall have this enabled.
parameter	Config Parameter	*	aggregation	The parameters defined within the ParamConfContainerDef.
reference	Config Reference	*	aggregation	The references defined within the ParamConfContainerDef.
subCon-tainer	Container Def	*	aggregation	The containers defined within the ParamConfContainerDef.

Table 3.5: ParamConfContainerDef

One example of a `ContainerDef` and its embedding in the ECU Configuration Parameter Definition is shown in figure 3.6. One `ModuleDef Rte` is specified being part of the `EcuParameterDefinition`. Two containers of type `ParameterConfParam-Def` are specified as part of the module definition.

When specifying the containment relationship between the `ModuleDef` and containers the role name `container` is used. When specifying the containment relationship between two containers an aggregation with the role name `subContainer` at the contained container is used.

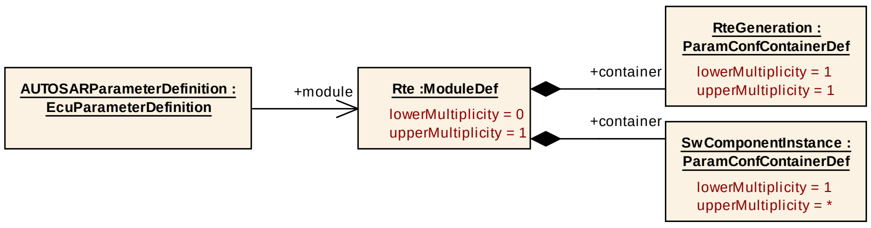

Figure 3.6: Example of an object diagram for container definition

In the XML outtake in example 3.4 only the relevant part from figure 3.6 is shown, not including the `EcuParameterDefinition`.

Example 3.4

```
      ...
        <MODULE-DEF>
          <SHORT-NAME>Rte</SHORT-NAME>
          <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
          <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
          <CONTAINERS>
            <PARAM-CONF-CONTAINER-DEF>
              <SHORT-NAME>RteGeneration</SHORT-NAME>
              <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
              <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
<MULTIPLE-CONFIGURATION-CONTAINER>false</MULTIPLE-CONFIGURATION-CONTAINER>
            </PARAM-CONF-CONTAINER-DEF>
            <PARAM-CONF-CONTAINER-DEF>
              <SHORT-NAME>SwComponentInstance</SHORT-NAME>
              <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
              <UPPER-MULTIPLICITY>*</UPPER-MULTIPLICITY>
<MULTIPLE-CONFIGURATION-CONTAINER>false</MULTIPLE-CONFIGURATION-CONTAINER>
            </PARAM-CONF-CONTAINER-DEF>
          </CONTAINERS>
        </MODULE-DEF>
      ...
```

3.3.3.1 Choice Container Definition

[ecuc_sws_2011] The `ChoiceContainerDef` can be used to specify that certain containers might occur exclusively in the ECU Configuration Description. In the ECU Configuration Parameter Definition the potential containers are specified as part of the `ChoiceContainerDef` and the constraint is that in the actual ECU Configuration Description only some of those specified containers will actually be present.

Class	**ChoiceContainerDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Used to define configuration containers that provide a choice between several ParamConfContainerDef. But in the actual ECU Configuration Description only one instance from the choice list will be present (depending on the multiplicites given).			
Base Class(es)	ContainerDef			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
choice	ParamConf Container Def	*	aggregation	The choices available in a ChoiceContainerDef.

Table 3.6: ChoiceContainerDef

[ecuc_sws_2067] The multiplicity of the *to be chosen* containers shall always be `0..1`, indicating that each time a choice is performed you can only choose one of these *to be chosen* containers at a time.

[ecuc_sws_2012] The `upperMultiplicity` of the `ChoiceContainerDef` does specify how often the choice can be performed. If the `upperMultiplicity` of the `ChoiceContainerDef > 1`, the *SUM* of the chosen containers has to be considered.

[ecuc_sws_2068] Each time a choice can be performed, the user is free to choose from the available *to be chosen* containers. The Configuration Editor needs to collect all the chosen containers of the ECU Configuration Description and make sure the sum is not greater than the `upperMultiplicity` of the `ChoiceContainerDef`.

This limits the usage of `ChoiceContainerDef`: Each *to be chosen* container can only be defined as a choice for one `ChoiceContainerDef` in the same parent container (otherwise it is not possible to distinguish from which `ChoiceContainerDef` a *to be chosen* container is from. And a container can not be a *to be chosen* container as well as a plain *subContainer* in the same parent container (otherwise it is not possible to distinguish between the plain container and the `ChoiceContainerDef` choice).

An example of the usage of a `ChoiceContainerDef` is shown in figure 3.7 and the corresponding XML definition is shown in example 3.5.

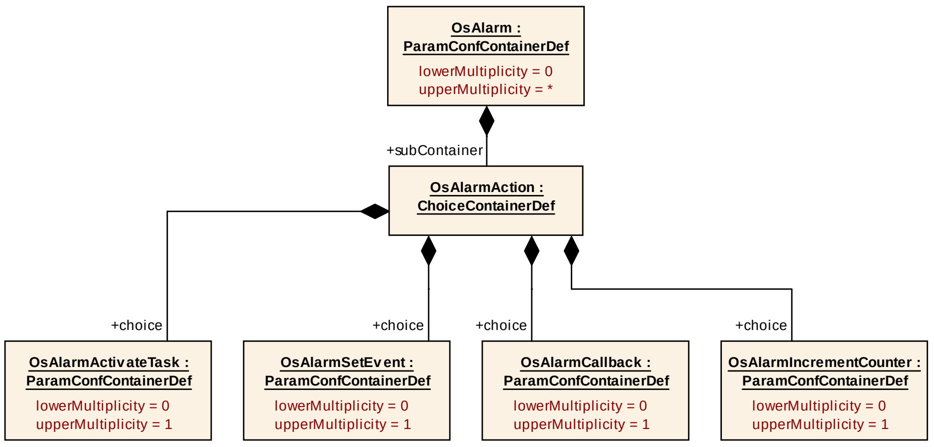

Figure 3.7: Example of an object diagram for choice container definition

The `ChoiceContainerDef OSAlarmAction` is defined to be able to hold one of the four given containers later in the ECU Configuration Description. Since the `upperMultiplicity` of `OsAlarmAction = 1` there can only be one choice taken.

Example 3.5

```
...
      <PARAM-CONF-CONTAINER-DEF>
        <SHORT-NAME>OsAlarm</SHORT-NAME>
        <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
        <UPPER-MULTIPLICITY>*</UPPER-MULTIPLICITY>
        <SUB-CONTAINERS>
          <CHOICE-CONTAINER-DEF>
            <SHORT-NAME>OsAlarmAction</SHORT-NAME>
            <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
            <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
            <CHOICES>
              <PARAM-CONF-CONTAINER-DEF>
                <SHORT-NAME>OsAlarmActivateTask</SHORT-NAME>
                <!-- ... -->
              </PARAM-CONF-CONTAINER-DEF>
              <PARAM-CONF-CONTAINER-DEF>
                <SHORT-NAME>OsAlarmSetEvent</SHORT-NAME>
                <!-- ... -->
              </PARAM-CONF-CONTAINER-DEF>
              <PARAM-CONF-CONTAINER-DEF>
                <SHORT-NAME>OsAlarmCallback</SHORT-NAME>
                <!-- ... -->
              </PARAM-CONF-CONTAINER-DEF>
              <PARAM-CONF-CONTAINER-DEF>
                <SHORT-NAME>OsAlarmIncrementCounter</SHORT-NAME>
                <!-- ... -->
              </PARAM-CONF-CONTAINER-DEF>
            </CHOICES>
          </CHOICE-CONTAINER-DEF>
        </SUB-CONTAINERS>
      </PARAM-CONF-CONTAINER-DEF>
...
```

The containers from the example, which the choice is from, will of course have to be specified in more detail in an actual definition file.

3.3.3.2 Multiple Configuration Set Definition

To allow the description of several Ecu Configuration Sets a `ModuleDef` may contain *one* `ParamConfContainerDef` which is specified to be the `multipleConfigurationContainer`.

[ecuc_sws_2091] A `ParamConfContainerDef` does specify whether the defined container is the `multipleConfigurationContainer`. Each `ModuleDef` shall contain exactly one such container if it supports multiple configuration sets.

If a `ParamConfContainerDef` is specified to be the `multipleConfigurationSetContainer` there can be several `Container` descriptions of this container in the

Ecu Configuration Description for PostBuild configurations[6]. The `shortName` of the `multipleConfigurationContainer` does define the name of the configuration set (see also section 3.4.7 for details on the Ecu Configuration Description).

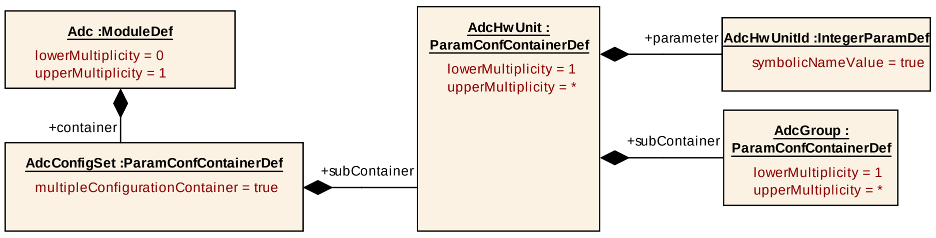

Figure 3.8: Example of an object diagram for multiple configuration container definition

Example 3.6

```
    ...
      <MODULE-DEF>
        <SHORT-NAME>Adc</SHORT-NAME>
        <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
        <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
        <CONTAINERS>
          <PARAM-CONF-CONTAINER-DEF>
            <SHORT-NAME>AdcConfigSet</SHORT-NAME>
            <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
            <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
<MULTIPLE-CONFIGURATION-CONTAINER>true</MULTIPLE-CONFIGURATION-CONTAINER>
            <SUB-CONTAINERS>
              <PARAM-CONF-CONTAINER-DEF>
                <SHORT-NAME>AdcHwUnit</SHORT-NAME>
<MULTIPLE-CONFIGURATION-CONTAINER>false</MULTIPLE-CONFIGURATION-CONTAINER>
                <PARAMETERS>
                  <INTEGER-PARAM-DEF>
                    <SHORT-NAME>AdcHwUnitIt</SHORT-NAME>
                  </INTEGER-PARAM-DEF>
                </PARAMETERS>
                <SUB-CONTAINERS>
                  <PARAM-CONF-CONTAINER-DEF>
                    <SHORT-NAME>AdcGroup</SHORT-NAME>
<MULTIPLE-CONFIGURATION-CONTAINER>false</MULTIPLE-CONFIGURATION-CONTAINER>
                    <!-- ... -->
                  </PARAM-CONF-CONTAINER-DEF>
                </SUB-CONTAINERS>
              </PARAM-CONF-CONTAINER-DEF>
            </SUB-CONTAINERS>
          </PARAM-CONF-CONTAINER-DEF>
        </CONTAINERS>
      </MODULE-DEF>
    ...
```

[6] In case of PreComiple and LinkTime configuration variants the features of the `multipleConfigurationContainer` are not used.

For the ECU Configuration Description of this example please refer to section 3.4.7.

3.3.4 Common Configuration Elements

Configuration Parameters and references have some common attributes which are described in this section.

3.3.4.1 Parameter Configuration Multiplicity

[ecuc_sws_2008] To be able to specify how often a specific configuration element (container, parameter or reference) may occur in the ECU Configuration Description the class `ParamConfMultiplicity` is introduced. With the two attributes `lowerMultiplicity` and `upperMultiplicity` the minimum and maximum occurrence of the configuration element is specified.

[ecuc_sws_2009] When there is no multiplicity specified the default is exactly '1' meaning the element is mandatory in the ECU Configuration Description and has to occur exactly once. To express an optional element the `lowerMultiplicity` has to be set to '0'. To express unlimited number of occurrences of this element the `upperMultiplicity` is to be set to '*'.

Configuration Parameter and Reference definitions with an `upperMultiplicity` > `1` have to be considered with care, since it is not possible to reference to individual parameters. So such multiple occurrences of a parameter in the description will just be mere collections, it is neither guaranteed that the order will be preserved nor that that individual elements do have a special semantics.

[ecuc_sws_2010] In the specification object diagrams the multiplicity attributes may be omitted if both values are equal to the default value of '1'. Otherwise both attributes are shown.

Class	**ParamConfMultiplicity (abstract)**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Common class used to express multiplicities in the definition of configuration parameters, references and containers. If not stated otherwise the default multiplicity is exactly one mandatory occurrence of the specified element.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
lower Multiplicity	String	1	aggregation	The lower multiplicity of the specified element. 0: optional 1: at least one occurence n: at least n occurrences
upper Multiplicity	String	1	aggregation	The upper multiplicity of the specified element. 1: at most one occurrence m: at most m occurrences *: arbitrary number of occurrences

Table 3.7: ParamConfMultiplicity

For examples please refer to figure 3.6 and example 3.4

3.3.4.2 Common Configuration Attributes

Several attributes are available on both, parameters and references. These common attributes are shown in figure 3.9.

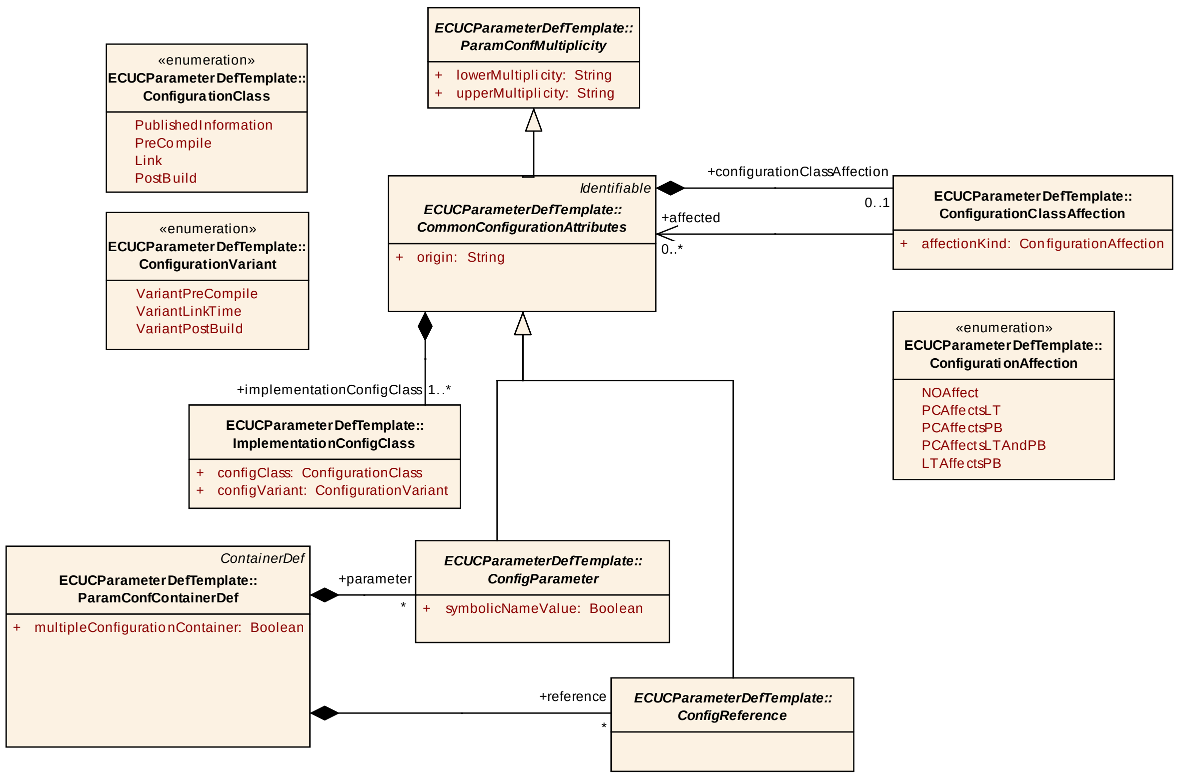
«enumeration»
ECUCParameterDefTemplate::ConfigurationClass
PublishedInformation
PreCompile
Link
PostBuild
«enumeration»
ECUCParameterDefTemplate::ConfigurationVariant
VariantPreCompile
VariantLinkTime
VariantPostBuild
ECUCParameterDefTemplate::ParamConfMultiplicity
+ lowerMultiplicity: String
+ upperMultiplicity: String
Identifiable
ECUCParameterDefTemplate::CommonConfigurationAttributes
+ origin: String
+configurationClassAffection
0..1
+affected
0..*
ECUCParameterDefTemplate::ConfigurationClassAffection
+ affectionKind: ConfigurationAffection
«enumeration»
ECUCParameterDefTemplate::ConfigurationAffection
NOAffect
PCAffectsLT
PCAffectsPB
PCAffectsLTAndPB
LTAffectsPB
+implementationConfigClass
1..*
ECUCParameterDefTemplate::ImplementationConfigClass
+ configClass: ConfigurationClass
+ configVariant: ConfigurationVariant
ContainerDef
ECUCParameterDefTemplate::ParamConfContainerDef
+ multipleConfigurationContainer: Boolean
+parameter
*
ECUCParameterDefTemplate::ConfigParameter
+ symbolicNameValue: Boolean
+reference
*
ECUCParameterDefTemplate::ConfigReference

Figure 3.9: Common Attributes for parameters and references

Class	**CommonConfigurationAttributes (abstract)**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Attributes used by Configuration Parameters as well as References.			
Base Class(es)	ParamConfMultiplicity , Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
configuration ClassAf- fection	Configuration ClassAf- fection	0..1	aggregation	Specifes whether changes on this parameter have some affection on other parameters.
implementation Config Class	Implementation Config Class	1..*	aggregation	Specifies which in which ConfigurationClass this parameter or reference is available for which ConfigurationVariant.
origin	String	1	aggregation	String specifying if this configuration parameter is an AUTOSAR standardized configuration parameter or if the parameter is hardware- or vendor-specific.

Table 3.8: CommonConfigurationAttributes

3.3.4.2.1 Parameter Origin

[ecuc_sws_2015] Each parameter type has to provide information on its `origin`, which contains a string describing if the parameter is defined in the AUTOSAR standard ('AUTOSAR_ECUC') or if the parameter is defined as a vendor specific parameter (e.g. 'VendorXYZ_v1.3').

Example 3.7

```
...
    <INTEGER-PARAM-DEF>
      <SHORT-NAME>ClockRate</SHORT-NAME>
      <ORIGIN>AUTOSAR_ECUC</ORIGIN>
    </INTEGER-PARAM-DEF>
    <BOOLEAN-PARAM-DEF>
      <SHORT-NAME>VendorExtensionEnabled</SHORT-NAME>
      <ORIGIN>VendorXYZ_v1.3</ORIGIN>
    </BOOLEAN-PARAM-DEF>
...
```

In example 3.7 two parameters are defined, one which belongs to the AUTOSAR standard and one which is introduced by the module vendor in a specific version of his own ECU Configuration tools.

3.3.4.2.2 Implementation Configuration Classes

[ecuc_sws_2016] The attribute `implementationConfigClass` provides information what kind of configuration class this parameter shall be implemented for each of the supported configuration variants. The different configuration classes defined within AUTOSAR are[7]:

- **[ecuc_sws_2070]** `PublishedInformation`
- **[ecuc_sws_2017]** `PreCompile`
- **[ecuc_sws_2018]** `Link`
- **[ecuc_sws_2019]** `PostBuild`[8]

The element `PublishedInformation` is used to specify the fact that certain information is fixed even before the pre-compile stage.

[ecuc_sws_2071] If `PublishedInformation` is selected as configuration class it has to be the for all configuration variants.

[ecuc_sws_2022] The configuration parameter definition of the BSW has the possibility to define up to three configuration variants how actual configuration parameters can be implemented. So the implementor of the module does not have complete freedom how the configuration classes are chosen for each individual configuration parameter but needs to select one of the specified variants.

[ecuc_sws_2097] The supported configuration variants are[9]:

- **[ecuc_sws_2098]** `VariantPreCompile`
- **[ecuc_sws_2099]** `VariantLinkTime`
- **[ecuc_sws_2100]** `VariantPostBuild`

The mapping of the `ConfigurationVariant` to the `ConfigurationClass` is done using the `ImplementationConfigClass`:

[7]In the XML-Schema the values are represented as `PUBLISHED-INFORMATION`, `PRE-COMPILE`, `LINK`, `POST-BUILD`.

[8]The configuration classes `PostBuildLoadable` and `PostBuildSelectable` are no longer required for the ECU Configuration Parameter Definition because the difference between these two configuration classes is only relevant for the memory mapping of the configuration data during linking.

[9]In the XML-Schema the values are represented as `VARIANT-PRE-COMPILE`, `VARIANT-LINK-TIME`, `VARIANT-POST-BUILD`.

Class	**ImplementationConfigClass**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Specifies which ConfigurationClass this parameter has in the individual ConfigurationVariants.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
config Class	Configuration Class	1	aggregation	Specifies the ConfigurationClass for the given ConfigurationVariant.
config Variant	Configuration Variant	1	aggregation	Specifies the ConfigurationVariant the ConfigurationClass is specified for.

Table 3.9: ImplementationConfigClass

[ecuc_sws_2101] For each `ConfigurationVariant` the `ModuleDef` supports there shall be one `ImplementationConfigClass` element.

The supported configuration variants of the module are described in section 3.3.2.

[ecuc_sws_2102] Every `ImplementationConfigClass` specifies which `ConfigurationClass` this parameter or reference shall be implemented for this `ConfigurationVariant`.

The example 3.8 shows how the `ImplementationConfigClass` is provided in XML for three configuration variants of some module. The integer configuration parameter `SignalSize` shall be implemented as a `PRE-COMPILE` parameter for the configuration variants `VARIANT-PRE-COMPILE` and `VARIANT-LINK-TIME`. It shall be `POST-BUILD` for the configuration variant `VARIANT-POST-BUILD`.

Example 3.8

```
...
  <INTEGER-PARAM-DEF>
    <SHORT-NAME>SignalSize</SHORT-NAME>
    <IMPLEMENTATION-CONFIG-CLASSES>
      <IMPLEMENTATION-CONFIG-CLASS>
        <CONFIG-CLASS>PRE-COMPILE</CONFIG-CLASS>
        <CONFIG-VARIANT>VARIANT-PRE-COMPILE</CONFIG-VARIANT>
      </IMPLEMENTATION-CONFIG-CLASS>
      <IMPLEMENTATION-CONFIG-CLASS>
        <CONFIG-CLASS>PRE-COMPILE</CONFIG-CLASS>
        <CONFIG-VARIANT>VARIANT-LINK-TIME</CONFIG-VARIANT>
      </IMPLEMENTATION-CONFIG-CLASS>
      <IMPLEMENTATION-CONFIG-CLASS>
        <CONFIG-CLASS>POST-BUILD</CONFIG-CLASS>
        <CONFIG-VARIANT>VARIANT-POST-BUILD</CONFIG-VARIANT>
      </IMPLEMENTATION-CONFIG-CLASS>
    </IMPLEMENTATION-CONFIG-CLASSES>
  </INTEGER-PARAM-DEF>
 ...
```

The configuration tools are now able to derive the configuration class of each configuration parameter and reference from the ECU Configuration Parameter Definition XML file [13].

3.3.4.2.3 Configuration Class Affection

The `ConfigurationClassAffection` is used to describe whether a specific configuration parameter is affecting any other configuration parameters in the ECU Configuration Description and in which configuration phase this affection occurs. The actual affection will be described in the Vendor Specific Module Definition based on the actual implementation.

The possible values for the `affectionKind` of a `ConfigurationClassAffection` are[10]:

- **[ecuc_sws_2076]** `NOAffect`
- **[ecuc_sws_2077]** `PCAffectsLT`
- **[ecuc_sws_2078]** `PCAffectsPB`
- **[ecuc_sws_2079]** `PCAffectsLTAndPB`
- **[ecuc_sws_2080]** `LTAffectsPB`

[ecuc_sws_2081] The reference `affected` from the `ConfigurationClassAffection` to any subclass of `CommonConfigurationAttributes` is used to define which actual parameters and references are affected.

For a detailed description of the affection mechanism refer to section 2.3.4.1.

Class	**ConfigurationClassAffection**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Specifies in the "VendorSpecificModuleDefinition" whether changes on this parameter do affect other parameters in a later configuration step.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
affected	Common Config-uration Attributes	*	reference	Optional reference to parameters or references which are affected by the ConfigurationClassAffection.
affection Kind	Configuration Affection	1	aggregation	Specifies which affect do changes in this parameter have on other parameters.

Table 3.10: ConfigurationClassAffection

[10]In the XML-Schema the values are represented as `NO-AFFECT`, `PC-AFFECTS-LT`, `PC-AFFECTS-PB`, `PC-AFFECTS-LT-AND-PB`, `LT-AFFECTS-PB`.

3.3.5 Parameter Definition

[ecuc_sws_2013] Parameters are defined within a `ParamConfContainerDef` using an aggregation with the role name `parameter` at the parameter side.

[ecuc_sws_2014] The possible parameter types are specified using one of the specialized classes derived from `ParameterType`. The `ParameterType` does inherit from `Identifiable` and `ParamConfMultiplicity`.

The available parameter types are shown in figure 3.10.

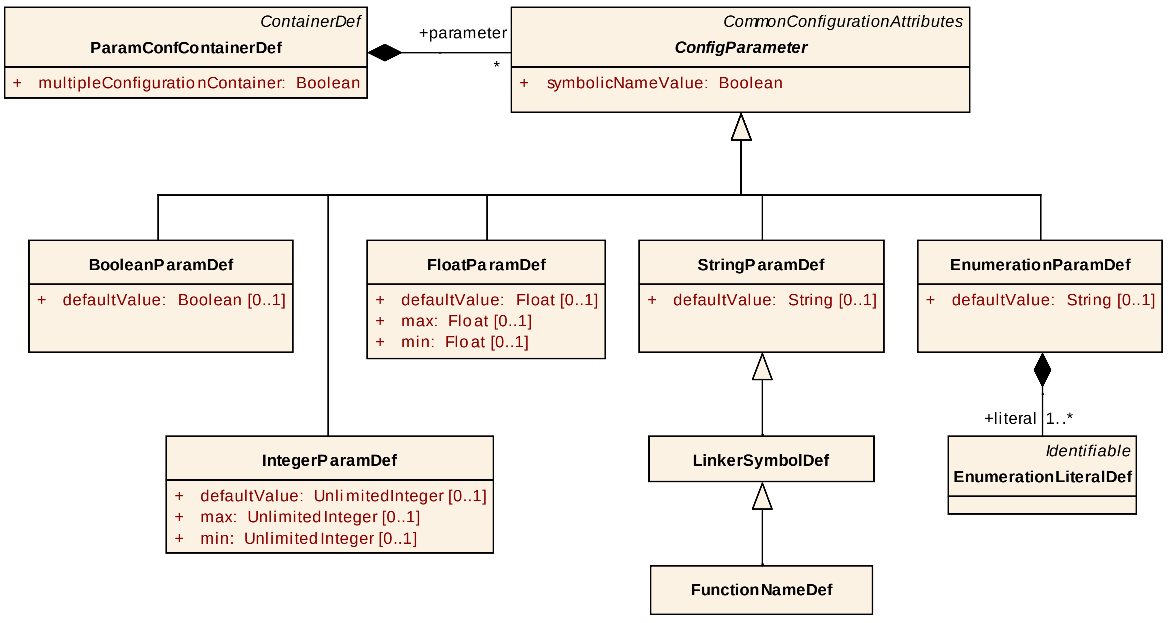
ContainerDef
ParamConfContainerDef
+ multipleConfigurationContainer: Boolean
+parameter
*
CommonConfigurationAttributes
ConfigParameter
+ symbolicNameValue: Boolean
BooleanParamDef
+ defaultValue: Boolean [0..1]
FloatParamDef
+ defaultValue: Float [0..1]
+ max: Float [0..1]
+ min: Float [0..1]
StringParamDef
+ defaultValue: String [0..1]
EnumerationParamDef
+ defaultValue: String [0..1]
+literal
1..*
Identifiable
EnumerationLiteralDef
IntegerParamDef
+ defaultValue: UnlimitedInteger [0..1]
+ max: UnlimitedInteger [0..1]
+ min: UnlimitedInteger [0..1]
LinkerSymbolDef
FunctionNameDef

Figure 3.10: Class diagram for parameter definition

Class	**ConfigParameter (abstract)**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Abstract class used to define the similarities of all ECU Configuration Parameter types defined as subclasses.			
Base Class(es)	CommonConfigurationAttributes			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
symbolic Name Value	Boolean	1	aggregation	Specifies that this parameter's value is used, together with the aggregating container, to derive a symbolic name definition. E.g.: #define "container_shortName" "this parameter's value".

Table 3.11: ConfigParameter

The use-case for the attribute `symbolicNameValue` will be described in section 3.3.6.5.

In the next sections these different parameter types will be described in detail. The examples for the individual parameters are taken from figure 3.11.

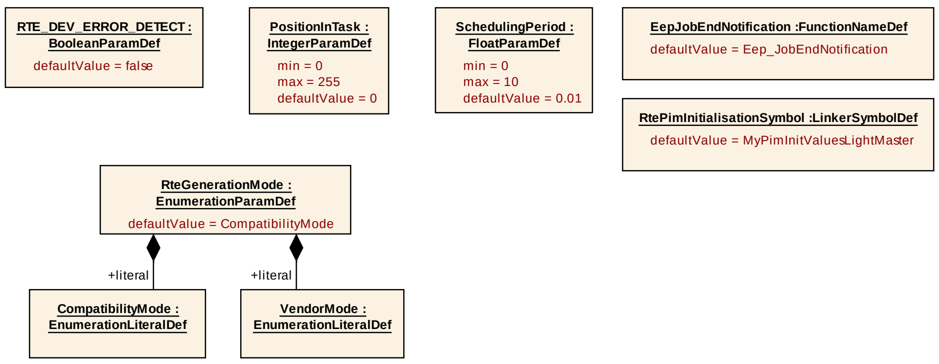
RTE_DEV_ERROR_DETECT : BooleanParamDef
defaultValue = false
PositionInTask : IntegerParamDef
min = 0
max = 255
defaultValue = 0
SchedulingPeriod : FloatParamDef
min = 0
max = 10
defaultValue = 0.01
EepJobEndNotification :FunctionNameDef
defaultValue = Eep_JobEndNotification
RtePimInitialisationSymbol :LinkerSymbolDef
defaultValue = MyPimInitValuesLightMaster
RteGenerationMode : EnumerationParamDef
defaultValue = CompatibilityMode
+literal
+literal
CompatibilityMode : EnumerationLiteralDef
VendorMode : EnumerationLiteralDef

Figure 3.11: Example of parameter definitions using different types

3.3.5.1 Boolean Type

[ecuc_sws_2026] With the `BooleanParamDef` parameter a `true` or `false` parameter can be specified. The only additional attribute is the `defaultValue` which may be specified while defining the parameter.

This parameter is also to be used for other 'boolean'-type configuration parameters which might result into values like:

- `ON / OFF`
- `ENABLE / DISABLE`
- `1 / 0`

The information stored in the ECU Configuration Description is always the same `true/false`, only the generated output will be different depending on actual specification in the BSW SWS.

Class	**BooleanParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Configuration parameter type for Boolean. allowed values are true and false.			
Base Class(es)	ConfigParameter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
default Value	Boolean	0..1	aggregation	Default value of the boolean configuration parameter.

Table 3.12: BooleanParamDef

Example 3.9

```
...
    <BOOLEAN-PARAM-DEF>
      <SHORT-NAME>RTE_DEV_ERROR_DETECT</SHORT-NAME>
      <DEFAULT-VALUE>false</DEFAULT-VALUE>
    </BOOLEAN-PARAM-DEF>
...
```

3.3.5.2 Integer Type

[ecuc_sws_2027] With the `IntegerParamDef` parameter a signed/unsigned whole number can be specified. With the additional attributes `min` and `max` the range of this parameters values in the ECU Configuration Description can be limited[11]. Also the `defaultValue` can be specified.

The value range of the `IntegerParamDef` has two use-cases, signed and unsigned, which both have to fit in a 64-bit number space.

[ecuc_sws_2072] If a signed value is represented the `min` value can be down to `-9223372036854775808` (in hex `0x8000000000000000`[12]) and the `max` value can be up to `9223372036854775807` (in hex `0x7FFFFFFFFFFFFFFF`).

[ecuc_sws_2073] If an unsigned value is represented the `min` value can be down to `0` and the `max` value can be up to `18446744073709551615` (in hex `0xFFFFFFFFFFFFFFFF`).

[ecuc_sws_2074] `IntegerValue` has a lexical representation consisting of an optional sign followed by a finite-length sequence of decimal digits. If the sign is omitted, "+" is assumed.

Class	**IntegerParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Configuration parameter type for Integer.			
Base Class(es)	ConfigParameter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
default Value	Unlimited Integer	0..1	aggregation	Default value of the integer configuration parameter.
max	Unlimited Integer	0..1	aggregation	max value allowed for the parameter defined.
min	Unlimited Integer	0..1	aggregation	min value allowed for the parameter defined.

Table 3.13: IntegerParamDef

Example 3.10

```
...
    <INTEGER-PARAM-DEF>
      <SHORT-NAME>PositionInTask</SHORT-NAME>
      <DEFAULT-VALUE>0</DEFAULT-VALUE>
      <MAX>255</MAX>
      <MIN>0</MIN>
    </INTEGER-PARAM-DEF>
...
```

[11] The `min` and `max` values are defined optional, however in the 'Vendor Specific Module Definition' these values are mandatory.

[12] The hexa-decimal notation is only used for illustrative purposes and can not be used in the actual XML-files.

3.3.5.3 Float Type

[ecuc_sws_2028] To be able to specify parameters with floating number values the `FloatParamDef` can be used. The additional attributes `min`, `max` and `defaultValue` can be specified as well[13].

[ecuc_sws_2075] For the representation the `IEEE double-precision 64-bit floating point` of the *IEEE 754-1985* standard [14] is used.

Class	**FloatParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Configuration parameter type for Float.			
Base Class(es)	ConfigParameter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
default Value	Float	0..1	aggregation	Default value of the float configuration parameter.
max	Float	0..1	aggregation	max value allowed for the parameter defined.
min	Float	0..1	aggregation	min value allowed for the parameter defined.

Table 3.14: FloatParamDef

Example 3.11

```
...
    <FLOAT-PARAM-DEF>
      <SHORT-NAME>SchedulingPeriod</SHORT-NAME>
      <DEFAULT-VALUE>0.01</DEFAULT-VALUE>
      <MAX>10</MAX>
      <MIN>0</MIN>
    </FLOAT-PARAM-DEF>
...
```

[13]The `min` and `max` values are defined optional, however in the 'Vendor Specific Module Definition' these values are mandatory.

3.3.5.4 String Parameter

[ecuc_sws_2029] When a string is needed in the ECU Configuration Description two classes are available. Each string parameter definition might provide a `defaultValue` with the definition.

[ecuc_sws_2030] The restriction on the value of a string parameters and its subclass are the common programming language identifier limitations: start with a letter followed by upper- and lower-case letters, digits and underscores:

```
    identifier := letter ( letter | digit | _ )*
```

where `letter` is `[a-z]` or `[A-Z]` and `digit` is `[0-9]`.

[ecuc_sws_2031] The restriction on the length of the string parameter values is set to 255 characters.

Class	**StringParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Configuration parameter type for String.			
Base Class(es)	ConfigParameter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
default Value	String	0..1	aggregation	Default value of the string configuration parameter.

Table 3.15: StringParamDef

When some text is needed as a parameter the class `StringParamDef` can be used. It does not apply any additional semantics to the parameter.

3.3.5.5 Linker Symbol Parameter

[ecuc_sws_2070] When a parameter represents a linker symbol in the configured software the `LinkerSymbolDef` shall be used. The actual values of the symbol defined will be specified by the implementing software and are not subject to configuration.

The class `LinkerSymbolDef` does not introduce any additional attributes.

The `LinkerSymbolDef` in fact represents the C-compiler symbol which later is translated into a linker symbol. With this element the usage of the `external` declaration of symbols (e.g. variables, constants) is possible.

Class	**LinkerSymbolDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Configuration parameter type for Linker Symbol Names like those used to specify memory locations of variables and constants.			
Base Class(es)	StringParamDef			
Attribute	*Datatype*	*Mul.*	*Link Type*	*Description*

Table 3.16: LinkerSymbolDef

Example 3.12

```
...
    <LINKER-SYMBOL-DEF>
      <SHORT-NAME>RtePimInitializationSymbol</SHORT-NAME>
      <DEFAULT-VALUE>MyPimInitValuesLightMaster</DEFAULT-VALUE>
    </LINKER-SYMBOL-DEF>
...
```

3.3.5.6 Function Name Parameter

[ecuc_sws_2033] When a parameter represents a function name in the configured software the `FunctionNameDef` shall be used. With this feature functions (like callbacks) can be specified.

The class `FunctionNameDef` does not introduce any additional attributes.

Class	**FunctionNameDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Configuration parameter type for Function Names like those used to specify callback functions.			
Base Class(es)	LinkerSymbolDef			
Attribute	*Datatype*	*Mul.*	*Link Type*	*Description*

Table 3.17: FunctionNameDef

Example 3.13

```
...
    <FUNCTION-NAME-DEF>
      <SHORT-NAME>EepJobEndNotification</SHORT-NAME>
      <DEFAULT-VALUE>Eep_JobEndNotification</DEFAULT-VALUE>
    </FUNCTION-NAME-DEF>
...
```

3.3.5.7 Enumeration Parameter

[ecuc_sws_2034] When the parameter can be one choice of several possibilities the `EnumerationParamDef` shall be used. It defines the parameter that will hold the actual value and may also define the `defaultValue` for the enumeration.

Class	**EnumerationParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Configuration parameter type for Enumeration.			
Base Class(es)	ConfigParameter			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
default Value	String	0..1	aggregation	Default value of the enumeration configuration parameter. This string nees to be one of the literals specified for this enumeration.
literal	Enumeration LiteralDef	1..*	aggregation	Aggregation on the literals used to define this enumeration parameter.

Table 3.18: EnumerationParamDef

3.3.5.8 Enumeration Literal Definition

[ecuc_sws_2035] To provide the available choices for the `EnumerationParamDef` the `EnumerationLiteralDef` is used. For each available choice there needs to be one `EnumerationLiteralDef` defined.

[ecuc_sws_2036] For the text used to define the `EnumerationLiteralDef` no additional attribute is needed because the `shortName` inherited from identifiable is used to define the literals.

[ecuc_sws_2054] For the allowed string in `shortName` the restrictions apply as defined in the Model Persistence Rules for XML [8], requirement [APRXML0020].

This basically restricts the `shortName` to only containing the characters `[a-zA-Z][a-zA-Z0-9_]` and have a maximum length of 32 characters. If a more human readable text shall be provided the `longName` can be used which has much more freedom. This requires that configuration tools will show the optional `longName` to the users, see also requirement [ecuc_sws_2088].

The relationship between the `EnumerationParamDef` and the available `EnumerationLiteralDef` is established using aggregations with the role name `literal` at the side of the `EnumerationLiteralDef`.

Class	**EnumerationLiteralDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Configuration parameter type for enumeration literals definition.			
Base Class(es)	Identifiable			
Attribute	*Datatype*	*Mul.*	*Link Type*	*Description*

Table 3.19: EnumerationLiteralDef

Example 3.14

```
...
    <ENUMERATION-PARAM-DEF>
      <SHORT-NAME>RteGenerationMode</SHORT-NAME>
      <DEFAULT-VALUE>CompatibilityMode</DEFAULT-VALUE>
      <LITERALS>
        <ENUMERATION-LITERAL-DEF>
          <SHORT-NAME>CompatibilityMode</SHORT-NAME>
  <LONG-NAME><L-4>Generate in Compatibility Mode</L-4></LONG-NAME>
        </ENUMERATION-LITERAL-DEF>
        <ENUMERATION-LITERAL-DEF>
          <SHORT-NAME>VendorMode</SHORT-NAME>
  <LONG-NAME><L-4>Generate in Vendor Mode</L-4></LONG-NAME>
        </ENUMERATION-LITERAL-DEF>
      </LITERALS>
    </ENUMERATION-PARAM-DEF>
...
```

3.3.6 References in Parameter Definition

There are five kinds of references available for the definition of configuration parameters referring to other entities.

- Reference to other configuration containers within the ECU Configuration Description (see section 3.3.6.1).
- A choice in the referenced configuration container can be specified and the ECU Configuration Description has the freedom (with restrictions) to choose to which target type the reference is pointing to (see section 3.3.6.2).
- Entities outside the ECU Configuration Description can be referenced when they have been specified in a different AUTOSAR Template (see section 3.3.6.3).
- Entities outside the ECU Configuration Description can be referenced using the `instanceRef` semantics defined in the Template UML Profile and Modeling Guide [12] (see section 3.3.6.4).

- A container can be referenced to achieve a symbolic name semantics (see section 3.3.6.5).

The metamodel of those references is shown in figure 3.12.

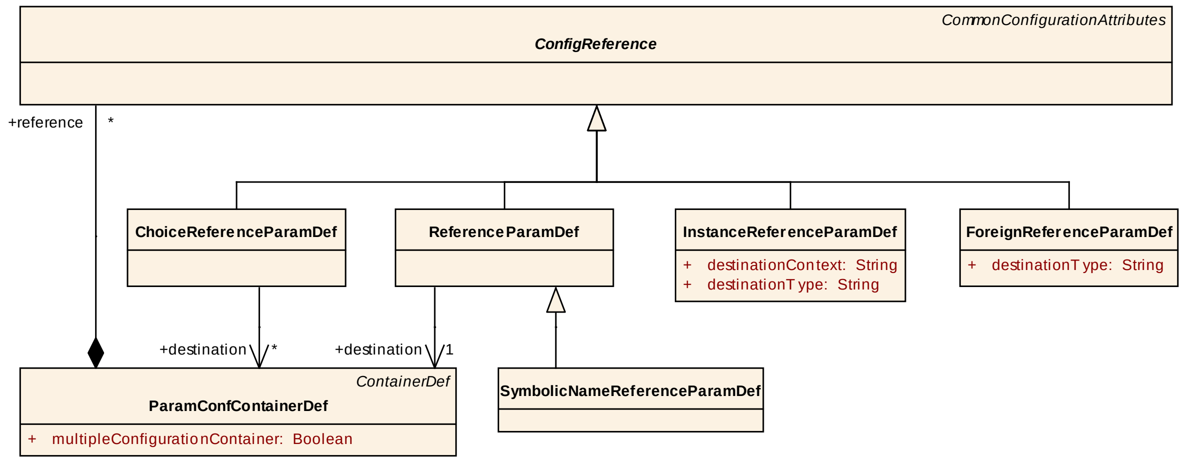
CommonConfigurationAttributes
ConfigReference
+reference
*
ChoiceReferenceParamDef
ReferenceParamDef
InstanceReferenceParamDef
+ destinationContext: String
+ destinationType: String
ForeignReferenceParamDef
+ destinationType: String
+destination
*
+destination
1
ContainerDef
ParamConfContainerDef
+ multipleConfigurationContainer: Boolean
SymbolicNameReferenceParamDef

Figure 3.12: Class diagram for parameter references

[ecuc_sws_2037] The abstract class `ConfigReference` is used to specify the common parts of all reference definitions. `ConfigReference` is an `Identifiable` so it is mandatory to give each reference a name. Also `ConfigReference` is inheriting from `ParamConfMultiplicity` so for each reference definition it can be specified how many such references might be present in the same configuration container later in the ECU Configuration Description.

Class	**ConfigReference (abstract)**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Common class to gather the attributes for the definition of references.			
Base Class(es)	CommonConfigurationAttributes			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.20: ConfigReference

3.3.6.1 Reference

[ecuc_sws_2039] The `ReferenceParamDef` is used to establish references from one `ParamConfContainerDef` to one other specific `ParamConfContainerDef` within the same ECU Configuration Description. For this purpose an object representing the reference has to be used.

[ecuc_sws_2038] The destination for the `ReferenceParamDef` and the `ChoiceReferenceParamDef` is both the `ParamConfContainerDef`. So it is not possible to reference to a specific `ParameterDef` directly but only to its container.

The reason is that there is no use-case where a direct reference to a parameter would be needed.

Class	**ReferenceParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Specify references within the ECU Configuration Description between parameter containers.			
Base Class(es)	ConfigReference			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
destination	ParamConf Container Def	1	reference	Exactly one reference to an parameter container is allowed as destination.

Table 3.21: ReferenceParamDef

The role name at the `ReferenceParamDef` has to be `reference` and the role name at the referenced container has to be `destination` (see figure 3.13 for an example).

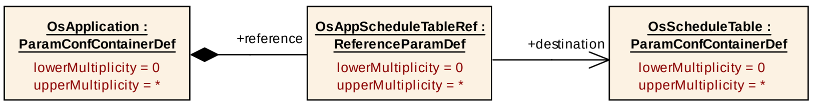

Figure 3.13: Example of an object diagram for a reference

In the example in figure 3.13 the 'OsApplication' is defined to contain references to the 'OsScheduleTable'. The references are called 'OsAppScheduleTableRef' and there can be several such references in the actual ECU Configuration Description document. For the multiplicity of references the multiplicity definition on the `ReferenceParamDef` are relevant (in the example the `lowerMultiplicity` is '0' and the `upperMulti-plicity` is '*'). The multiplicity of the referenced container is not considered for references.

In the ECU Configuration Parameter Definition XML file the `destination` has to be identified unambiguously because the names of configuration parameters are not required to be unique throughout the whole ECU Configuration Parameter Definition. So there might be a parameter defined in the CAN-Driver with the same name as one parameter defined in the ADC-Driver. For this reason the containment hierarchy of the referenced configuration parameter has to be denoted in the definition XML file, as shown in example 3.15. In this example the referenced parameter will be found in the definition of the `Os` module directly in the `AUTOSARParameterDefinition`.

Example 3.15

```
      ...
          <PARAM-CONF-CONTAINER-DEF>
            <SHORT-NAME>OsApplication</SHORT-NAME>
            <REFERENCES>
              <REFERENCE-PARAM-DEF>
                <SHORT-NAME>OsAppScheduleTableRef</SHORT-NAME>
                <DESTINATION-REF DEST="PARAM-CONF-CONTAINER-DEF">
  /AUTOSAR/Os/OsScheduleTable</DESTINATION-REF>
              </REFERENCE-PARAM-DEF>
            </REFERENCES>
          </PARAM-CONF-CONTAINER-DEF>
      ...
```

3.3.6.2 Choice Reference

[ecuc_sws_2040] With the `ChoiceReferenceParamDef` it is possible to define one reference where the destination is specified to be one of several possible kinds. To be able to define such a choice an object of the class `ChoiceReferenceParamDef` has to be aggregated in a container with the role name `reference` at the `ChoiceReferenceParamDef` object.

Class	**ChoiceReferenceParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Specify alternative references where in the ECU Configuration description only one of the specified references will actually be used.			
Base Class(es)	ConfigReference			
Attribute	*Datatype*	*Mul.*	*Link Type*	*Description*
destination	ParamConf Container Def	*	reference	All the possible parameter containers for the reference are specified.

Table 3.22: ChoiceReferenceParamDef

All the available choices are connected via associations with the role name `destination` at the referenced object (see example in figure 3.14.

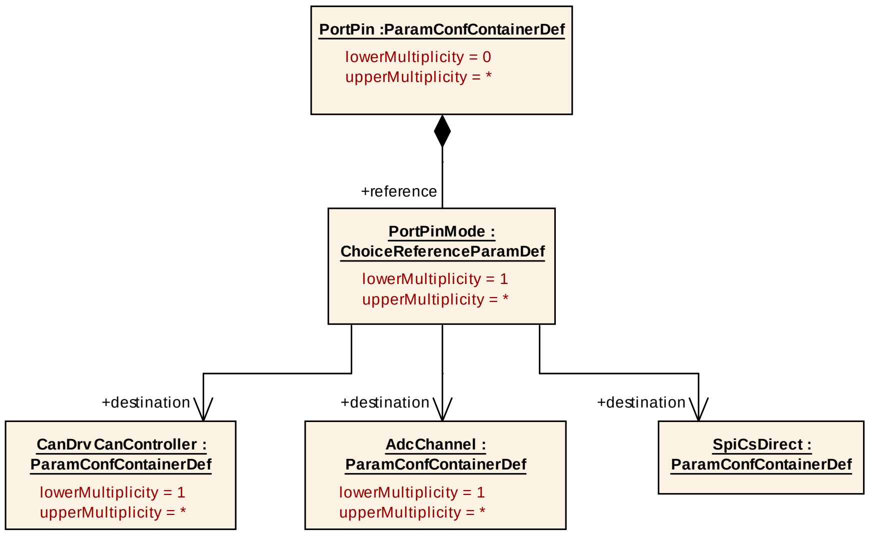

Figure 3.14: Example of an object diagram for a choice reference

In this example an actual instance of the ’PortPinMode’ container can reference one of the three defined containers. Once again the multiplicity is defined by the `ChoiceReferenceParamDef` (here the default ’1’ for lower and upper) and the multiplicities of the referenced containers are not relevant for choice references.

Also the destination needs to be defined unambiguously in the ECU Configuration Parameter Definition XML file like shown in example 3.16.

Example 3.16

```
      ...
        <PARAM-CONF-CONTAINER-DEF>
          <SHORT-NAME>PortPin</SHORT-NAME>
          <REFERENCES>
            <CHOICE-REFERENCE-PARAM-DEF>
              <SHORT-NAME>PortPinMode</SHORT-NAME>
              <DESTINATION-REFS>
                <DESTINATION-REF DEST="PARAM-CONF-CONTAINER-DEF">
  /AUTOSAR/Can/CanDrvCanController</DESTINATION-REF>
                <DESTINATION-REF DEST="PARAM-CONF-CONTAINER-DEF">
  /AUTOSAR/Adc/AdcChannel</DESTINATION-REF>
                <DESTINATION-REF DEST="PARAM-CONF-CONTAINER-DEF">
  /AUTOSAR/Spi/SpiCsDirect</DESTINATION-REF>
              </DESTINATION-REFS>
            </CHOICE-REFERENCE-PARAM-DEF>
          </REFERENCES>
        </PARAM-CONF-CONTAINER-DEF>
      ...
```

In the ECU Configuration Description the actual choice will be taken and there will be only one reference destination left[14].

[14]The `ParamConfMultiplicity` is used to specify the possible occurrences of each reference later in the ECU Configuration Description. The `ChoiceReference` specifies multiple possible destinations for one reference but later in the ECU Configuration Description there can only be exactly one destination described. So the freedom of multiple destinations is only available on the definition of references, if several containers need to be referenced the `ParamConfMultiplicity` has to be set to more than 1, even for the `ChoiceReference`.

3.3.6.3 Foreign Reference

[ecuc_sws_2041] To be able to reference to descriptions of other AUTOSAR templates the parameter definition `ForeignReferenceParamDef` is used. With the attribute `destinationType` the type of the referenced entity has to be specified. The string entered as `destinationType` has to be the name of a M2 class defined in the metamodel under 'M2::AUTOSAR Templates' and the referenced class needs to be derived (directly or indirectly) from `Identifiable`. This guarantees that it is possible to reference to descriptions of such classes. The correctness of the entered string will be checked during the generation of the ECU Configuration Parameter Definition XML file.

Class	**ForeignReferenceParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Specify a reference to an XML description of an entity desribed in another AUTOSAR template.			
Base Class(es)	ConfigReference			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
destination Type	String	1	aggregation	The type in the AUTOSAR Metamodel to which' instance this reference is allowed to point to.

Table 3.23: ForeignReferenceParamDef

[ecuc_sws_2042] Since the AUTOSAR Template UML Profile and Modeling Guide [12] requires the class names of all identifiables to be unique within the AUTOSAR 'M2:: AUTOSAR Templates' metamodel, it is sufficient to provide only the actual class name of the referenced class, as shown in example 3.17.

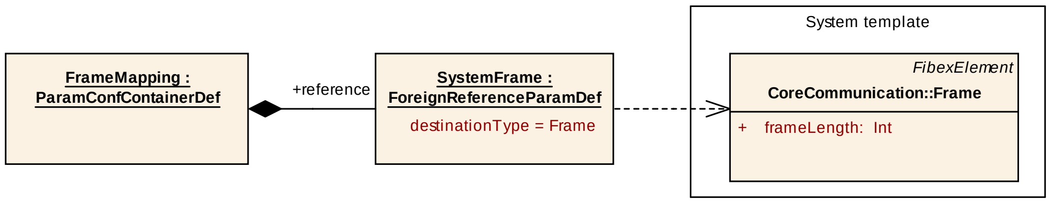

Figure 3.15: Example of an object diagram for a foreign reference

In the example in figure 3.15 the reference is defined to be pointing to a description of a `Frame`. The `Frame` is defined in the System Template metamodel [10] and is derived from `Identifiable`.

Example 3.17

```
...
    <PARAM-CONF-CONTAINER-DEF>
      <SHORT-NAME>FrameMapping</SHORT-NAME>
      <REFERENCES>
        <FOREIGN-REFERENCE-PARAM-DEF>
          <SHORT-NAME>SystemFrame</SHORT-NAME>
```

```
                <DESTINATION-TYPE>Frame</DESTINATION-TYPE>
              </FOREIGN-REFERENCE-PARAM-DEF>
            </REFERENCES>
          </PARAM-CONF-CONTAINER-DEF>
      ...
```

3.3.6.4 Instance Reference

[ecuc_sws_2060] To be able to reference to descriptions of other AUTOSAR templates with the `instanceRef` semantics[15] the parameter definition `InstanceReference-ParamDef` is used. With the attribute `destinationType` the type of the referenced entity has to be specified. With the attribute `destinationContext` the context expression has to be specified.

[ecuc_sws_2082] The string entered as `destinationType` has to be the name of a M2 class defined in the metamodel under 'M2::AUTOSAR Templates' and the referenced class needs to be derived (directly or indirectly) from `Identifiable`. This guarantees that it is possible to reference to instances of such classes.

[ecuc_sws_2083] The string entered as `destinationContext` has to be an ordered list of M2 class names defined in the metamodel under 'M2::AUTOSAR Templates', separated by the `SPACE` character. Additionally the `*` character can be used to indicate none or multiple occurrence of the M2 class BEFORE the `*` character.

Examples of `destinationContext` expressions are:

```
ComponentPrototype RPortPrototype
SoftwareComposition ComponentPrototype* PortPrototype
```

Class	**InstanceReferenceParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Specify a reference to an XML description of an entity desribed in another AUTOSAR template using the INSTANCE REFERENCE semantics.			
Base Class(es)	ConfigReference			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
destination Context	String	1	aggregation	The context in the AUTOSAR Metamodel to which' this reference is allowed to point to.
destination Type	String	1	aggregation	The type in the AUTOSAR Metamodel to which' instance this reference is allowed to point to.

Table 3.24: InstanceReferenceParamDef

[ecuc_sws_2061] Since the AUTOSAR Template UML Profile and Modeling Guide [12] requires the class names of all identifiables to be unique within the AUTOSAR 'M2::

[15]For a detailed description of the `instanceRef` concept please refer to the Template UML Profile and Modeling Guide [12]

AUTOSAR Templates' metamodel, it is sufficient to provide only the actual class names of the referenced class, as shown in example 3.18.

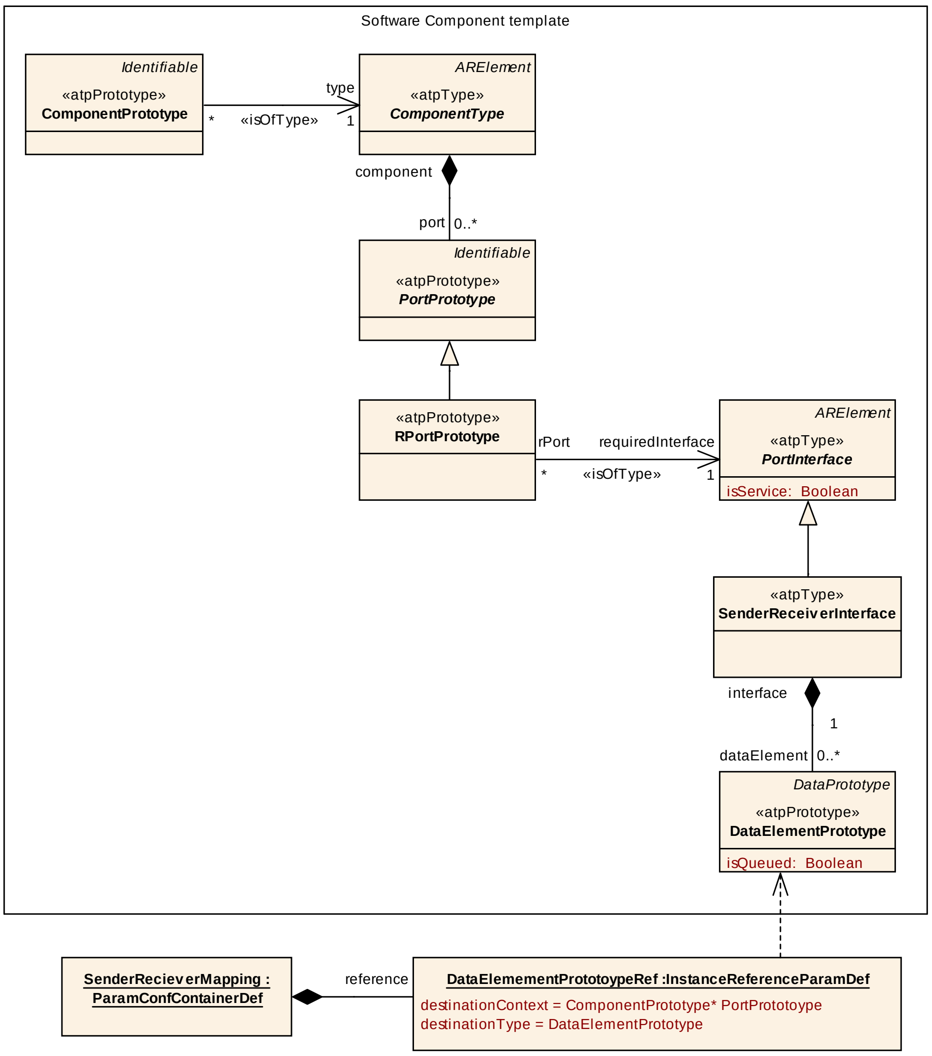

Figure 3.16: Example of an object diagram for an instance reference

In the example in figure 3.16 the reference is defined to be pointing to a description of a 'DataElementPrototype'. The 'DataElementPrototype' is defined in the Software Component Template metamodel [12] and is derived from `Identifiable`. Via the `destinationContext` it is specified that each 'DataElementPrototype' exists in the context of a 'PortPrototype', which itself is in the context of the 'ComponentPrototype'.

Example 3.18

```
...
    <PARAM-CONF-CONTAINER-DEF>
      <SHORT-NAME>SenderReceiverMapping</SHORT-NAME>
      <REFERENCES>
        <INSTANCE-REFERENCE-PARAM-DEF>
          <SHORT-NAME>DataElementPrototypeRef</SHORT-NAME>
```

```
<DESTINATION-CONTEXT>ComponentPrototype* PortPrototype</DESTINATION-CONTEXT>
<DESTINATION-TYPE>DataElementPrototype</DESTINATION-TYPE>
              </INSTANCE-REFERENCE-PARAM-DEF>
            </REFERENCES>
          </PARAM-CONF-CONTAINER-DEF>
      ...
```

Although the ECU Configuration Parameter Definition of the `ForeignReferenceParamDef` and `InstanceReferenceParamDef` are similar there is a difference how those references are represented in the ECU Configuration Description (see section 3.4.5).

3.3.6.5 Symbolic Name Reference

[ecuc_sws_2032] The `SymbolicNameReferenceParamDef` is used to establish the relationship between the user of a symbolic name and the provider of a symbolic name. The object defining the `SymbolicNameReferenceParamDef` is the user and the `destination` of the reference is the provider of the symbolic name.

The `SymbolicNameReferenceParamDef` inherits from `ReferenceParamDef`, so it can be used to point to elements of the kind of `ParamConfContainerDef` within the ECU Configuration Description only.

Class	**SymbolicNameReferenceParamDef**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	This specialization of a ReferenceParamDef specifies that the implementation of the reference is done using a symbolic name defined by the referenced Container's shortName.			
Base Class(es)	ReferenceParamDef			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.25: SymbolicNameReferenceParamDef

[ecuc_sws_2063] If the attribute `symbolicNameValue` of a configuration parameter (see section 3.3.5) is set to `true` this configuration parameter is used as the actual value for the symbolic name. Only one configuration parameter within a container may have this attribute set to `true`.

If the attribute is not present it shall be assumed to be set to `false`.

In the example definition shown in figure 3.17 the `IoHwAb` module can contain several `IoHwAbDemErrors`. Those errors need to be defined in the `Dem` module. And only the `Dem` module is able to define actual numbers associated with these errors when all errors have been specified and collected in the `Dem` module. Those associated values can be stored in the `DemErrorId` parameter which belongs to each `DemError`.

For an example how this is used in the ECU Configuration Description refer to section 3.4.5.2.

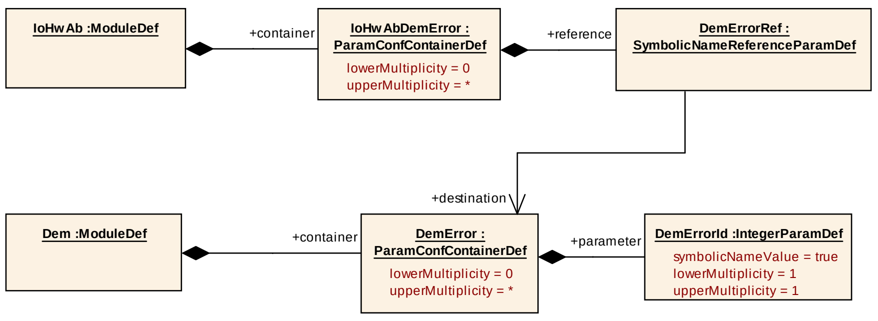

Figure 3.17: Example of an object diagram for a Symbolic Name Reference

Example 3.19

```
      ...
        <MODULE-DEF>
          <SHORT-NAME>IoHwAb</SHORT-NAME>
          <CONTAINERS>
            <PARAM-CONF-CONTAINER-DEF>
              <SHORT-NAME>IoHwAbDemError</SHORT-NAME>
              <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
              <UPPER-MULTIPLICITY>*</UPPER-MULTIPLICITY>
              <REFERENCES>
                <SYMBOLIC-NAME-REFERENCE-PARAM-DEF>
                  <SHORT-NAME>DemErrorRef</SHORT-NAME>
                  <DESTINATION-REF DEST="PARAM-CONF-CONTAINER-DEF">
  /AUTOSAR/Dem/DemError</DESTINATION-REF>
                </SYMBOLIC-NAME-REFERENCE-PARAM-DEF>
              </REFERENCES>
            </PARAM-CONF-CONTAINER-DEF>
          </CONTAINERS>
        </MODULE-DEF>
        <MODULE-DEF>
          <SHORT-NAME>Dem</SHORT-NAME>
          <CONTAINERS>
            <PARAM-CONF-CONTAINER-DEF>
              <SHORT-NAME>DemError</SHORT-NAME>
              <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
              <UPPER-MULTIPLICITY>*</UPPER-MULTIPLICITY>
              <PARAMETERS>
                <INTEGER-PARAM-DEF>
                  <SHORT-NAME>DemErrorId</SHORT-NAME>
                  <LOWER-MULTIPLICITY>1</LOWER-MULTIPLICITY>
                  <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
                  <SYMBOLIC-NAME-VALUE>true</SYMBOLIC-NAME-VALUE>
                </INTEGER-PARAM-DEF>
              </PARAMETERS>
            </PARAM-CONF-CONTAINER-DEF>
          </CONTAINERS>
        </MODULE-DEF>
      ...
```

3.3.7 Derived Parameter Definition

The parameter definitions introduced in the previous sections are meant to define configuration parameter types regardless how the actual values will be captured. But since the ECU Configuration is dependent on lots of other input information many values for the configuration of the BSW and the RTE can be taken over or calculated from other values already available in another description (e.g. the System Extract or the Software-Component description). Such configuration parameters are called Derived Configuration Parameters.

[ecuc_sws_2046] For the major configuration parameter types defined in section 3.3.5 there is a specialization specified which does represent a Derived Configuration Parameter.

The available Derived Configuration Parameters are illustrated in figure 3.18. The abstract class `DerivedValueType` does provide the common attributes for all inheriting parameter definitions.

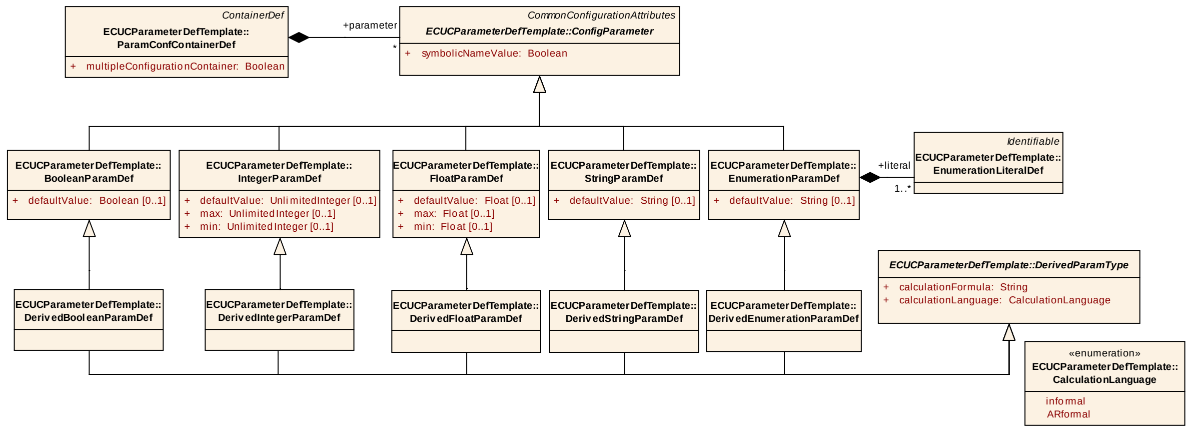
ContainerDef
ECUCParameterDefTemplate::ParamConfContainerDef
+ multipleConfigurationContainer: Boolean
+parameter
*
CommonConfigurationAttributes
ECUCParameterDefTemplate::ConfigParameter
+ symbolicNameValue: Boolean
ECUCParameterDefTemplate::BooleanParamDef
+ defaultValue: Boolean [0..1]
ECUCParameterDefTemplate::IntegerParamDef
+ defaultValue: UnlimitedInteger [0..1]
+ max: UnlimitedInteger [0..1]
+ min: UnlimitedInteger [0..1]
ECUCParameterDefTemplate::FloatParamDef
+ defaultValue: Float [0..1]
+ max: Float [0..1]
+ min: Float [0..1]
ECUCParameterDefTemplate::StringParamDef
+ defaultValue: String [0..1]
ECUCParameterDefTemplate::EnumerationParamDef
+ defaultValue: String [0..1]
+literal
1..*
Identifiable
ECUCParameterDefTemplate::EnumerationLiteralDef
ECUCParameterDefTemplate::DerivedBooleanParamDef
ECUCParameterDefTemplate::DerivedIntegerParamDef
ECUCParameterDefTemplate::DerivedFloatParamDef
ECUCParameterDefTemplate::DerivedStringParamDef
ECUCParameterDefTemplate::DerivedEnumerationParamDef
ECUCParameterDefTemplate::DerivedParamType
+ calculationFormula: String
+ calculationLanguage: CalculationLanguage
«enumeration»
ECUCParameterDefTemplate::CalculationLanguage
informal
ARformal

Figure 3.18: Definition of Derived Parameters

Class	**DerivedParamType (abstract)**			
Package	M2::AUTOSARTemplates::ECUCParameterDefTemplate			
Class Desc.	Allows to define configuration items that are calculated based on the value of * other parameters values * elements (attributes /classes) defined in other AUTOSAR templates such as System template and SW component template. A calculation definition is given which defines how Configuration Editors and Generators can calculate the value from other values when needed to display/use the configuration item. The actual value is stored in the "value" attribute of the derived parameter.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
calculation Formula	String	1	aggregation	Definition of the formula used to calculate the value of the configuration element.
calculation Language	Calculation Language	1	aggregation	Definition of the languag used to specify the formula in attribute calculationFormula. Currentlfy, only informal definition of the formula is supported.

Table 3.26: DerivedParamType

[ecuc_sws_2047] For each Derived Configuration Parameter it can be specified how the parameter will be computed. This is captured in the attribute `calculationFormula`[16] as a plain String.

[16]However currently there is no formal language available to specify the computational rules. It is a major challenge to develop a formal language which is capable of representing the concepts required to

[ecuc_sws_2048] What kind this calculation formula is can be described in the `calculationLanguage` attribute which currently only supports `informal` to represent human language without formal restrictions.

3.3.7.1 Derived Parameter Types

The actual type the Derived Configuration Parameter will be is specified by using a specialized class. Available types are:

- **[ecuc_sws_2049]** Boolean in `DerivedBooleanParamDef`
- **[ecuc_sws_2050]** Integer in `DerivedIntegerParamDef`
- **[ecuc_sws_2051]** Float in `DerivedFloatParamDef`
- **[ecuc_sws_2052]** String in `DerivedStringParamDef`
- **[ecuc_sws_2053]** Enumeration in `DerivedEnumerationParamDef`

which can be instantiated in the ECU Parameter Definition model.

The major advantage of each `DerivedParamType` indirectly inheriting from a `ParameterType` is the compatibility of the two. Even if a parameter currently is defined as a plain `ParameterType` it can be easily redefined to become a `DerivedParamType` in a later version of the parameter definition. The use-case is that currently not all necessary input information might be available and therefore the parameter is defined plainly. But in an later AUTOSAR release more input information might be included in the parameter definition or other AUTOSAR templates and therefore some parameter definitions might be redefined to become `DerivedParamTypes`.

3.3.7.2 Derived Parameter Calculation Formula

For the content of the `calculationFormula` of Derived Configuration Parameters the references do provide essential information. Since the `calculationFormula` provides information how the values can be computed it utilizes the references to address foreign elements to gather the needed information.

The `DerivedParamType` and the `ConfigReference` are both defined within the context of a `ParamConfContainerDef` and therefore belong to the same namespace. In the `calculationFormula` this is used to address the different parts the value needs to be derived from. This is shown in the example 3.20 which represents the XML for figure 3.19.

specify the formula and rules how the derived values are computed. This will NOT be developed in this work package of AUTOSAR.

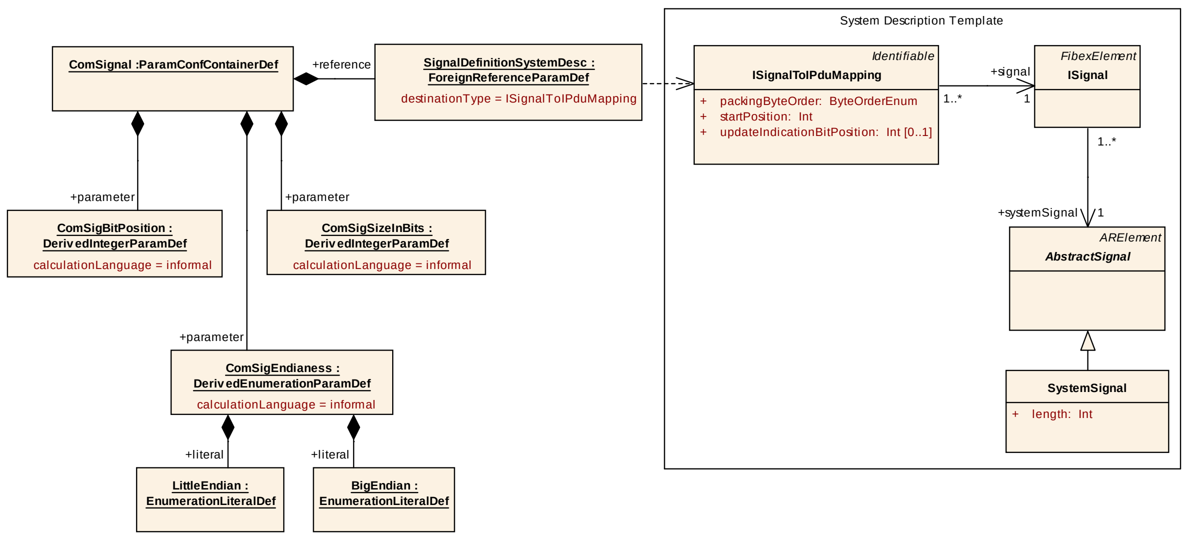
ComSignal :ParamConfContainerDef
+reference
SignalDefinitionSystemDesc : ForeignReferenceParamDef
destinationType = ISignalToIPduMapping
System Description Template
Identifiable
ISignalToIPduMapping
+ packingByteOrder: ByteOrderEnum
+ startPosition: Int
+ updateIndicationBitPosition: Int [0..1]
+signal
1..*
1
FibexElement
ISignal
1..*
+systemSignal
1
ARElement
AbstractSignal
SystemSignal
+ length: Int
+parameter
ComSigBitPosition : DerivedIntegerParamDef
calculationLanguage = informal
+parameter
ComSigSizeInBits : DerivedIntegerParamDef
calculationLanguage = informal
+parameter
ComSigEndianess : DerivedEnumerationParamDef
calculationLanguage = informal
+literal
+literal
LittleEndian : EnumerationLiteralDef
BigEndian : EnumerationLiteralDef

Figure 3.19: Example of derived parameters

In this example the container is called `ComSignal` and it has a foreign reference called `SignalDefinitionSystemDescription` to an instance of type `ISignalToIPduMapping` from the System Description Template. The `ISignalToIPduMapping` has attributes itself and also references to the `ISignal` which itself references to the `SystemSignal`, all in the System Description Template.

The three defined Derived Configuration Parameters have an `informal calculationFormula` which is not shown in the figure but is shown in the XML example.

Example 3.20

```
   ...
    <PARAM-CONF-CONTAINER-DEF>
      <SHORT-NAME>ComSignal</SHORT-NAME>
      <PARAMETERS>
        <DERIVED-INTEGER-PARAM-DEF>
          <SHORT-NAME>ComSigBitPosition</SHORT-NAME>
          <CALCULATION-FORMULA>
value = SignalDefinitionSystemDesc.startPosition
          </CALCULATION-FORMULA>
          <CALCULATION-LANGUAGE>INFORMAL</CALCULATION-LANGUAGE>
        </DERIVED-INTEGER-PARAM-DEF>
        <DERIVED-INTEGER-PARAM-DEF>
          <SHORT-NAME>ComSigSizeInBits</SHORT-NAME>
          <CALCULATION-FORMULA>
value = SignalDefinitionSystemDesc.signal.systemSignal.length
          </CALCULATION-FORMULA>
          <CALCULATION-LANGUAGE>INFORMAL</CALCULATION-LANGUAGE>
        </DERIVED-INTEGER-PARAM-DEF>
        <DERIVED-ENUMERATION-PARAM-DEF>
          <SHORT-NAME>ComSigEndianess</SHORT-NAME>
          <LITERALS>
            <ENUMERATION-LITERAL-DEF>
              <SHORT-NAME>LittleEndian</SHORT-NAME>
            </ENUMERATION-LITERAL-DEF>
            <ENUMERATION-LITERAL-DEF>
              <SHORT-NAME>BigEndian</SHORT-NAME>
            </ENUMERATION-LITERAL-DEF>
          </LITERALS>
          <CALCULATION-FORMULA>
if(SignalDefinitionSystemDesc.packingByteOrder==LittleEndian){
  value = LittleEndian }
if(SignalDefinitionSystemDesc.packingByteOrder==BigEndian){
  value = BigEndian }
          </CALCULATION-FORMULA>
          <CALCULATION-LANGUAGE>INFORMAL</CALCULATION-LANGUAGE>
        </DERIVED-ENUMERATION-PARAM-DEF>
      </PARAMETERS>
      <REFERENCES>
        <FOREIGN-REFERENCE-PARAM-DEF>
          <SHORT-NAME>SignalDefinitionSystemDesc</SHORT-NAME>
          <DESTINATION-TYPE>ISignalToIPduMapping</DESTINATION-TYPE>
        </FOREIGN-REFERENCE-PARAM-DEF>
      </REFERENCES>
    </PARAM-CONF-CONTAINER-DEF>
   ...
```

Although the `calculationFormula` is set to `INFORMAL` there can be some programming language syntax and semantics interpreted.

Since the foreign reference `SignalDefinitionSystemDesc` is present in the same container as the derived parameter definitions it is possible to address the reference directly using its name. In the formula of ComSigBitPosition the attributes of the referenced element can be queried and assigned to the value of this derived parameter.

In the ComSigSizeInBits also the reference is used but then another reference within the System Description is followed with the . operator. Then again the attribute value is assigned to the value of the derived parameter.

In the last example section on ComSigEndianess there is actual some logic described using if clauses to define a mapping from the System Description naming to the ECU Configuration naming.

Those have been just three short examples how a calculationFormula might be composed. Since currently only informal description are supported also a combination of natural language with such references is possible to express more complex relationships.

3.3.7.3 Restrictions on Configuration Class of Derived Parameters

Derived Parameters have to be defined similar to plain configuration parameters which means that also the configuration class has to be specified in the actual implementation of the configuration. But since derived parameters do depend on other information there are certain restrictions applicable which reduce the degree of freedom what kind of configuration class a derived parameter might be.

[ecuc_sws_2055] If the derived parameter is only derived from information coming from other AUTOSAR templates using the ForeignReferenceParamDef or InstanceReferenceParamDef relationship it is assumed that those documents do not change during the configuration process and therefore there are no restrictions on the configuration class of derived parameters.

If the derived parameter is derived from other Configuration Parameters in the ECU Configuration Description then certain rules have to be applied:

- **[ecuc_sws_2058]** If the derived parameter uses information from parameters defined as PreCompile then the derived parameter can be any configuration class.
- **[ecuc_sws_2056]** If the derived parameter uses information from parameters defined as Link then the derived parameter needs to be Link or PostBuild configurable.
- **[ecuc_sws_2057]** If the derived parameter uses information from parameters defined as PostBuild then the derived parameter needs to be PostBuild configurable as well.

3.4 ECU Configuration Description Metamodel

As mentioned in section 3.2 the ECU Configuration Definition metamodel provides the means to declare the parameters and their permitted occurrences within a configuration file. This section will specify the complement to that ECU Configuration Parameter Definition on the actual description side, namely the ECU Configuration Description.

In general, the structure of the ECU Configuration Description metamodel follows the same approach the parameter definition is based upon. One important difference is the fact that there are no special configuration descriptions for `ChoiceReferenceParamDef`, `ForeignReferenceParamDef`, `DerivedParamDef` and `SymbolicNameReferenceParamDef`. How the ECU Configuration Description handles these definition types is described in the appropriate sections below. Because of these prerequisites the ECU Configuration Description metamodel is kept very simple.

The following sections will depict the ECU Configuration Description metamodel. Sections 3.4.1 and 3.4.2 will introduce the top-level structure of a configuration description and the module configurations, whereas the sections 3.4.3, 3.4.4 and 3.4.5 will describe the means to file and structure the actual configuration values.

3.4.1 ECU Configuration Description Top-Level Structure

The top-level entry point to an AUTOSAR ECU Configuration Description is the `EcuConfiguration` (see figure 3.20). Because of the inheritance from `ARElement` the `EcuConfiguration` can be part of an AUTOSAR description like its counterpart the `EcuParameterDefinition` does. A valid `EcuConfiguration` needs to reference the System description (provided as an `ecuExtract`) [10] that specifies the environment in which the configured ECU operates. Additionally it references all Software Module configurations (see section 3.4.2) that are part of this ECU Configuration.

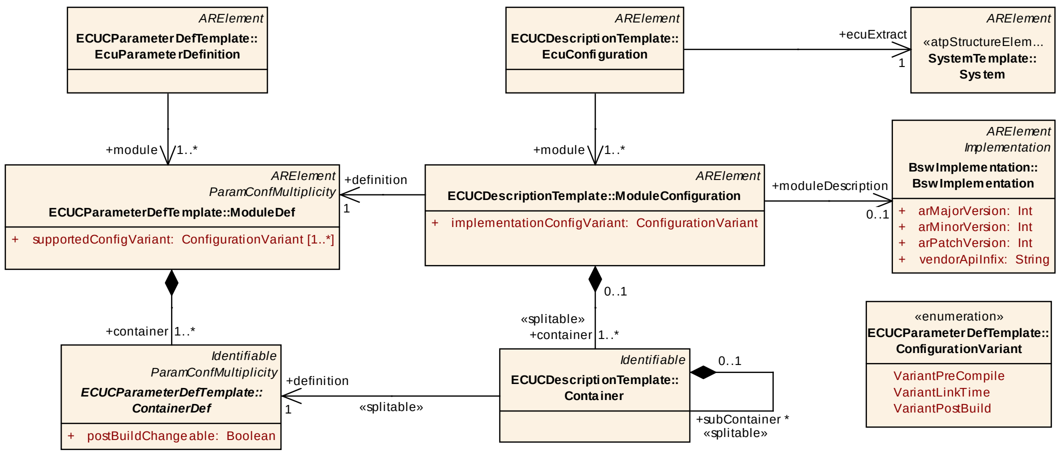

Figure 3.20: ECU Configuration Description Top-Level Structure

Class	**EcuConfiguration**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	This represents the anchor point of the ECU configuration description.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
ecuExtract	System	1	reference	Represents the extract of the System Configuration that is relevant for the ECU configured with that ECU Configuration Description.
ecuSw Composi- tion	EcuSw Composi- tion	1	reference	Reference to the EcuSwComposition which holds the AUTOSAR Service Software Components.
module	Module Configura- tion	1..*	reference	References to the configuration of individual software modules that are present on this ECU.

Table 3.27: EcuConfiguration

3.4.2 Module Configurations

[ecuc_sws_3016] The `ModuleConfiguration` subsumes all configuration objects that belong to one managed Software Module, namely Application Software Components, BSW modules, RTE and generic ECU Configuration artifacts (e.g. memory maps).

[ecuc_sws_2089] The `ModuleConfiguration` aggregates the `Container` with the role `container` and the stereotype `<<splitable>>` which allows the content of a `ModuleConfiguration` to be split among several XML-Files (see also section 3.4.2.1).

[ecuc_sws_3017] If the `ModuleConfiguration` holds the configuration values of a BSW module, a reference to the according `BswImplementation` shall be provided.

The reference is established to the `BswImplementation` because this is the most detailed information available for the configuration.

[ecuc_sws_3035] The reference `definition` assigns the `ModuleConfiguration` to the according `ModuleDef` it is depending on.

[ecuc_sws_3031] The `ModuleDef`, to which the `ModuleConfiguration` is associated to, is specified by the implementor of the according Software Module. Therefore the `ModuleDef` includes standardized as well as vendor-specific parameter definitions.

Class	**ModuleConfiguration**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Head of the configuration of one Module. A Module can be a BSW module as well as the RTE and ECU Infrastructure. As part of tthe BSW module description, the ModuleConfiguration has two different roles: The recommendedConfiguration contains parameter values recommended by the BSW module vendor. The preconfiguredConfiguration contains values for those parameters which are fixed by the implementation and cannot be changed. These two ModuleConfigurations are used when the base ModuleConfiguration (as part of the base ECU configuration) is created to fill parameters with initial values.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
container	Container	1..*	aggregation	Aggregates all containers that belong to this module configuration.
definition	ModuleDef	1	reference	Reference to the definition of this ModuleConfiguration. Typically, this is a vendor specific module configuration.
implementationConfigVariant	ConfigurationVariant	1	aggregation	Specifies the ConfigurationVariant used for this ModuleConfiguration.
moduleDescription	BswImplementation	0..1	reference	Referencing the BSW module description, which this ModuleConfiguration is configuring. This is optional because the ModuleConfiguration is also used to configure the ECU infrastructure (memory map) or Application SW-Cs.

Table 3.28: ModuleConfiguration

Figure 3.21 depicts the different associations between the ModuleConfiguration and the Basic Software Module Description. The BswImplementation may specify a vendor specific pre-configured configuration description (preconfiguredConfiguration) that includes the configuration values already assigned by the implementor of the Software Module and a vendor specific recommended configuration description (recommendedConfiguration) that can be used to initialize configuration editors.

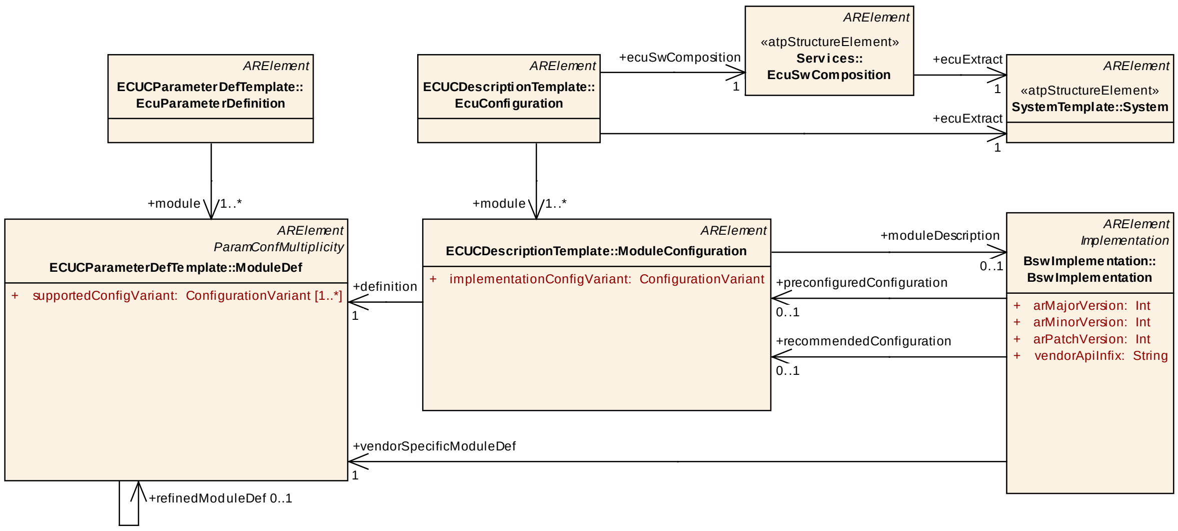

Figure 3.21: Dependencies of ModuleConfigurations

[ecuc_sws_2103] The implementationConfigVariant specifies which configuration variant has been chosen for this ModuleConfiguration. The choice is taken from the supportedConfigVariant elements specified in the ModuleDef associated to this ModuleConfiguration.

The element supportedConfigVariant is described in section 3.3.2 and section 3.3.4.2.2.

To illustrate the structure of an ECU Configuration Description example 3.21 depicts the top-level structure of an ECU Configuration Description XML file that conforms to the ECU Configuration Definition XML file that was presented in example 3.3.

The only supportedConfigVariant of example 3.3 is taken for the implementationConfigVariant element.

Example 3.21

```
<AUTOSAR>
  <TOP-LEVEL-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>ECUC</SHORT-NAME>
      <ELEMENTS>
        <ECU-CONFIGURATION>
          <SHORT-NAME>Configuration</SHORT-NAME>
          <ECU-EXTRACT-REF DEST="SYSTEM">
            /some_package/.../theEcuExtractForEcuXY
          </ECU-EXTRACT-REF>
          <MODULE-REFS>
            <MODULE-REF DEST="MODULE-CONFIGURATION>
              /ECUC/theRteConfig
            </MODULE-REF>
          </MODULE-REFS>
        </ECU-CONFIGURATION>
        <MODULE-CONFIGURATION>
          <SHORT-NAME>theRteConfig</SHORT-NAME>
          <DEFINITION-REF DEST="MODULE-DEF">/AUTOSAR/Rte</DEFINITION-REF>
          <IMPLEMENTATION-CONFIG-VARIANT>
            VARIANT-PRE-COMPILE
          </IMPLEMENTATION-CONFIG-VARIANT>
          <MODULE-DESCRIPTION-REF DEST="BSW-MODULE-DESCRIPTION">
            /some_package/.../theUsed_Rte_BSWModuleDescription
          </MODULE-DESCRIPTION-REF>
          <CONTAINERS>
            <!-- ... -->
          </CONTAINERS>
        </MODULE-CONFIGURATION>
      </ELEMENTS>
    </AR-PACKAGE>
  </TOP-LEVEL-PACKAGES>
</AUTOSAR>
```

3.4.2.1 Splitabe ModuleConfiguration

In the document *Specification of Interoperability of Authoring Tools* [11] requirement `[ATI0042]` it is specified that the elements of an aggregation are allowed to be split over several XML files if the relationship is marked with the stereotype `<<splitable>>`.

The stereotype `<<splitable>>` has been introduced to support the delivery of *one* module's `ModuleConfiguration` in *several* XML files, see also section 2.3.2.3 for use-cases.

In Example 3.22 a simple definition of a module's configuration parameters is shown. It just consists of one container which has two parameters, one parameter defined to be `LINK` time configurable, the other parameter is `POST-BUILD` time configurable. The values for these parameters are defined in different process steps and therefore two XML files can be used to describe both values.

In example 3.23 the value for the LINK time parameter SignalSize is specified, while in exmaple 3.24 the POST-BUILD parameter's BitPosition value is given.

The XML structure in both ModuleConfiguration XML files is equivalent with respect to the packages and containers. In both XML files a container with the name theSignal is defined. It is up to the configuration tool to *merge* the content of these two files into one model. Also is the number of possible XML files not limited, so it would be possible (although probably not reasonable) to put each parameter value into one XML file.

Example 3.22

```
  ...
    <MODULE-DEF>
      <SHORT-NAME>Com</SHORT-NAME>
      <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
      <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
<IMPLEMENTATION-CONFIG-VARIANT>Variant3</IMPLEMENTATION-CONFIG-VARIANT>
      <CONTAINERS>
        <PARAM-CONF-CONTAINER-DEF>
          <SHORT-NAME>ComSignal</SHORT-NAME>
          <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
          <UPPER-MULTIPLICITY>*</UPPER-MULTIPLICITY>
<MULTIPLE-CONFIGURATION-CONTAINER>false</MULTIPLE-CONFIGURATION-CONTAINER>
          <PARAMETERS>
            <INTEGER-PARAM-DEF>
              <SHORT-NAME>SignalSize</SHORT-NAME>
<IMPLEMENTATION-CONFIG-CLASS>LINK</IMPLEMENTATION-CONFIG-CLASS>
              <ORIGIN>AUTOSAR_ECUC</ORIGIN>
            </INTEGER-PARAM-DEF>
            <INTEGER-PARAM-DEF>
              <SHORT-NAME>BitPosition</SHORT-NAME>
<IMPLEMENTATION-CONFIG-CLASS>POST-BUILD</IMPLEMENTATION-CONFIG-CLASS>
              <ORIGIN>AUTOSAR_ECUC</ORIGIN>
            </INTEGER-PARAM-DEF>
          </PARAMETERS>
        </PARAM-CONF-CONTAINER-DEF>
      </CONTAINERS>
    </MODULE-DEF>
  ...
```

Example 3.23

```
<AUTOSAR>
  <TOP-LEVEL-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>ECUC</SHORT-NAME>
      <ELEMENTS>
        <MODULE-CONFIGURATION>
          <SHORT-NAME>theComConfig</SHORT-NAME>
          <DEFINITION-REF DEST="MODULE-DEF">/Vendor/Com</DEFINITION-REF>
          <MODULE-DESCRIPTION-REF DEST="BSW-MODULE-DESCRIPTION">
            /some_package/some_path/theUsed_Com_BSWModuleDescription
          </MODULE-DESCRIPTION-REF>
          <CONTAINERS>
            <CONTAINER>
              <SHORT-NAME>theSignal</SHORT-NAME>
              <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
                /Vendor/Com/ComSignal
              </DEFINITION-REF>
              <PARAMETER-VALUES>
                <INTEGER-VALUE>
                  <DEFINITION-REF DEST="INTEGER-PARAM-DEF">
                    /Vendor/Com/ComSignal/SignalSize
                  </DEFINITION-REF>
                  <VALUE>8</VALUE>
                </INTEGER-VALUE>
              </PARAMETER-VALUES>
            </CONTAINER>
          </CONTAINERS>
        </MODULE-CONFIGURATION>
      </ELEMENTS>
    </AR-PACKAGE>
  </TOP-LEVEL-PACKAGES>
</AUTOSAR>
```

Example 3.24

```
<AUTOSAR>
  <TOP-LEVEL-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>ECUC</SHORT-NAME>
      <ELEMENTS>
        <MODULE-CONFIGURATION>
          <SHORT-NAME>theComConfig</SHORT-NAME>
          <DEFINITION-REF DEST="MODULE-DEF">/Vendor/Com</DEFINITION-REF>
          <MODULE-DESCRIPTION-REF DEST="BSW-MODULE-DESCRIPTION">
            /some_package/some_path/theUsed_Com_BSWModuleDescription
          </MODULE-DESCRIPTION-REF>
          <CONTAINERS>
            <CONTAINER>
              <SHORT-NAME>theSignal</SHORT-NAME>
              <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
                /Vendor/Com/ComSignal
              </DEFINITION-REF>
              <PARAMETER-VALUES>
                <INTEGER-VALUE>
                  <DEFINITION-REF DEST="INTEGER-PARAM-DEF">
                    /Vendor/Com/ComSignal/BitPosition
                  </DEFINITION-REF>
                  <VALUE>4</VALUE>
                </INTEGER-VALUE>
              </PARAMETER-VALUES>
            </CONTAINER>
          </CONTAINERS>
        </MODULE-CONFIGURATION>
      </ELEMENTS>
    </AR-PACKAGE>
  </TOP-LEVEL-PACKAGES>
</AUTOSAR>
```

3.4.3 Parameter Container Description

Symmetrically to the parameter container definition (see section 3.3.3) the parameter container description is specified to group other containers, parameter values and references. Figure 3.22 depicts the general structure of the configuration container description and its association to the configuration definition. The dependencies reflect the direct relationship between a `Container` and a `ContainerDef` as well as a `ParameterValue` and a `ParameterType`.

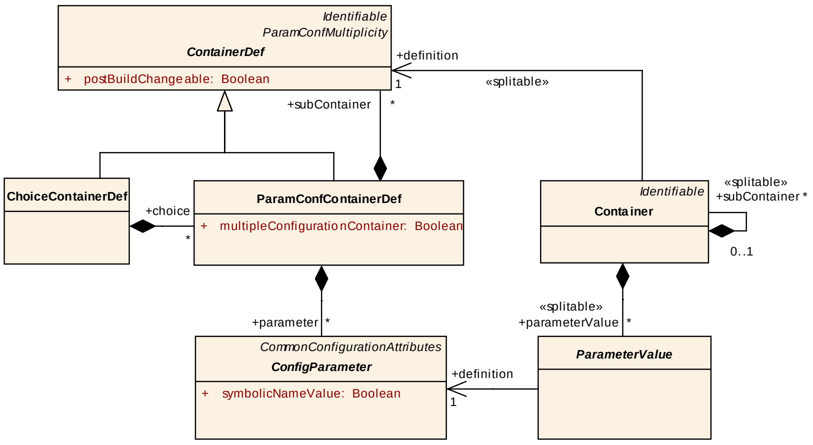

Figure 3.22: Parameter container description

[ecuc_sws_3012] The `Container` inherits from `Identifiable` defining a namespace for all `Container`, `ParameterValue` and `ReferenceValue` that belong to that `Container`.

[ecuc_sws_3019] The reference `definition` assigns the `Container` to the according `ContainerDef`[17] it is depending on.

If the configuration description would be provided without an according configuration definition an editor could not reconstruct what kind of `ContainerDef` a `Container` is based upon.

[ecuc_sws_3011] If a `ContainerDef` has specified a `lowerMultiplicity` < 1 the corresponding `Container` may be omitted in the ECU Configuration Description because of being treated as optional.

[17]including all `ContainerDef`'s decendants

Class	**Container**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Represents a Container definition in the ECU Configuration Description.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
definition	ParamConf Container Def	1	reference	Reference to the definition of this Container in the ECU Configuration Parameter Definition.
parameter Value	Parameter Value	*	aggregation	Aggregates all ECU Configuration Values within this Container.
reference Value	Config Reference Value	*	aggregation	Aggregates all References with this container.
subCon-tainer	Container	*	aggregation	Aggregates all sub-containers within this container.

Table 3.29: Container

[ecuc_sws_2092] If a `ParamConfContainerDef` is specified to be the `multipleConfigurationContainer` there can be several `Container` elements defined in the ECU Configuration. Each `Container shortName` does specify the name of the configuration set it contains.

The `multipleConfigurationContainer` is further detailed in section 3.4.7.

In example 3.25 a snippet of an ECU Configuration Description XML file is shown that conforms to the ECU Configuration Parameter Definition described in example 3.4. The container `RteGeneration` is specified to have an `upperMultiplicity` of `1`, so there can only be one `Container` representation. The container `SwComponentInstance` has an `upperMultiplicity` of `*`, so there can be several representations of this `Container`.

Example 3.25

```
...
  <MODULE-CONFIGURATION>
    <SHORT-NAME>theRteConfig</SHORT-NAME>
    <DEFINITION-REF DEST="MODULE-DEF">/AUTOSAR/Rte</DEFINITION-REF>
    <MODULE-DESCRIPTION-REF DEST="BSW-MODULE-DESCRIPTION">
      /some_package/some_path/theUsed_Rte_BSWModuleDescription
    </MODULE-DESCRIPTION-REF>
    <CONTAINERS>
      <CONTAINER>
        <SHORT-NAME>theGeneration</SHORT-NAME>
        <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/Rte/RteGeneration
        </DEFINITION-REF>
        <SUB-CONTAINERS>
          <!-- ... -->
        </SUB-CONTAINERS>
      </CONTAINER>
      <CONTAINER>
        <SHORT-NAME>SwcInstance1</SHORT-NAME>
        <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/Rte/SwComponentInstance
        </DEFINITION-REF>
        <SUB-CONTAINERS>
          <!-- ... -->
        </SUB-CONTAINERS>
      </CONTAINER>
      <CONTAINER>
        <SHORT-NAME>SwcInstance2</SHORT-NAME>
        <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/Rte/SwComponentInstance
        </DEFINITION-REF>
        <SUB-CONTAINERS>
          <!-- ... -->
        </SUB-CONTAINERS>
      </CONTAINER>
    </CONTAINERS>
  </MODULE-CONFIGURATION>
...
```

3.4.3.1 Choice Containers

[ecuc_sws_3020] In the ECU Configuration Parameter Definition the container choices are specified as part of the `ChoiceContainerDef`. On the description side a `ChoiceContainerDef` is treated as a usual container, though it depends on the `upperMultiplicity` of the `ChoiceContainerDef` how often the choice can be taken. Which choice has been taken is defined by the `<DEFINITION-REF>` of the `<SUB-CONTAINER>`.

Example 3.26 depicts the notation of a filled out `ChoiceContainerDef` as described in example 3.5.

Example 3.26

```
...
  <CONTAINER>
    <SHORT-NAME>OsAlarm</SHORT-NAME>
    <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
      /AUTOSAR/Os/OsApplication/OsAlarm
    </DEFINITION-REF>
    <SUB-CONTAINERS>
      <CONTAINER>
        <SHORT-NAME>myOsAlarmAction</SHORT-NAME>
        <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/Os/OsApplication/OsAlarm/OsAlarmAction
        </DEFINITION-REF>
        <SUB-CONTAINERS>
          <CONTAINER>
            <SHORT-NAME>myTaskActivation</SHORT-NAME>
            <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
/AUTOSAR/Os/OsApplication/OsAlarm/OsAlarmAction/OsAlarmActivateTask
            </DEFINITION-REF>
            <!--...-->
          </CONTAINER>
        </SUB-CONTAINERS>
      </CONTAINER>
    </SUB-CONTAINERS>
  </CONTAINER>
...
```

3.4.4 Parameter Values

Symmetrically to the Configuration Parameter Definitions (as specified in section 3.3.5) there do exist metamodel classes which represent the actual configured values.

[ecuc_sws_3006] All these metamodel classes are derived from `ParameterValue` (see figure 3.23).

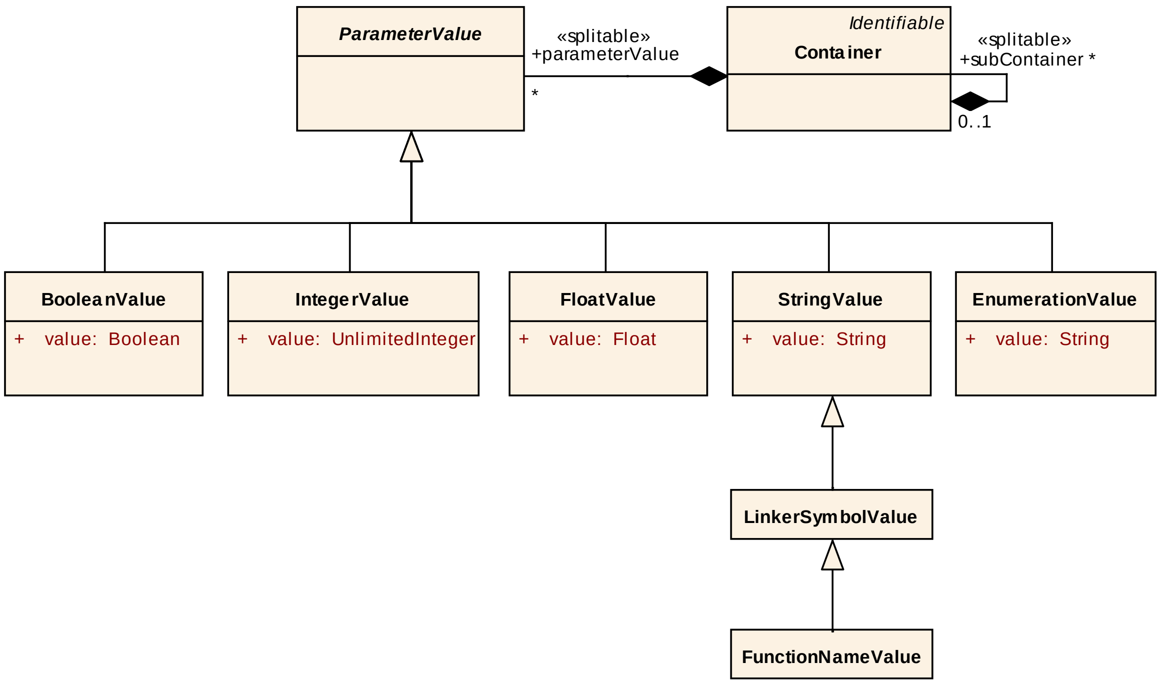

Figure 3.23: Parameter description

[ecuc_sws_3007] All inherited metamodel classes representing an ECU Configuration Value specify an attribute `value` that stores the configuration value in XML-based description.

[ecuc_sws_3009] If a `defaultValue` is specified in the ECU Configuration Parameter Definition that given value can be used as the initial `value` of the according `ParameterValue` for the ECU Configuration Description as explained in section 5.2.

[ecuc_sws_3034] In a well-formed and completed ECU Configuration Description each provided parameter needs to have a `value` specified even if it is just copied from the `defaultValue` of the ECU Configuration Definition.

For further rules how a `value` can be provided if no `defaultValue` is specified in the ECU Configuration Definition see section 5.2.

[ecuc_sws_3038] The reference `destination` assigns the `ParameterValue`[18] to the according `ParameterType` it is depending on.

[18] and all its descendants

[ecuc_sws_3010] If an ECU Configuration Parameter has specified a `lowerMultiplicity` $<$ 1 an ECU Configuration Value may be omitted in the ECU Configuration Description because of being treated as optional.

Class	**ParameterValue (abstract)**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Common class to all types of configuration values			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
definition	Config Parameter	1	reference	Reference to the definition of this ParameterValue subclasses in the ECU Configuration Parameter Definition.

Table 3.30: ParameterValue

All specialized parameter values that are derived from `ParameterValue` are described in the sections below. The used examples represent configuration values that correspond to their appropriate parameter definition examples depicted in figure 3.11 and their XML representation.

3.4.4.1 Boolean Values

[ecuc_sws_3000] A `BooleanValue` stores a configuration value that is of definition type `BooleanParamDef`.

Class	**BooleanValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Representing a configuration value of definition type BooleanParamDef			
Base Class(es)	ParameterValue			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
value	Boolean	1	aggregation	Stores the value of the Boolean parameter.

Table 3.31: BooleanValue

Example 3.27 depicts the configuration description of definition type `BooleanParamDef` for example 3.9.

Example 3.27

```
...
  <BOOLEAN-VALUE>
    <DEFINITION-REF DEST="BOOLEAN-PARAM-DEF">
      /AUTOSAR/Rte/RteGeneration/RTE_DEV_ERROR_DETECT
    </DEFINITION-REF>
    <VALUE>true</VALUE>
  </BOOLEAN-VALUE>
...
```

3.4.4.2 Integer Values

[ecuc_sws_3001] An `IntegerValue` stores a configuration value that is of definition type `IntegerParamDef`.

[ecuc_sws_3040] The value has to be specified in signed decimal notation without decimal point.

Class	**IntegerValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Representing a configuration value of definition type IntegerParamDef			
Base Class(es)	ParameterValue			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
value	Unlimited Integer	1	aggregation	Stores the value of the Integer parameter.

Table 3.32: IntegerValue

Example 3.28 depicts the configuration description of definition type `IntegerParamDef` for example 3.10.

Example 3.28

```
...
  <INTEGER-VALUE>
    <DEFINITION-REF DEST="INTEGER-PARAM-DEF">
      /AUTOSAR/Rte/RunnableEntityMapping/PositionInTask
    </DEFINITION-REF>
    <VALUE>5</VALUE>
  </INTEGER-VALUE>
...
```

3.4.4.3 Float Values

[ecuc_sws_3002] A `FloatValue` stores a configuration value that is of definition type `FloatParamDef`.

Class	**FloatValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Representing a configuration value of definition type FloatParamDef			
Base Class(es)	ParameterValue			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
value	Float	1	aggregation	Stores the value of the Float parameter.

Table 3.33: FloatValue

Example 3.29 depicts the configuration description of definition type `FloatParamDef` for example 3.11.

Example 3.29

```
...
  <FLOAT-VALUE>
    <DEFINITION-REF DEST="FLOAT-PARAM-DEF">
      /AUTOSAR/Rte/RunnableEntityMapping/SchedulingPeriod
    </DEFINITION-REF>
    <VALUE>74.8</VALUE>
  </FLOAT-VALUE>
...
```

3.4.4.4 String Values

[ecuc_sws_3003] A `StringValue` stores a configuration value that is of definition type `StringParamDef`.

Class	**StringValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Representing a configuration value of definition type StringParamDef			
Base Class(es)	ParameterValue			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
value	String	1	aggregation	Stores the value of the String parameter.

Table 3.34: StringValue

3.4.4.5 Linker Symbol Values

[ecuc_sws_3041] A `LinkerSymbolValue` stores a configuration value that is of definition type `LinkerSymbolParameter`.

Class	**LinkerSymbolValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Representing a configuration value of definition type LinkerSymbolDef			
Base Class(es)	StringValue			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.35: LinkerSymbolValue

Example 3.30 depicts the configuration description of definition type `LinkerSymbol-Def` for example 3.12.

Example 3.30

```
...
  <LINKER-SYMBOL-VALUE>
    <DEFINITION-REF DEST="LINKER-SYMBOL-DEF">
      /AUTOSAR/Rte/Resource/Pim/RtePimInitializationSymbol
    </DEFINITION-REF>
    <VALUE>MyPimInitValuesLightMaster</VALUE>
  </LINKER-SYMBOL-VALUE>
...
```

3.4.4.6 Function Name Values

[ecuc_sws_3005] A `FunctionNameValue` stores a configuration value that is of definition type `FunctionNameParamDef`.

Class	**FunctionNameValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Representing a configuration value of definition type FunctionNameDef			
Base Class(es)	LinkerSymbolValue			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 3.36: FunctionNameValue

Example 3.31 depicts the configuration description of definition type `FunctionName-Def` for example 3.13.

Example 3.31

```
...
  <FUNCTION-NAME-VALUE>
    <DEFINITION-REF DEST="FUNCTION-NAME-DEF">
      /AUTOSAR/Eep/EepInitConfiguration/EepJobEndNotification
    </DEFINITION-REF>
    <VALUE>Eep_VendorXY_JobEndNotification</VALUE>
  </FUNCTION-NAME-VALUE>
...
```

3.4.4.7 Enumeration Values

[ecuc_sws_3005] A `EnumerationValue` stores a configuration value that is of definition type `EnumerationParamDef`.

Class	**EnumerationValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Representing a configuration value of definition type EnumerationParamDef			
Base Class(es)	ParameterValue			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
value	String	1	aggregation	Stores the chosen literal.

Table 3.37: EnumerationValue

Example 3.32 depicts the configuration description of definition type `Enumeration-ParamDef` for example 3.14.

Example 3.32

```
...
  <ENUMERATION-VALUE>
    <DEFINITION-REF DEST="ENUMERATION-PARAM-DEF">
      /AUTOSAR/Rte/RteGeneration/RteGenerationMode
    </DEFINITION-REF>
    <VALUE>CompatibilityMode</VALUE>
  </ENUMERATION-VALUE>
...
```

3.4.5 References in the ECU Configuration Metamodel

Figure 3.24 depicts the ECU Configuration Metamodel to reference other description elements.

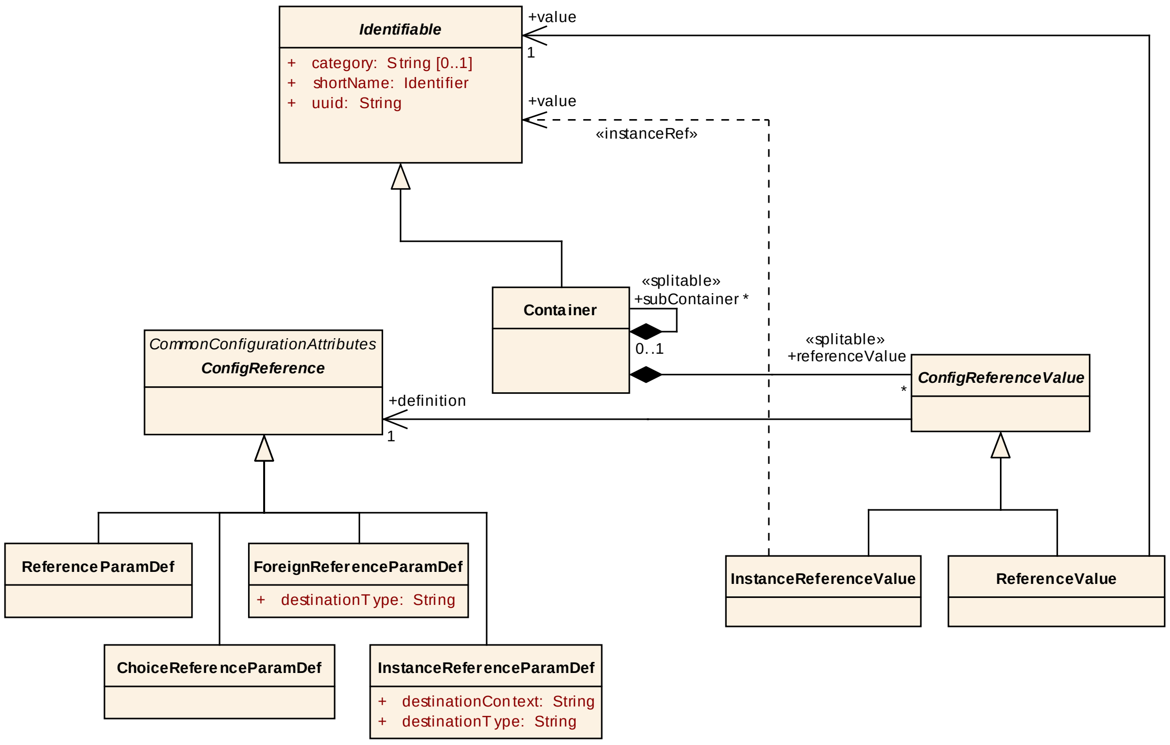

Figure 3.24: Parameter references

[ecuc_sws_3032] The metamodel class `ConfigReferenceValue` acts as the generalization of all reference types in the ECU Configuration Description.

[ecuc_sws_3039] The reference `destination` assigns the `ConfigReferenceValue`[19] to the according `ConfigReference` it is depending on.

[ecuc_sws_3030] If a `ConfigReference` has specified a `lowerMultiplicity` $<$ 1 an according `ConfigReferenceValue` may be omitted in the ECU Configuration Description because of being treated as optional.

[19]and all its descendants

Class	**ConfigReferenceValue (abstract)**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Abstract class to be used as common parent for all reference values in the ECU Configuration Description.			
Base Class(es)	ARObject			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
definition	Config Reference	1	reference	Reference to the definition of this ConfigReferenceValue subclasses in the ECU Configuration Parameter Definition.

Table 3.38: ConfigReferenceValue

[ecuc_sws_3027] The metamodel class `ReferenceValue` provides the mechanism to reference to any model element of type `Identifiable`.

[ecuc_sws_3028] Therefore this class provides the means to describe all kinds of reference definitions except an `InstanceReferenceParamDef`, which is described in section 3.4.5.1 in more detail.

[ecuc_sws_3029] A `ChoiceReferenceParamDef` translates to a `ReferenceValue` in the ECU Configuration Description because the choice has to be resolved in that description. Therefore no special configuration description type is introduced.

Class	**ReferenceValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	Used to represent a configuration value that has a parameter definition of type ConfigReference (used for all of its specializations excluding InstanceReferenceParamDef).			
Base Class(es)	ConfigReferenceValue			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
value	Identifiable	1	reference	Specifes the destination of the reference.

Table 3.39: ReferenceValue

[ecuc_sws_2093] If a `ConfigReferenceValue` references a container within some `ModuleConfiguration` the referenced container shall be part of a `ModuleConfiguration` which is itself part of the `EcuConfiguration`.

According to figure 3.20 a `ModuleConfiguration` is part of the `EcuConfiguration` if it is referenced with the `module` role.

The following examples will picture that `ReferenceValue` can be used to represent most of the specializations of `ConfigReference` (namely `ReferenceParamDef`, `ChoiceReferenceParamDef`, `ForeignReferenceParamDef` and `SymbolicNameReferenceParamDef`).

Example 3.33 depicts the configuration description of definition type `ReferenceParamDef` for example 3.15.

Example 3.33

```
...
  <CONTAINER>
    <SHORT-NAME>myOsApplication</SHORT-NAME>
    <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
      /AUTOSAR/Os/OsApplication
    </DEFINITION-REF>
    <REFERENCE-VALUES>
      <REFERENCE-VALUE>
        <DEFINITION-REF DEST="REFERENCE-PARAM-DEF">
          /AUTOSAR/Os/OsApplication/OsAppScheduleTableRef
        </DEFINITION-REF>
        <VALUE-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /ECUC/myOs/myOsScheduleTable1
        </VALUE-REF>
      </REFERENCE-VALUE>
    </REFERENCE-VALUES>
  </CONTAINER>
...
```

Example 3.34 depicts the configuration description of definition type `ChoiceReferenceParamDef` for example 3.16. To illustrate the usage of a `ChoiceReferenceParamDef` in more detail, this example takes advantage of the fact that a `PortPin` may be used in several modes at once. Therefore it has multiple references of different type.

Example 3.34

```
...
  <CONTAINER>
    <SHORT-NAME>myPortPin</SHORT-NAME>
    <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
      /AUTOSAR/some_module/PortPin
    </DEFINITION-REF>
    <REFERENCE-VALUES>
      <REFERENCE-VALUE>
        <DEFINITION-REF DEST="CHOICE-REFERENCE-PARAM-DEF">
          /AUTOSAR/some_module/PortPin/PortPinMode
        </DEFINITION-REF>
        <VALUE-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /ECUC/mySpi/aSpiExternalDevice1
        </VALUE-REF>
      </REFERENCE-VALUE>
      <REFERENCE-VALUE>
        <DEFINITION-REF DEST="CHOICE-REFERENCE-PARAM-DEF">
          /AUTOSAR/some_module/PortPin/PortPinMode
        </DEFINITION-REF>
        <VALUE-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /ECUC/myAdc/anAdcChannel2
        </VALUE-REF>
      </REFERENCE-VALUE>
    </REFERENCE-VALUES>
  </CONTAINER>
...
```

Example 3.35 depicts the configuration description of definition type `ForeignReferenceParamDef` for example 3.17.

Example 3.35

```
...
  <CONTAINER>
    <SHORT-NAME>myFrameMapping</SHORT-NAME>
    <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
      /AUTOSAR/Rte/Communication/FrameMapping
    </DEFINITION-REF>
    <REFERENCE-VALUES>
      <REFERENCE-VALUE>
        <DEFINITION-REF DEST="FOREIGN-REFERENCE-PARAM-DEF">
          /AUTOSAR/Rte/Communication/FrameMapping/SystemFrame
        </DEFINITION-REF>
        <VALUE-REF DEST="FRAME">
          /SystemDescription/SystemFrameNo42
        </VALUE-REF>
      </REFERENCE-VALUE>
    </REFERENCE-VALUES>
  </CONTAINER>
...
```

3.4.5.1 Instance Reference Values

Due to the formalization of prototypes in the AUTOSAR Templates (see [12]) the reference to the instance of a prototype needs to declare the complete context in which the instance is residing.

[ecuc_sws_3033] The metamodel class `InstanceReferenceValue` provides the mechanism to reference to an actual instance of a prototype. This is achieved by specifying a relation with the stereotype `<<instanceRef>>`.

In figure 3.25 the detailed modeling of the `InstanceReferenceValue` `<<instanceRef>>` is specified.

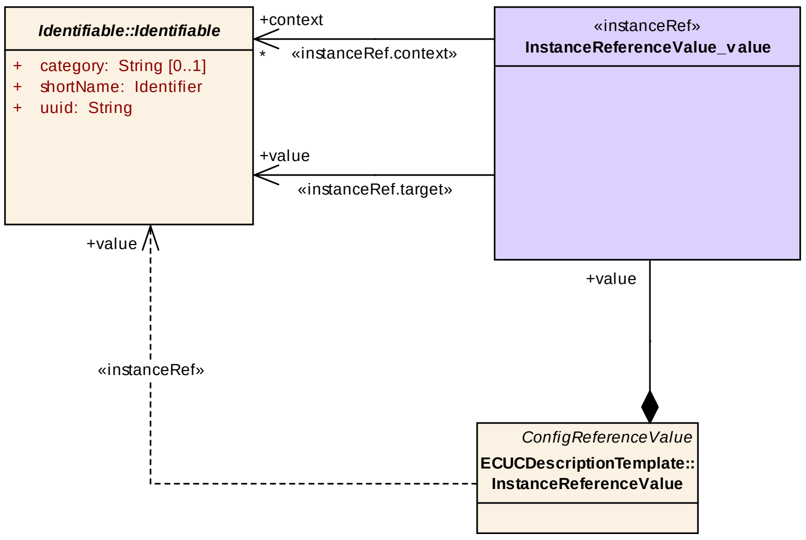

Figure 3.25: Instance Reference Value details

Class	**InstanceReferenceValue**			
Package	M2::AUTOSARTemplates::ECUCDescriptionTemplate			
Class Desc.	InstanceReference representation in the ECU Configuration.			
Base Class(es)	ConfigReferenceValue			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
value	Identifiable	1	instanceRef	InstanceReference representation in the ECU Configuration.

Table 3.40: InstanceReferenceValue

Example 3.36 depicts the configuration description of definition type `InstanceReferenceParamDef` for example 3.18. As one can see in the example the reference value is decomposed of the context path of the instance and the reference to the instance itself.

Example 3.36

```
...
  <CONTAINER>
    <SHORT-NAME>mySenderReceiverMapping</SHORT-NAME>
    <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
      /AUTOSAR/Rte/DataMappings/DataSRMapping
    </DEFINITION-REF>
    <REFERENCE-VALUES>
      <INSTANCE-REFERENCE-VALUE>
        <DEFINITION-REF DEST="INSTANCE-REFERENCE-PARAM-DEF">
          /AUTOSAR/Rte/DataMappings/DataSRMapping/DataElementPrototypeRef
        </DEFINITION-REF>
        <VALUE-IREF>
          <CONTEXT-REF DEST="COMPONENT-PROTOTYPE">
            /DoorFR
          </CONTEXT-REF>
          <CONTEXT-REF DEST="R-PORT-PROTOTYPE">
            /DoorAntennaReceiver
          </CONTEXT-REF>
          <VALUE-REF DEST="DATA-ELEMENT-PROTOTYPE">
            /AntennaStatus
          </VALUE-REF>
        </VALUE-IREF>
      </INSTANCE-REFERENCE-VALUE>
    </REFERENCE-VALUES>
  </CONTAINER>
...
```

3.4.5.2 Representation of Symbolic Names

[ecuc_sws_3036] A `SymbolicNameReferenceParamDef` is represented by an usual `ReferenceValue` in the ECU Configuration Description.

[ecuc_sws_3037] The `shortName` of the referenced `destination` is expected to be the provided symbolic name in the implementation later on. Therefore the code generator of the providing module has the responsibility to associate the provided symbolic name[20] to its actual value.

[ecuc_sws_2107] Configuration parameter values which represent symbolic name values shall be stored in the corresponding XML file at allocation time, regardless whether the values are allocated by the configuration editor or the module generator.

Example 3.37 depicts the configuration description of definition type `SymbolicName-ReferenceParamDef` for example 3.19. To give a better impression how the referencing mechanism and code generation may work the `ModuleConfiguration` of the using and the providing modules are shown here.

Example 3.37

[20]The one that is referenced to

```
...
  <MODULE-CONFIGURATION>
    <SHORT-NAME>myIoHardwareAbstraction</SHORT-NAME>
    <DEFINITION-REF DEST="MODULE-DEF">
      /AUTOSAR/IoHwAb
    </DEFINITION-REF>
    <CONTAINERS>
      <CONTAINER>
        <SHORT-NAME>Dem_PLL_lock_error</SHORT-NAME>
        <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/IoHwAb/IoHwAbDemError
        </DEFINITION-REF>
        <REFERENCE-VALUES>
          <REFERENCE-VALUE>
            <DEFINITION-REF DEST="SYMBOLIC-NAME-REFERENCE-PARAM-DEF">
              /AUTOSAR/IoHwAb/IoHwAbDemError/DemErrorRef
            </DEFINITION-REF>
            <VALUE-REF DEST="PARAM-CONF-CONTAINER-DEF">
              /ECUC/myDem/PLL_Lock_Error
            </VALUE-REF>
          </REFERENCE-VALUE>
        </REFERENCE-VALUES>
      </CONTAINER>
    </CONTAINERS>
  </MODULE-CONFIGURATION>
  <MODULE-CONFIGURATION>
    <SHORT-NAME>myDem</SHORT-NAME>
    <DEFINITION-REF DEST="MODULE-DEF">
      /AUTOSAR/Dem
    </DEFINITION-REF>
    <CONTAINERS>
      <CONTAINER>
        <SHORT-NAME>PLL_Lock_Error</SHORT-NAME>
        <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/Dem/DemError
        </DEFINITION-REF>
        <PARAMETER-VALUES>
          <INTEGER-VALUE>
            <DEFINITION-REF DEST="INTEGER-PARAM-DEF">
              /AUTOSAR/Dem/DemError/DemErrorId
            </DEFINITION-REF>
            <VALUE>17</VALUE>
          </INTEGER-VALUE>
        </PARAMETER-VALUES>
      </CONTAINER>
    </CONTAINERS>
  </MODULE-CONFIGURATION>
...
```

[ecuc_sws_2108] The values of configuration parameters which are defined as `symblicNameValue = true` shall be generated into the header file of the declaring module. The symbol shall be the `shortName` of the container which holds the configuration parameter value prefixed with the module short name `<MSN>` of the declaring BSW Module followed by an underscore.

Taking the specification requirements above the configuration snippet results in the according symbolic name definition in the header file of the providing `Dem` module:

```
...
#define Dem_PLL_Lock_Error 17
...
```

3.4.6 Derived Paramters in an ECU Configuration Description

[ecuc_sws_3021] Providing the configuration value for an instance of a `DerivedParamType` results in an usual `ParameterValue` that was already introduced in the section above.

This results in the following:

- **[ecuc_sws_3022]** A `DerivedBooleanParamDef` will be represented by a `BooleanValue` (see section 3.4.4.1)
- **[ecuc_sws_3023]** A `DerivedIntegerParamDef` will be represented by a `IntegerValue` (see section 3.4.4.2)
- **[ecuc_sws_3024]** A `DerivedFloatParamDef` will be represented by a `FloatValue` (see section 3.4.4.3)
- **[ecuc_sws_3025]** A `DerivedStringParamDef` will be represented by a `StringValue` (see section 3.4.4.4)
- **[ecuc_sws_3026]** A `DerivedEnumerationParamDef` will be represented by a `EnumerationValue` (see section 3.4.4.7)

Example 3.38 depicts the configuration description of derived parameters for example 3.20.

Example 3.38

```
...
  <CONTAINER>
    <SHORT-NAME>myComSignal</SHORT-NAME>
    <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
/AUTOSAR/Com/ComSignal
    </DEFINITION-REF>
    <PARAMETER-VALUES>
      <INTEGER-VALUE>
        <DEFINITION-REF DEST="INTEGER-PARAM-DEF">
/AUTOSAR/Com/ComSignal/ComSigBitPosition
        </DEFINITION-REF>
        <VALUE>4</VALUE>
      </INTEGER-VALUE>
      <INTEGER-VALUE>
        <DEFINITION-REF DEST="INTEGER-PARAM-DEF">
/AUTOSAR/Com/ComSignal/ComSigSizeInBits
        </DEFINITION-REF>
        <VALUE>4</VALUE>
      </INTEGER-VALUE>
      <ENUMERATION-VALUE>
        <DEFINITION-REF DEST="ENUMERATION-PARAM-DEF">
/AUTOSAR/Com/ComSignal/ComSigEndianess
        </DEFINITION-REF>
        <VALUE>BigEndian</VALUE>
      </ENUMERATION-VALUE>
    </PARAMETER-VALUES>
    <REFERENCE-VALUES>
      <REFERENCE-VALUE>
        <DEFINITION-REF DEST="INSTANCE-REFERENCE-PARAM-DEF">
/AUTOSAR/Com/ComSignal/SignalDefinitionSystemDesc
        </DEFINITION-REF>
        <VALUE-REF DEST="SYSTEM-SIGNAL">
/SystemDescription/CommunicationMatrix/SignalInstance27
        </VALUE-REF>
      </REFERENCE-VALUE>
    </REFERENCE-VALUES>
  </CONTAINER>
...
```

3.4.7 Multiple Configuration Sets

As mentioned in section 3.3.3 each `ModuleDef` may specify exactly one `Container` which represents the root container of the multiple configuration set.

[ecuc_sws_3042] The `definition` of this `Container` has to reference the `ParamConfContainerDef` which has the attribute `multipleConfigurationContainer` set to `true`.

[ecuc_sws_3043] The `Container` may occur as often as configuration sets exist. Even if the `upperMultiplicity` of the corresponding `ParamConfContainerDef` is exactly "1". The configuration tools have to check that the multiplicity of this `Container` results from the presence of multiple configuration sets.

[ecuc_sws_3044] The `shortName` of the `Container` has to be the name of the configuration set, i.e. the configuration set is part of the namespace path.

[ecuc_sws_2104] The `shortName` of the `Container`, which is marked as `multipleConfigurationContainer`, shall be a valid C-literal for defining a symbol.

The `shortName` shall be a valid C-literal to be used as a symbol for references within the ECU.

[ecuc_sws_2105] The `shortName` of the `Container`, which is marked as `multipleConfigurationContainer`, shall be unique for the whole ECU. That uniqueness includes all software modules' symbols used on that ECU.

The `shortName` shall be unique for the whole ECU to allow distinct addressing of the configuration data in the ECU memory.

Because the configuration set name is used in the `shortName` of the `Container` each configuration set can be addressed individually. So it is possible to define which configuration set shall be used for a certain initialization of the module.

[ecuc_sws_3045] The parameter description structure underneath the `Container` will be copied for each configuration set, that includes all pre-compile time and link time parameters.

[ecuc_sws_3046] Configuration tools have to check that pre-compile time and link time parameters have the same values throughout all configuration sets.

[ecuc_sws_3047] `ReferenceValue` have to include absolute paths when used in multiple configuration sets. Otherwise it cannot be distinguished which `Identifiable` in which configuration set is referenced.

Example 3.39 depicts a `Container` with according `ParamConfContainerDef` `AdcConfigSet` (see example 3.6) is defined as a multiple configuration set container:

Example 3.39

```
...
  <MODULE-CONFIGURATION>
    <SHORT-NAME>myAdcConfig</SHORT-NAME>
    <DEFINITION-REF DEST="MODULE-DEF">
      /AUTOSAR/Adc
    </DEFINITION-REF>
    <CONTAINERS>
      <CONTAINER>
        <SHORT-NAME>ConfDoorFrontLeft</SHORT-NAME>
        <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/Adc/AdcConfigSet
        </DEFINITION-REF>
        <SUB-CONTAINERS>
          <CONTAINER>
            <SHORT-NAME>hwUnit1</SHORT-NAME>
            <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
              /AUTOSAR/Adc/AdcConfigSet/AdcHwUnit
            </DEFINITION-REF>
            <PARAMETER-VALUES>
              <INTEGER-VALUE>
                <DEFINITION-REF DEST="INTEGER-PARAM-DEF">
                  /AUTOSAR/Adc/AdcConfigSet/AdcHwUnit/AdcHwUnitId
                </DEFINITION-REF>
                <VALUE>5</VALUE>
              </INTEGER-VALUE>
            </PARAMETER-VALUES>
            <SUB-CONTAINERS>
              <!-- ... -->
            </SUB-CONTAINERS>
          </CONTAINER>
        </SUB-CONTAINERS>
      </CONTAINER>
      <CONTAINER>
        <SHORT-NAME>ConfDoorFrontRight</SHORT-NAME>
        <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/Adc/AdcConfigSet
        </DEFINITION-REF>
        <SUB-CONTAINERS>
          <CONTAINER>
            <SHORT-NAME>hwUnit1</SHORT-NAME>
            <DEFINITION-REF DEST="PARAM-CONF-CONTAINER-DEF">
              /AUTOSAR/Adc/AdcConfigSet/AdcHwUnit
            </DEFINITION-REF>
            <PARAMETER-VALUES>
              <INTEGER-VALUE>
                <DEFINITION-REF DEST="INTEGER-PARAM-DEF">
                  /AUTOSAR/Adc/AdcConfigSet/AdcHwUnit/AdcHwUnitId
                </DEFINITION-REF>
                <VALUE>7</VALUE>
              </INTEGER-VALUE>
            </PARAMETER-VALUES>
            <SUB-CONTAINERS>
              <!-- ... -->
            </SUB-CONTAINERS>
          </CONTAINER>
        </SUB-CONTAINERS>
      </CONTAINER>
    </CONTAINERS>
  </MODULE-CONFIGURATION>
...
```

In this example the `Adc` module is used to illustrate the two configuration sets. The `Adc` module is initialized by the `EcuM` module. So the `EcuM` needs to know which configuration set to be used for the initialization of the `Adc`. So in the configuration description of the `EcuM` there needs to be a reference defined to choose the configuration set. For the example 3.39 the references to the two configuration set are:

`<VALUE-REF>/myAdcConfig/ConfDoorFrontLeft</VALUE-REF>` and

`<VALUE-REF>/myAdcConfig/ConfDoorFrontRight</VALUE-REF>`.

With such references any configuration set can be addressed explicitly.

4 ECU Configuration Parameter Definition SWS implications

In this section several aspects of applying the ECU Configuration Specification to AUTOSAR specifications are described.

The ECU Configuration Parameter Definitions are distributed over the BSW SWS documents. How these parameters are specified in the documents is described in section 4.1.

How the AUTOSAR COM-Stack is configured from an inter-module perspective is described in section 4.3.

4.1 Formalization aspects

The goal of this section is to describe how the ECU Configuration Parameter Definitions of BSW modules are specified in the SWS documents. Therefore there is not necessarily a simple translation of the ECU Configuration Parameter's values in the ECU Configuration Description (XML file) into the module's configuration (header file). It is the duty of the module's generation tool to transform the configuration information from the XML file into a header file.

The ECU Configuration Parameter Definitions are formalized in an UML model. This UML model is used to partly generate the specification tables of the BSW SWS and to generate the ECU Configuration Parameter Definition XML file.

Some formalization patterns have been applied when developing the ECU Configuration Parameter Definition:

- *Modified parameter names*: Due to the limitations imposed by the AUTOSAR XML format (32 character limit starting with a letter, etc.) the names of parameters and containers have been redefined. Also a different naming schema has been applied. The original names from the SWS are provided in this document as well.
- *Added parameter multiplicities*: In the original tables from the BSW SWS there is no possibility to specify the optionality and multiplicity of parameters. The parameter multiplicities have been added.
- *Added references*: To allow a better interaction of the configuration descriptions of several modules references between the configuration have been introduced.
- *Harmonized parameter types*:
  - Boolean: Some parameters have been defined as `enumeration` or `#define` where the actual information stored is of type boolean. In those cases they have been modeled as `boolean`.
  - Float: Some parameters store a time value as `integer` where it is stated that this is a time in e.g. micro-seconds. If the time specified is an absolute

time it has been formalized as a `float` in seconds. If the time is a factor of some given time-base the `integer` is preserved.

4.1.1 ECU Configuration Parameter Definition table

The configuration parameters are structured into containers which can hold parameters, references and other containers. Beside the graphical visualization in UML diagrams, tables are used to specify the structure of the parameters.

In the following table one container is specified which holds two parameters and also two additional containers as an example.

SWS Item	[SWS requirement IDs]
Container Name	ContainerName {original name from SWS}
Description	Container description.
Configuration Parameters	

Name	ParameterName {original name from SWS} [SWS requiremenets IDs]		
Description	Parameter description.		
Multiplicity	Parameter multiplicity		
Type	Parameter type		
Configuration Class	**Pre-compile time**	X	Variant1 (Pre-compile Configuration)
	Link time	X	Variant2 (Link-time Configuration), Variant3 (Post-build Configuration)
	Post-build time	–	
Scope / Dependency			

Name	EnumerationTest {ENUMERATION_TEST} [SWS0815]		
Description	description.		
Multiplicity	0..1 (optional)		
Type	EnumerationParamDef		
Range	ReceiveUnqueuedExternal		
	ReceiveUnqueuedInternal		
	SendStaticExternal		
	SendStaticInternal		
Configuration Class	**Pre-compile time**	X	Variant1 (Pre-compile Configuration)
	Link time	X	Variant2 (Link-time Configuration), Variant3 (Post-build Configuration)
	Post-build time	–	
Scope / Dependency			
Included Containers			
Container Name	**Multiplicity**	**Scope / Depedency**	
Container_1	0..1	Optional sub-container.	
Container_2	0..*	Optional sub-container which can be present several times.	

For a detailed description of the elements in the tables please refer to chapter 3.

4.2 AUTOSAR Stack Overview

The software architecture of an AUTOSAR ECU has been divided into several parts to allow independent modules with clean definitions of the interfaces between the different modules. This architecture is depicted in figure 4.1.

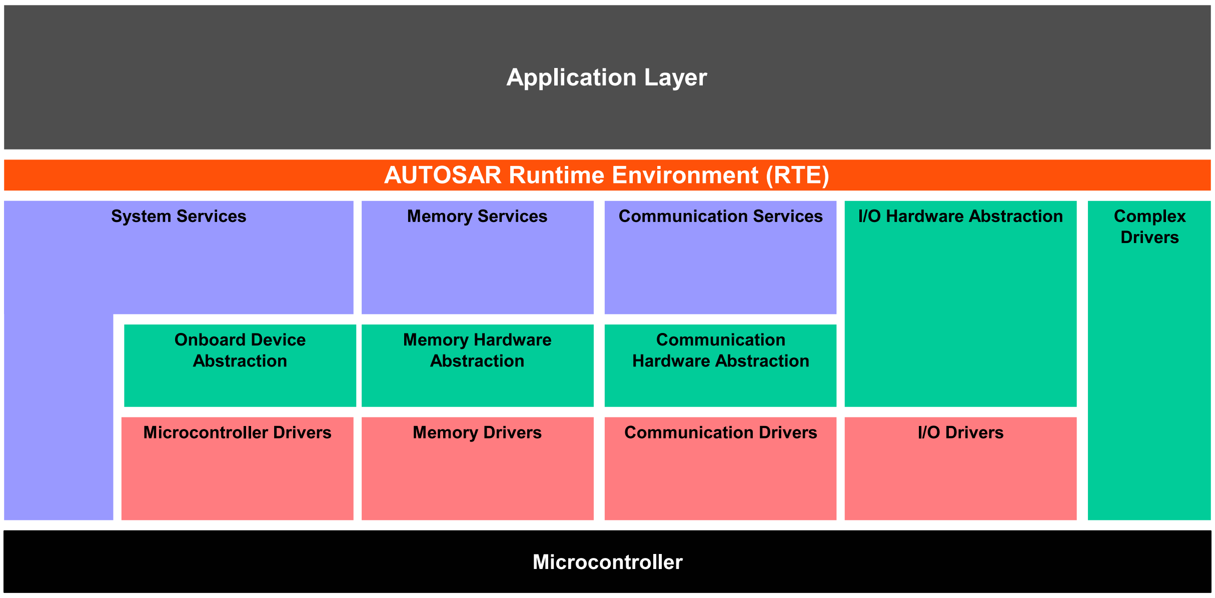

Figure 4.1: ECU Architecture Overview [15]

The Application SW-Components are located at the top and can gain access to the rest of the ECU and also to other ECUs only through the RTE.

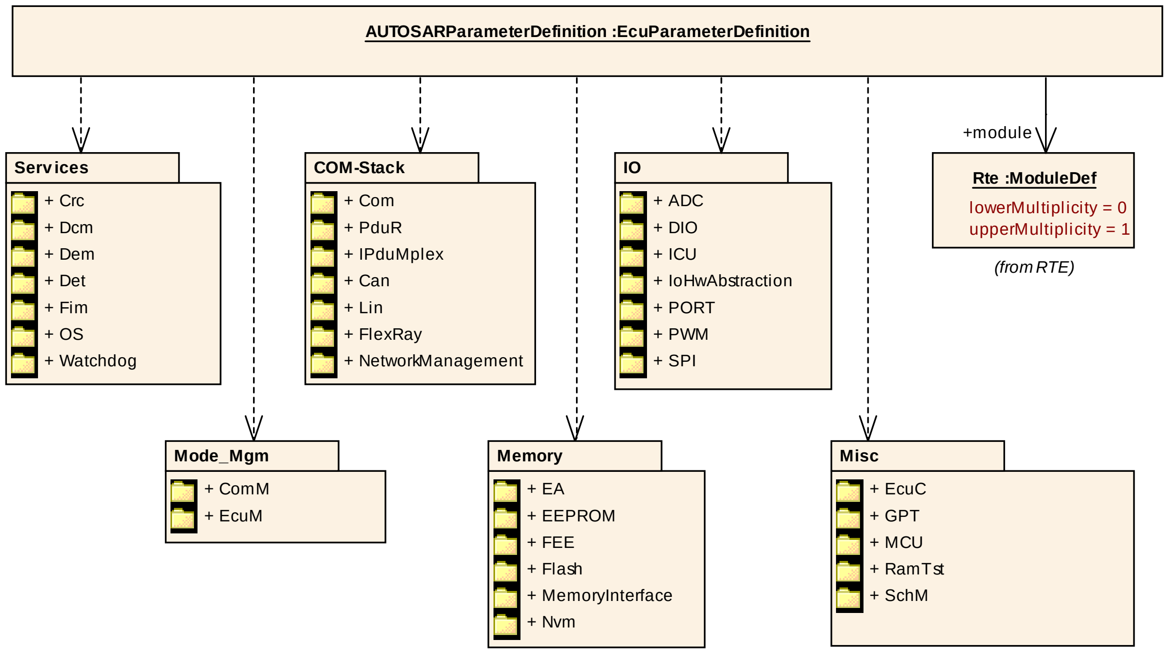

Figure 4.2: AUTOSAR Parameter Definition Overview

The RTE provides the encapsulation of communication and basic services to the Application SW-Components, so it is possible to map the Application SW-Components between different ECUs.

The Basic Software Modules are located below the RTE. The Basic Software itself is divided into the subgroups: System Services, Memory, Communication and IO HW-Abstraction. The Complex Drivers are also located below the RTE.

Among other, the Operating System (OS), the Watchdog manager and the Diagnostic services are located in the System Services subgroup.

The Memory subgroup contains modules to provide access to the non-volatile memories, namely Flash and EEPROM.

In the Communication subgroup the whole AUTOSAR communication stack (COM-Stack) is specified including the COM, Network Management and the communication drivers.

The top-level structure of the `AUTOSARParameterDefinition` is shown in figure 4.2.

The container `AUTOSARParameterDefinition` is the top-level element of the AUTOSAR ECU Configuration Parameter Definition structure. Inside this container references to the diverse configuration container definitions for the different SW modules are defined.

ECU Conf. Name	AUTOSARParameterDefinition	
ECU Conf. Description	Top level container for the definition of AUTOSAR configuration parameters. All of the parameter definitions for the different modules are contained in this container.	
Included Modules		
Module Name	**Multiplicity**	**Scope / Dependency**
Adc	0..1	Configuration of the Adc (Analog Digital Conversion) module.
Can	0..*	This container holds the configuration of a single CAN Driver.
CanIf	0..1	This container includes all necessary configuration sub-containers according the CAN Interface configuration structure.
CanNm	0..1	Configuration Parameters for the Can Nm module.
CanSM	0..1	Configuration of the CanSM module
CanTp	0..1	Configuration of the CanTp (CAN Transport Protocol) module.
CanTrcv	0..*	Configuration of the CanTrcv (CAN Transceiver driver) module.
Com	0..1	COM540: Configuration of the Com module.
ComM	0..1	Configuration of the ComM (Communications Manager) module.
Crc	0..1	Configuration of the Crc (Crc routines) module.
Dcm	0..1	Configuration of the Dcm (Diagnostic Communications Manager) module.
Dem	0..1	Configuration of the Dem (Diagnostic Event Manager) module.

Module Name	Multiplicity	Scope / Dependency
Det	0..1	Configuration of the Det (Development Error Tracer) module. There are NO standardized configuration parameters defined. All the configuration of the Det is done via vendor-specific configuration parameters which need to be defined inside this ModuleDef.
Dio	0..1	Configuration of the Dio (Digital IO) module.
Ea	0..*	Configuration of the Ea (EEPROM Abstraction) module. The module shall abstract from the device specific addressing scheme and segmentation and provide the upper layers with a virtual addressing scheme and segmentation as well as a ”virtually” unlimited number of erase cycles.
EcuC	0..1	Virtual module to collect ECU Configuration specific / global configuration information.
EcuM	0..1	Configuration of the EcuM (ECU State Manager) module.
Eep	0..*	Configuration of the Eep (internal or external EEPROM driver) module. Its multiplicity describes the number of EEPROM drivers present, so there will be one container for each EEPROM driver in the ECUC template. When no EEPROM driver is present then the multiplicity is 0.
Fee	0..*	Configuration of the Fee (Flash EEPROM Emulation) module.
Fim	0..1	Configuration of the Fim (Function Inhibition Manager) module.
Fls	0..*	Configuration of the Fls (internal or external flash driver) module. Its multiplicity describes the number of flash drivers present, so there will be one container for each flash driver in the ECUC template. When no flash driver is present then the multiplicity is 0.
Fr	0..*	Configuration of the Fr (FlexRay driver) module.
FrIf	0..1	Configuration of the FrIf (FlexRay Interface) module.
FrNm	0..1	The Flexray Nm module
FrSm	0..1	
FrTp	0..1	Configuration of the FrTp (FlexRay Transport Protocol) module.
FrTrcv	0..*	Configuration of the FrTrcv (FlexRay Transceiver driver) module.
Gpt	0..1	Configuration of the Gpt (General Purpose Timer) module.
IPduMplex	0..1	Configuration of the IPduMplex (IPdu Multiplexer) module.
Icu	0..*	Configuration of the Icu (Input Capture Unit) module.
IoHwAbstraction	0..1	Configuration of IO HW Abstraction.
Lin	0..*	Configuration of the Lin (LIN driver) module.
LinIf	0..1	Configuration of the LinIf (LIN Interface) module.
LinSM	0..1	Configuration of the Lin State Manager module.
LinTp	0..1	Singleton descriptor for the LIN Transport Protocol.
Mcu	0..1	Configuration of the Mcu (Microcontroler Unit) module.

Module Name	Multiplicity	Scope / Dependency
MemIf	0..1	Configuration of the MemIf (Memory Abstraction Interface) module.
Nm	0..1	The Generic Network Management Interface module
NvM	0..1	Configuration of the NvM (NvRam Manager) module.
Os	0..1	Configuration of the Os (Operating System) module.
PduR	0..1	Configuration of the PduR (PDU Router) module.
Port	0..1	Configuration of the Port module.
Pwm	0..*	Configuration of Pwm (Pulse Width Modulation) module.
RamTst	0..1	Configuration of the RamTst module.
Rte	0..1	Configuration of the Rte (Runtime Environment) module.
SchM	0..1	Configuration of the SchM (BSW Scheduler) module.
Spi	0..1	Configuration of the Spi (Serial Peripheral Interface) module.
Wdg	0..*	Configuration of the Wdg (Watchdog driver) module. This should be the extern watchdog driver. For internal watchdogs, no port pins / DIO is used.
WdgIf	0..1	Configuration of the WdgIf (Watchdog Interface) module.
WdgM	0..1	Configuration of the WdgM (Watchdog Manager) module.

4.3 COM-Stack configuration

To cope with the complexity of the COM-Stack configuration, reoccurring patterns have been applied which will be described in this section. Only the patterns, together with some examples, are shown. To get detailed specification of the configuration for each individual module please refer to the actual BSW SWS documents of these modules.

4.3.1 Handle IDs

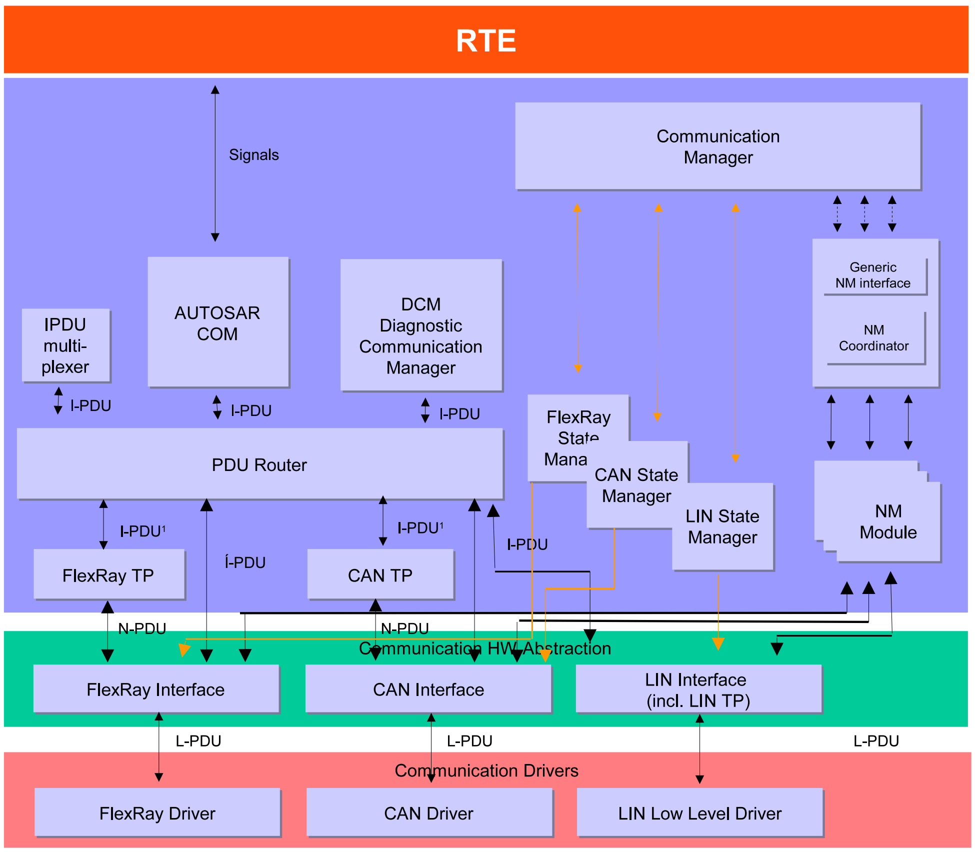

Figure 4.3: Interfaces in the COM-Stack [15]

In figure 4.3 a detailed view of the COM-Stack modules and their interaction is shown. There are several kinds of interactions between adjacent[1] modules.

[1]Modules are called adjacent if they share an interface, so PduR and Com are adjacent, while PduR and Can driver are not.

4.3.1.1 Handle ID concept

The API definitions in the COM-Stack utilize two concepts to achieve the interaction between adjacent modules:

- Pointers to PDU data buffer (the PDU data buffer contains the actual communicated information, depending on the actual layer the interaction happens)
- Handle IDs to identify to what PDU the pointer is referring to.

A typical API call is for instance:

```
PduR_ComTransmit(PduIdType ComTxPduId, PduInfoType *PduInfoPtr)
```

Handle IDs are defined by the module providing the API and used by the module calling the API.

The choice of the value for a Handle ID is open to the implementation of the providing module. There might be different strategies to optimize the Handle ID values and therefore the internal structures of the implementation may have an influence on the choice of the values.

Also the Handle IDs can be chosen freely per module, so a PDU might be sent from Com to the PduR with the `ID=5` and then the PduR transmits it further to the CanIf with `ID=19`. In the configuration information of the PduR it has to be possible to conclude that if a PDU arrives from Com with `ID=5` it has to be forwarded to the CanIf with `ID=19`.

It has to be guaranteed that each Pdu does have a unique handle ID within the scope of the corresponding API. For example: The PduR gets transmission requests from both, the Com and the Dcm modules. But there are also two distinct APIs defined for those requests:

- `PduR_ComTransmit(...)`
- `PduR_DcmTransmit(...)`

Therefore the PduR can distinguish two PDUs, even when they have the same handle ID but are requested via different APIs.

Another use-case in the COM-Stack only provides one API for all the callers: the interface layer (CanIf, FrIf, LinIf).

- `CanIf_Transmit(...)`

Here it has to be guaranteed that each transmit request for a distinct PDU does have a unique handle ID.

The actual values of the handle IDs can only be allocated properly when the configuration of one module is completed, since only then the internal data structures can be defined.

In the next sections the patterns used to define and utilize Handle IDs are described.

4.3.1.2 Definition of Handle IDs

The ECU Configuration Description (which holds the actual values of configuration parameters) is structured according to the individual BSW Module instances. Therefore the ECU Configuration Parameter Definition is also structured in this way.

In figure 4.4 an exemplary definition of a partial Can Interface transmit configuration is shown.

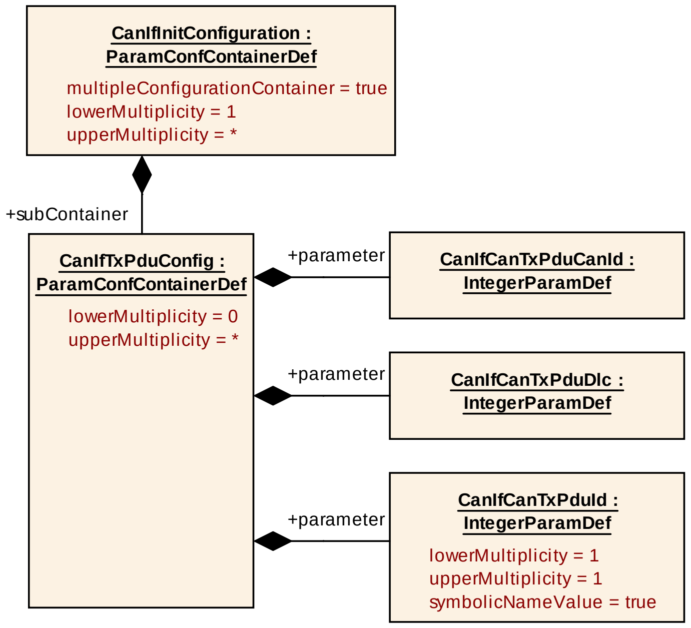
CanIfInitConfiguration : ParamConfContainerDef
multipleConfigurationContainer = true
lowerMultiplicity = 1
upperMultiplicity = *
+subContainer
CanIfTxPduConfig : ParamConfContainerDef
lowerMultiplicity = 0
upperMultiplicity = *
+parameter
CanIfCanTxPduCanId : IntegerParamDef
+parameter
CanIfCanTxPduDlc : IntegerParamDef
+parameter
CanIfCanTxPduId : IntegerParamDef
lowerMultiplicity = 1
upperMultiplicity = 1
symbolicNameValue = true

Figure 4.4: Example of Can Interface Tx configuration

The configuration of the module `CanIf` may contain several `CanIfTxPduConfig` objects.

Each `CanIfTxPduConfig` object contains information on one Pdu which is coming from an upper layer (e.g. PduR or Nm) and is going to some Can driver. In this example the `CanIfCanTxPduCanId` and `CanIfCanTxPduDlc` are specified for each to be transmitted Pdu. There is a similar structure needed for the receive use-case as well.

Additionally the parameter `CanIfCanTxPduId` is specified. This integer parameter will later hold the actual value for the handle ID. So the handle ID value is stored inside the structure of the defining module.

Since the handle ID `CanIfCanTxPduId` is part of the container `CanIfTxPduConfig` the semantics of the symbolic names can be applied.

[ecuc_sws_2106] If a configuration parameter holds a handle Id which needs to be shared between several modules it shall have the `symbolicNameValue = true` set.

Thus it is required that all handle Id values are accessible via a symbolic name reference (see section 4.3.1.5).

4.3.1.3 Definition of PDUs

With the possibility to define Handle IDs a module is now able to publish the Handle ID values to other modules. In this section it shall be described how other modules can relate to those Handle IDs and how to define the flow of a PDU[2] through the COM-Stack.

To be able to define a PDU ”flowing” through the COM-Stack two modules need to be able to refer to the same PDU object. Therefore a generic `Pdu` container has been defined which does not belong to any module but to the whole COM-Stack.

Since the PDU flowing through the COM-Stack does not belong to an individual module, the ”virtual” module `EcuC` has been introduced in the ECU Configuration. This module is used to collect configuration information not associated with any specific standardized module.

The `PduCollection` may contain several ”global” `Pdu` objects as shown in figure 4.5. Each `Pdu` may be representing an actual `PduToFrameMapping`[3] from the AUTOSAR System Description[10] (the ECU Extract), therefore there is an optional reference to an element in the System Template. The reference is optional because the `PDU`s which are transported within an ECU only are not necessarily part of the ECU Extract. Especially PDUs handled by the Transport Protocol modules have no representation in the ECU Extract (There is a PDU coming over the bus which is represented by a `Pdu` object, but when the TP does the conversion and a new `Pdu` is created which then is forwarded to the upper layer. This created `Pdu` does not have a reference to a `PduToFrameMapping`).

[2]For this aspect of the configuration it does not matter what kind of PDU it is, i.e. I-PDU, L-PDU or N-PDU.

[3]The element `PduToFrameMapping` represents the actual PDU in the specific ECU (formerly known as `PduInstance`)

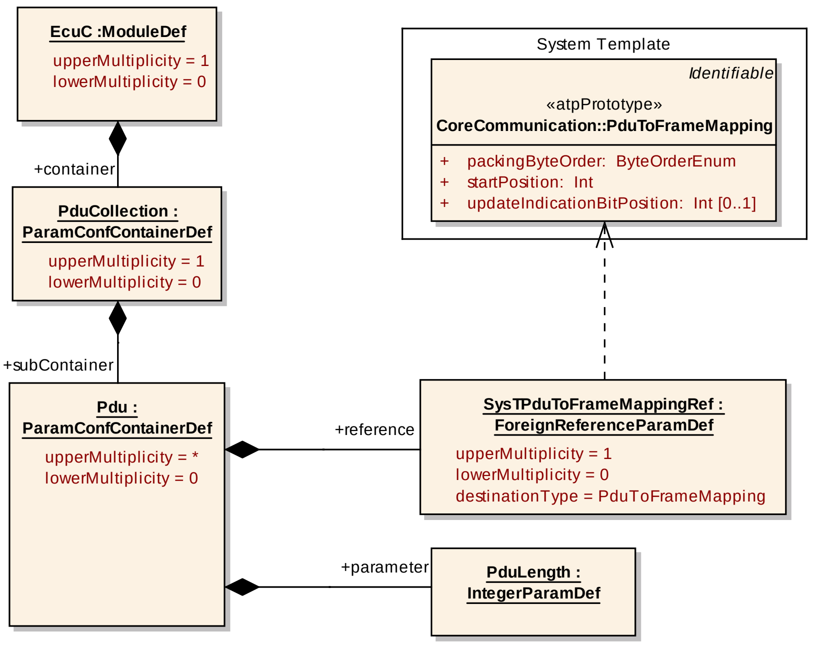
EcuC :ModuleDef
upperMultiplicity = 1
lowerMultiplicity = 0
+container
PduCollection : ParamConfContainerDef
upperMultiplicity = 1
lowerMultiplicity = 0
+subContainer
Pdu : ParamConfContainerDef
upperMultiplicity = *
lowerMultiplicity = 0
System Template
Identifiable
«atpPrototype»
CoreCommunication::PduToFrameMapping
+ packingByteOrder: ByteOrderEnum
+ startPosition: Int
+ updateIndicationBitPosition: Int [0..1]
+reference
SysTPduToFrameMappingRef : ForeignReferenceParamDef
upperMultiplicity = 1
lowerMultiplicity = 0
destinationType = PduToFrameMapping
+parameter
PduLength : IntegerParamDef

Figure 4.5: Generic Pdu Container

Pdu

SWS Item		
Container Name	Pdu	
Description	One Pdu flowing through the COM-Stack. This Pdu is used by all Com-Stack modules to aggree on referencing the same Pdu.	
Configuration Parameters		

Name	PduLength		
Description	Length in bits of the Pdu.		
Multiplicity	1		
Type	IntegerParamDef		
Default Value			
Configuration Class	**Pre-compile time**	–	
	Link time	–	
	Post-build time	–	
Scope / Dependency			

Name	SystemTemplatePduToFrameMappingRef		
Description	Optional reference to the PduToFrameMapping from the SystemTemplate which this Pdu represents.		
Multiplicity	0..1		
Type	Foreign reference to PduToFrameMapping		
Configuration Class	**Pre-compile time**	–	
	Link time	–	
	Post-build time	–	
Scope / Dependency			
No Included Containers			

4.3.1.4 Agreement on Handle IDs

During the configuration of a module, information for each Pdu flowing through this module is created (see again figure 4.4: CanIfTxPduConfig) which hold module-specific configuration information. Now each of these ”local” Pdu configurations needs to be related to a ”global” Pdu element representing information flowing through the COM-Stack. This is done by introducing a ReferenceParamDef from the ”local” Pdu to the ”global” Pdu.

In figure 4.6 this relationship is shown for the PduRDestPdu and the CanIfTxPduConfig.

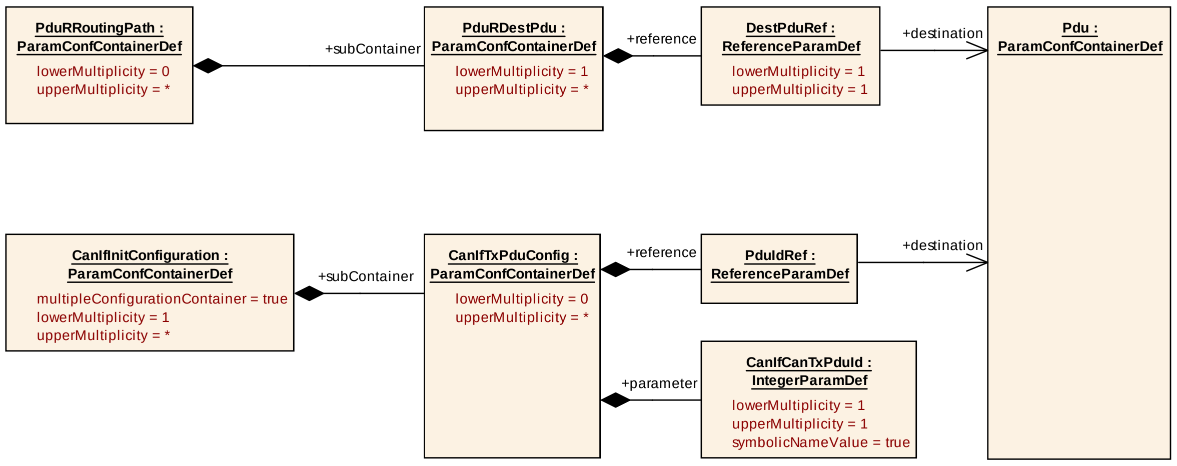

Figure 4.6: Transmission from PduR to CanIf

There are two reasons why the ”global” PDU has been introduced and why all ”local” PDUs have to point to the ”global” PDU only.

- When doing the configuration of module PduR only the ”global” PDU needs to be present, there is no need for the ”local” PDU in the CanIf to be present yet.
- The References are stored in the ”local” PDU structure, so changes applied do only influence the structure of the changed module.

Taking the structure shown in figure 4.6 it is now possible to generate both modules.

The CanIf (automatic) configuration editor collects all ”local” CanIfTxPduConfigs and generates/stores the values for their handle ID in CanIfCanTxPduId. If the CanIf needs to know where the Pdu transmit request is coming from it can follow the PduIdRef to the ”global” Pdu and then ”query” all references pointing to that Pdu. By following those references in reversed direction the transmitting module can be found.

The PduR generator has to know which handle ID to use for each Pdu that has to be sent to the CanIf. To get the actual handle ID value the mechanism is the same in the CanIf use-case: follow the ”global” Pdu reference and ”query” the modules pointing to that ”global” Pdu. Then find the module(s) type this Pdu is going to be transmitted to. In case of a multicast there might be several modules to send the same PDU to.

With this approach a high degree of decoupling has been achieved between the configuration information of the involved modules. Even when modules are adjacent and need to share information like handle ID, the references between the modules are always indirect using the ”global” `Pdu` elements.

4.3.1.5 Handle IDs with symbolic names

The usage of handle Ids together with symbolic names is targeting several use-cases for the methodology of configuring adjacent modules. For the definition of possible configuration approaches please refer to section A.1.1.

For the discussion of the Handle Id use-cases two basic approaches can be distinguished when dividing the methodology into the steps configuration editing and module generation:

- Handle Ids allocated by the configuration editor
- Handle Ids allocated by the module generator

It is assumed that the configuration and generation of the whole stack is done using different tools (possibly from different vendors) which might have either of the two approaches implemented.

In requirement [ecuc_sws_2106] it is required that all handle Ids are represented as `symbolicNameValue = true` configuration parameters thus decoupling the value from its usage.

In requirement [ecuc_sws_2107] it is required that the allocated values are stored in the XML (latest after module generation) so the allocated values are documented. In case the allocation of values has to be performed at a later point in time again (with updated input information) the non affected values can be preserved. It is also needed to support debugging.

In requirement [ecuc_sws_2108] it is required that the handle Id values are always generated into the module’s header file. With this approach it is possible to freely choose the configuration approach of the adjacent modules.

This approach has significant effect on the methodology due to the circular dependencies between the adjacent modules(Com sends to the PduR using PduR handle Ids, PduR indicates to Com using Com handle Ids). Therefore the configuration of all adjacent modules has to be re-visited in case some handle Id changes happen. This contributes to the approach that FIRST the *configuration* of the stack is performed and SECOND the *generation* is triggered.

An example of this approach is provided below: By adding the attribute `symbolicNameValue=true` to the parameter holding the handle ID (in figure 4.6 this is the parameter `CanIfTxPduId`) the code generator doing the `CanIf` will generate a `#define` in the `CanIf_cfg.h` file. The name of the define is the name of the parent container (each container does have a short name) and the value is the actual number assigned to that handle ID.

For example in `CanIf_cfg.h`:

```
#define CanIfTx_Pdu_2345634_985_symbol   17
```

The benefit is that the generator of the `PduR` does not need to wait for the `CanIf` to be configured completely and handle IDs are generated. If the `CanIf` publishes the symbolic names for the handle IDs, the `PduR` can expect those symbolic names and generate the `PduR` code using those symbolic names.

For example in `PduR.c`:

```
CanIf_Transmit( CanIfTx_Pdu_2345634_985_symbol, PduPtr )
```

Therefore the `PduR` can be generated as soon as its own configuration is finished and there is no need to wait for the `CanIf` to be finished completely. However, at least the ”local” `Pdu` in the `CanIf` has to be already created to allow this, because the name of the symbol has to be fetched from this configuration.

Of course the `PduR` can only be compiled after the `CanIf` has been generated as well, but with the utilization of the symbolic names together with handle IDs an even higher degree of decoupling in the configuration process is achieved.

4.3.2 Configuration examples for the Pdu Router

In this section several use-cases of the PduR are described from the configuration point of view. The focus is on the interaction of the PduR configuration with the configuration of the other COM-Stack modules. Therefore only some configuration parameters are actually shown in these examples.

4.3.2.1 Tx from Com to CanIf

In the example in figure 4.7 a Pdu is sent from the Com module – via the Pdu Router – to the Can Interface. Since this one Pdu is handed over through these layers there is only need for one global Pdu object `System_Pdu`.

The Com module's configuration points to the `System_Pdu` to indicate which Pdu shall be sent. The actual Handle Id which has to be used in the API call will however be defined by the PduR in the parameter `PduRSrcPdu::HandleId`. In this example the Com module has to use the Hanlde Id `23` to transmit this Pdu to the PduR.

Then, since the CanIf is pointing to the same `System_Pdu` the PduR can be configured to send this Pdu to the CanIf. The Handle Id is defined in the CanIf configuration in the parameter value of `CanIfCanTxPduId`.

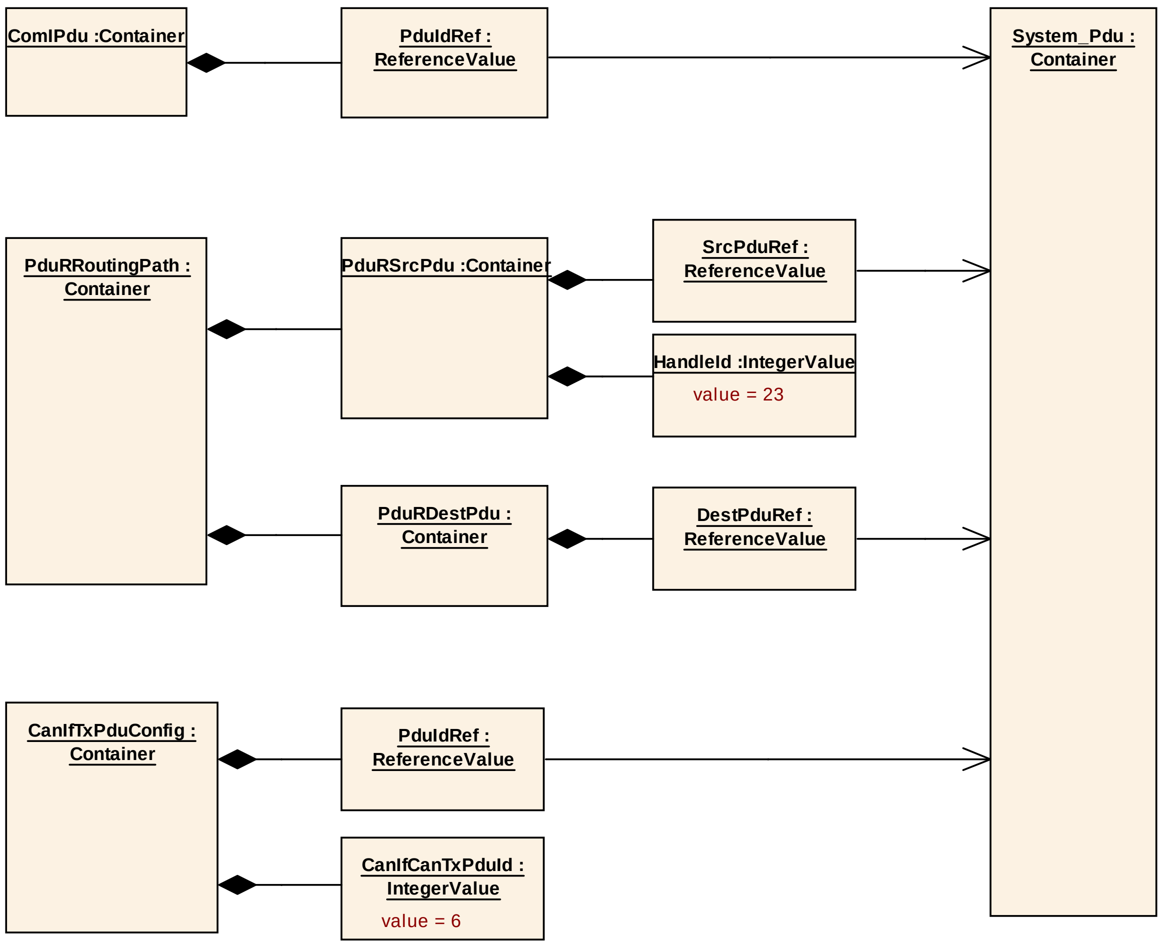
ComIPdu :Container
PduIdRef : ReferenceValue
System_Pdu : Container
PduRRoutingPath : Container
PduRSrcPdu :Container
SrcPduRef : ReferenceValue
HandleId :IntegerValue
value = 23
PduRDestPdu : Container
DestPduRef : ReferenceValue
CanIfTxPduConfig : Container
PduIdRef : ReferenceValue
CanIfCanTxPduId : IntegerValue
value = 6

Figure 4.7: Tx from Com to CanIf example

4.3.2.2 Rx from CanIf to Com

In the example in figure 4.8 the reception use-case from the CanIf to the Com module is configured. Here the Handle Ids are defined in the PduR and the Com module's configuration.

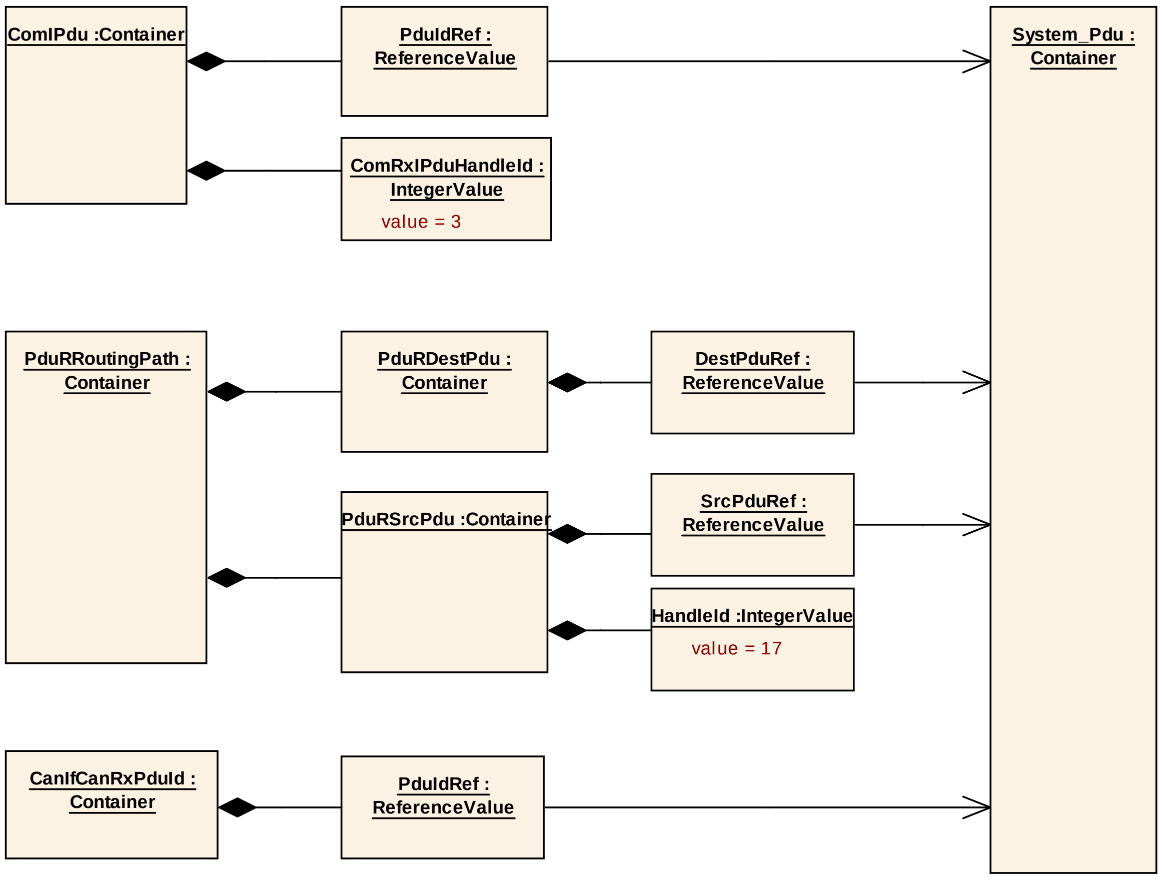
ComIPdu :Container
PduIdRef : ReferenceValue
ComRxIPduHandleId : IntegerValue
value = 3
System_Pdu : Container
PduRRoutingPath : Container
PduRDestPdu : Container
DestPduRef : ReferenceValue
PduRSrcPdu :Container
SrcPduRef : ReferenceValue
HandleId :IntegerValue
value = 17
CanIfCanRxPduId : Container
PduIdRef : ReferenceValue

Figure 4.8: Rx from CanIf to Com example

4.3.2.3 Gateway from CanIf to FrIf

In the example in figure 4.9 the gateway use-case is shown. Since there are two Pdus involved there are two `System_Pdu` objects defined: one which is representing the Can Pdu and one which represents the Fr Pdu. Via the references to these two `System_Pdu` objects the gateway is configured.

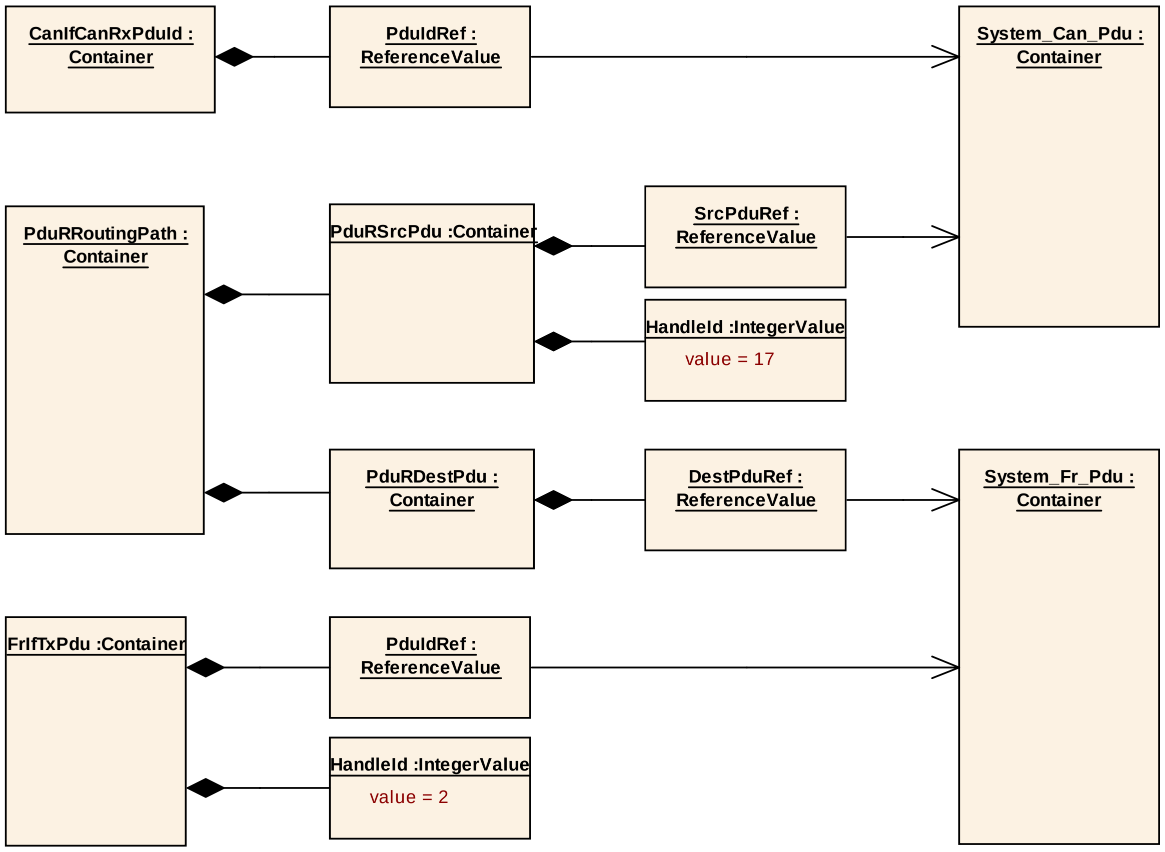
CanIfCanRxPduId : Container
PduIdRef : ReferenceValue
System_Can_Pdu : Container
PduRRoutingPath : Container
PduRSrcPdu :Container
SrcPduRef : ReferenceValue
HandleId :IntegerValue
value = 17
PduRDestPdu : Container
DestPduRef : ReferenceValue
System_Fr_Pdu : Container
FrIfTxPdu :Container
PduIdRef : ReferenceValue
HandleId :IntegerValue
value = 2

Figure 4.9: Gateway from CanIf to FrIf example

4.4 Calculation of ticks

Typically the time related parameters in AUTOSAR are given as float values. Nevertheless for some parameters the unit [ticks] is required. The advantage of having ticks in the ECU configuration is that the final value is already known before the code generator is called. Otherwise it depends on the implementer of the code generator what final value is calculated.

To avoid this situation, in both cases (manually or automatically calculated) the same rules shall be applicable that resolve float values into tick units.

These rules shall be used

- manually by the user prior to the call of the code generator (if ticks are specified and expected)
- automatically (implicitly) by the code generators (if floats are specified and expected)

[ecuc_sws_7000] Calculation formula for min values:
IF (((Required Min Time) MOD (Main Function Period Time)) != 0)
THEN Number of Ticks = INT (Required Min Time / Main Function Period Time) +1
ELSE Number of Ticks = INT (Required Min Time / Main Function Period Time)

[ecuc_sws_7001] Calculation formula for max or other values:
Number of Ticks = INT (Required Time / Main Function Period Time)

[ecuc_sws_7002] Restrictions in case of generator usage:

- An error shall be generated if the calculated number of ticks is less than 1 (except for min values) since they are anyway above its limit.
- A warning shall be generated if the calculated number of ticks is not dividable without rest.

Remark: Due to non specified implementation constraints its up to the vendor to define additional checks if necessary.

Example 1 (Main Period = 1.5 ms = 0.0015 sec):

Parameter	Val [ms]	Val [sec]	Calc Val [ticks]	Calc Val [ms]	Message
Avg Time	10	0.01	6.667	9	warning
Min Time	6	0.006	4.0	6	none

Example 2 (Main Period = 10 ms = 0.01 sec):

Parameter	Val [ms]	Val [sec]	Calc Val [ticks]	Calc Val [ms]	Message
Max Time	8	0.008	0.8	0	error
Min Time	5	0.005	0.5	1	warning

5 Rules to follow in different configuration activities

This chapter defines rules relevant for the relation between standardized module definitions and vendor specific module definitions, rules for building the base `ECU configuration Description` and rules for configuration editors.

5.1 Deriving vendor specific module definitions from standardized module definitions

The following rules must be followed when generating the `Vendor Specific Module Definition` (abbreviated with VSMD in this chapter) from the `Standardized Module Definition` (abbreviated StMD in this chapter). The basic relationship between these two kinds of parameter definitions are depicted in figure 5.1.

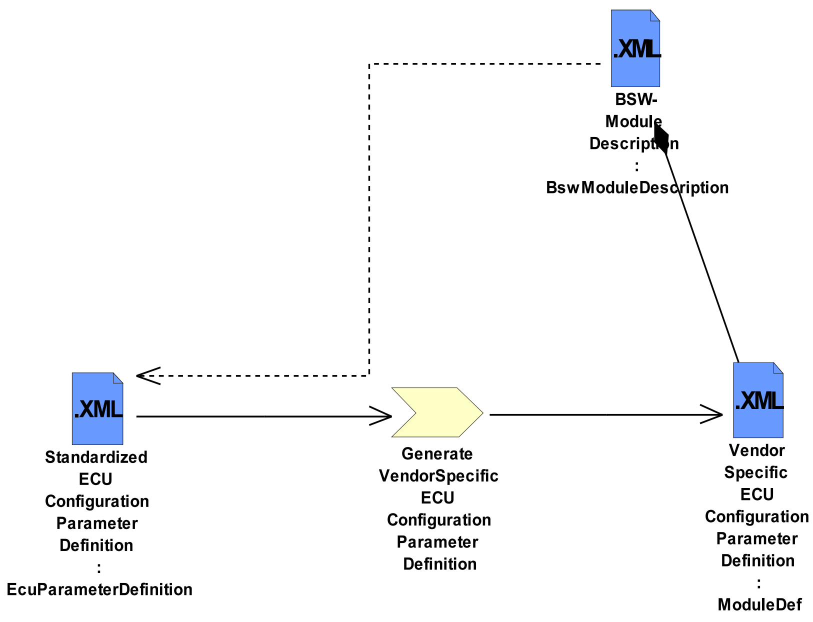
.XML
BSW-Module Description : BswModuleDescription
.XML
Standardized ECU Configuration Parameter Definition : EcuParameterDefinition
Generate VendorSpecific ECU Configuration Parameter Definition
.XML
Vendor Specific ECU Configuration Parameter Definition : ModuleDef

Figure 5.1: Generating Vendor Specific Module Definitions (per module)

[ecuc_sws_1000] These rules shall be checked by tools that validate that a SW module implementation conforms to its AUTOSAR specification.

- **[ecuc_sws_1001]** The `lowerMultiplicity` of the module in the VSMD must be 1 or bigger to what is defined in the StMD. The `upperMultiplicity` of that module may be equal or less to what is defined in the StMD.
- **[ecuc_sws_6001]** The `shortName` of a VSMD module shall be the same as the `shortName` of the StMD.
- **[ecuc_sws_6003]** The package structure of the VSMD has to be different than "/AUTOSAR/" so that it is possible to distinguish the standardized from the vendor specific module definitions. Example 5.1 shows the difference between the VSMD and StMD. The package structure of the vendor specific CanIf module definition begins with "/VendorX/CanIf" and the package structure of the vendor specific CanDrv module definition begins with "/VendorY/Can".

Example 5.1

CanIf and CanDrv AUTOSAR standardized XML:

```
<TOP-LEVEL-PACKAGES>
   <AR-PACKAGE>
   <SHORT-NAME>AUTOSAR</SHORT-NAME>
      <ELEMENTS>
      <MODULE-DEF>
         <SHORT-NAME>CanIf</SHORT-NAME>
         <CONTAINERS>
            <PARAM-CONF-CONTAINER-DEF>
               <SHORT-NAME>CanIfDriverConfig</SHORT-NAME>
               <REFERENCES>
                  <!--Reference Definition:CanIfDriverRef-->
                  <REFERENCE-DEF>
                     <SHORT-NAME>CanIfDriverRef</SHORT-NAME>
               <DESTINATION-REF DEST="PARAM-CONF-CONTAINER-DEF">
                  /AUTOSAR/Can/CanGeneral
               </DESTINATION-REF>
                  </REFERENCE-DEF>
               </REFERENCES>
            </PARAM-CONF-CONTAINER-DEF>
         </CONTAINERS>
      </MODULE-DEF>
      <MODULE-DEF>
         <SHORT-NAME>Can</SHORT-NAME>
            <CONTAINERS>
               <PARAM-CONF-CONTAINER-DEF>
                  <SHORT-NAME>CanGeneral</SHORT-NAME>
                  <PARAMETERS>
                     ...
                  </PARAMETERS>
               </PARAM-CONF-CONTAINER-DEF>
            </CONTAINERS>
      </MODULE-DEF>
   ...
```

CanIf VendorX XML:

```
<TOP-LEVEL-PACKAGES>
   <AR-PACKAGE>
      <SHORT-NAME>VendorX</SHORT-NAME>
      <ELEMENTS>
         <MODULE-DEF>
            <SHORT-NAME>CanIf</SHORT-NAME>
            <REFINED-MODULE-DEF>AUTOSAR/CanIf</REFINED-MODULE-DEF>
            <CONTAINERS>
               <PARAM-CONF-CONTAINER-DEF>
                  <SHORT-NAME>CanIfDriverConfig</SHORT-NAME>
                  <REFERENCES>
                     <!--Reference Definition:CanIfDriverRef-->
                     <REFERENCE-DEF>
                        <SHORT-NAME>CanIfDriverRef</SHORT-NAME>
                  <DESTINATION-REF DEST="PARAM-CONF-CONTAINER-DEF">
                     /AUTOSAR/Can/CanGeneral
                  </DESTINATION-REF>
                     </REFERENCE-DEF>
                  </REFERENCES>
               </PARAM-CONF-CONTAINER-DEF>
            </CONTAINERS>
         </MODULE-DEF>
  ...
```

The DESTINATION-REF content shall not be changed from "/AUTOSAR/..." in the VSMD.

CanIf VendorY XML:

```
<TOP-LEVEL-PACKAGES>
   <AR-PACKAGE>
      <SHORT-NAME>VendorY</SHORT-NAME>
      <ELEMENTS>
         <MODULE-DEF>
            <SHORT-NAME>Can</SHORT-NAME>
            <REFINED-MODULE-DEF>AUTOSAR/Can</REFINED-MODULE-DEF>
            <CONTAINERS>
               <PARAM-CONF-CONTAINER-DEF>
                  <SHORT-NAME>CanGeneral</SHORT-NAME>
                  <PARAMETERS>
                     ...
                  </PARAMETERS>
               </PARAM-CONF-CONTAINER-DEF>
            </CONTAINERS>
         </MODULE-DEF>
      ...
```

For all `ContainerDefs` and `ParameterTypes` and `ConfigReferences` defined within the `ModuleDef` in the StMD, it holds:

- **[ecuc_sws_1002]** Optional elements (with `lowerMultiplicity` $= 0$) in the StMD may be omitted from the VSMD. This means that implementations may or may not support parameters and containers defined as optional in the StMD.

- **[ecuc_sws_1003]** Elements defined with `lowerMultiplicity` > 0 in StMD must be present in the VSMD and must not be omitted. `lowerMultiplicity` in the VSMD must be bigger or equal and `upperMultiplicity` must be equal or less than in the StMD.
- **[ecuc_sws_1034]** Elements taken over from the StMD to the VSMD must use exactly the same `shortName`, since the short name identifies the element. This holds for container definitions and individual parameters. The short names of optional elements that are omitted from the VSMD must not be used for vendor specific parameters.
- **[ecuc_sws_1035]** Elements taken over from the StMD to the VSMD must have unique `uuid` in each description. Thus a new `uuid` must be generated when taking over an element.
- **[ecuc_sws_1005]** The `origin` attribute must not be changed for any parameter taken over from the StMD, even when attributes of the parameter are modified in the VSMD.
- **[ecuc_sws_1006]** The `defaultValue` attributed may be changed (or added, if missing).
- **[ecuc_sws_1007]** The `min` values specified in the VSMD must be bigger or equal, the `max` value must be less or equal than the corresponding value specified in the StMD.
- **[ecuc_sws_1008]** Parameter definitions in the StMD may be changed to derived parameter definitions of the same type. If the implementation has some means to derive a parameter value from other parameters or other templates, it is allowed to add the rules that shall be applied to derive that value.

Parameter defined in the StMD	May become in the VSMD
`BooleanParamDef`	`BooleanParamDef` or `DerivedBooleanParamDef`
`IntegerParamDef`	`IntegerParamDef` or `DerivedIntegerParamDef`
`FloatParamDef`	`FloatParamDef` or `DerivedFloatParamDef`
`StringParamDef`	`StringParamDef` or `DerivedStringParamDef`
`EnumerationParamDef`	`EnumerationParamDef` or `DerivedEnumerationParamDef`

- **[ecuc_sws_1009]** For derived parameters defined in the StMD, the values of the `calculationFormula` and `calculationLanguage` may change in the VSMD.
- **[ecuc_sws_5004]** The purpose of `calculationFormula` is to derive a value/result for a given object from its dependent object. So reference is a must and should be used to represent the location of the dependent object.
- **[ecuc_sws_5005]** The `calculationFormula` should not contain any characters other than alphabets, digits, space and following special characters.

”	’	=	{	}	(	)	.	+	-	*	/

- **[ecuc_sws_5006]** The conditional operators in the `calculationFormula` will be written as EQ, LT and so on. Given below are different operators and there meaning

Conditional Operators	Meaning
EQ	Equal To
NE	Not Equal To
LT	Lesser Than
GT	Greater Than To
LE	Lesser Than or Equal To
GE	Greater Than or Equal To

- **[ecuc_sws_1010]** A `ChoiceContainerDef` may include less aggregated `choices` in the VSMD than in the StMD. At least one aggregated `choice` must be included in the VSMD if the `ChoiceContainerDef` is included, however.
- **[ecuc_sws_1011]** Additional vendor specific `choices` (i.e. aggregated `ParamConfContainerDefs`) may be added for `ChoiceReferenceParamDef` in the VSMD.
- **[ecuc_sws_1012]** A `ChoiceReferenceParamDef` may include less references in the VSMD than in the StMD. At least one aggregated `reference` must be included in the VSMD if the `ChoiceReferenceParamDef` is included, however.
- **[ecuc_sws_1013]** Additional vendor specific references may be added for `ChoiceReferenceParamDef` in the VSMD.
- **[ecuc_sws_1014]** Additional vendor specific parameter definitions (using `ParameterTypes`), container definitions and references may be added anywhere within the `ModuleDef` in the VSMD.
- **[ecuc_sws_1015]** The `origin` attribute of vendor specific additional elements shall contain the name of the vendor that defines the element.
- **[ecuc_sws_2084]** For an `EnumerationParamDef` from the StMD there can be additional `EnumerationLiteralDef` added in the VSMD.
- **[ecuc_sws_6002]** For an `EnumerationParamDef` from the StMD there can be `EnumerationLiteralDefs` that are removed in the VSMD.
- **[ecuc_sws_5001]** Avoid redundant entries of module definition. If the module definition is present in the StMD, the same should not be present in VSMD. If parameters in VSMD can be derived from the StMD directly or by computation, these parameters need not be defined in the VSMD.
- **[ecuc_sws_5002]**Induce VSMD in to the StMD in a simplified manner, so that the configuration can be carried out without any disarray.
- **[ecuc_sws_5003]** The `desc` in VSMD can be used to specify detailed information about the respective parameter.

Figure 5.2 shows an overview about rules, which shall be checked by tools that validate that a SW module implementation conforms to its AUTOSAR specification. In this example three parameters are defined in the StMD.

The multiplicity in each of this parameter definitions specifies how often a parameter is allowed to occur in the ECU Configuration Description. In the VSMD optional elements (with `lowerMultiplicity` = 0) may be omitted, as it happens with parameter A (1). Parameters defined with `lowerMultiplicity` $>$ 0 in StMD must be present in the VSMD (2). The `upperMultiplicity` of parameters in the VSMD must be equal or less than in the StMD (2, 3). New vendor specific parameters may also be added in the VSMD (4).

The VSMD defines which parameters are available in which container and what kind of restrictions are to be respected.

If the `upperMultiplicity` of a parameter definition in the VSMD is 0, the parameter shall not exist in the parameter description(5). If the `lowerMultiplicity` of a parameter definition is bigger than 0, the parameter must exist (6). Missing parameters shall be detected by tools (8). Parameters without parameter definitions shall also be detected (9). The number of parameters in the ECUC Description shall not exceed the `upperMultiplicity` of the parameter definition in the VSMD (7).

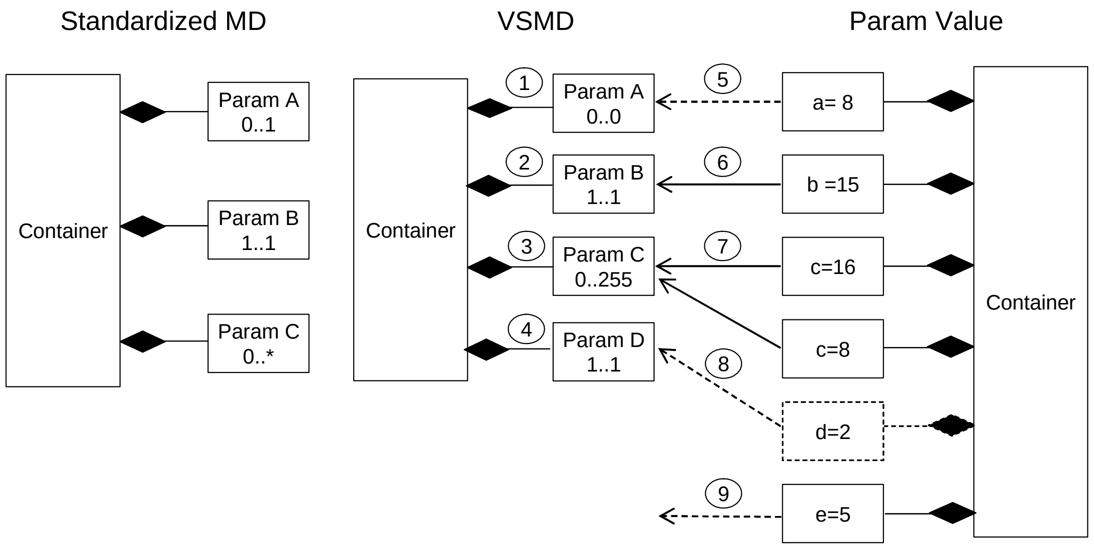

Figure 5.2: Relation between standardized module definition and vendor specific module definition

Example 5.2 depicts the usage of VSMD in case of parameter definition.

Example 5.2

```
...
    <INTEGER-PARAM-DEF>
      <SHORT-NAME>ClockRate</SHORT-NAME>
      <ORIGIN>AUTOSAR_ECUC</ORIGIN>
    </INTEGER-PARAM-DEF>
    <BOOLEAN-PARAM-DEF>
      <SHORT-NAME>VendorExtensionEnabled</SHORT-NAME>
      <ORIGIN>VendorXYZ_v1.3</ORIGIN>
    </BOOLEAN-PARAM-DEF>
...
```

Example 5.3 depicts the usage of VSMD in case of parameter description.

Example 5.3

```
...
    <INTEGER-PARAM-DEF>
      <DEFINITION-REF>
        /VendorXYZ/Mcu/McuGeneral/ClockRate
      </DEFINITION-REF>
      <VALUE>123</VALUE>
    </INTEGER-PARAM-DEF>
    <BOOLEAN-PARAM-DEF>
     <DEFINITION-REF>
        /VendorXYZ/Mcu/McuGeneral/VendorExtensionEnabled
      </DEFINITION-REF>
      <VALUE>true</VALUE>
    </BOOLEAN-PARAM-DEF>
...
```

5.2 Rules for building the Base ECU configuration

Chapter 2.3.3 defines the activity how to generate the base ECU configuration description. The following rules apply during generation of the base ECU configuration for a module:

- **[ecuc_sws_1016]** For mandatory containers, parameters and references (i.e. with `lowerMultiplicity` > 0 in their definition) at least the number of instances defined by the `lowerMultiplicity` shall be generated.

  E.g. the configuration of a CAN controller may contain the configuration of one or more hardware objects, depending on the hardware. The configuration of hardware objects is done in a subcontainer. Since at least one hardware object is always present, one instance of this subcontainer always has to be present and must be generated together with the enclosing container for the CAN controller.

- **[ecuc_sws_1017]** For optional containers, parameters and references (i.e. with `lowerMultiplicity` $= 0$ in their definition), no instances may be generated.

  E.g. the configuration may contain the definition of RX PDUs in a subcontainer. One subcontainer instance is defined for each PDU received. Since there may be no RX PDUs, it is well possible that no container instance needs to be generated.

- **[ecuc_sws_1018]** For containers with variable multiplicity (i.e. `lowerMultiplicity` $<$ `upperMultiplicity`), any number of instances between lower and upper multiplicity may be generated. (additional instances may be added during Editing of the configuration description).

  E.g., continuing the previous example, several instances may be generated if the definition of RX PDUs can be derived from the ECU extract of System description. If the ECU receives several frames on a CAN bus, at least one RX PDU is normally present per received frame.

- **[ecuc_sws_1019]** For the setting of the initial values for configuration parameters, the following sources shall be used (in decreasing priority)

  - **[ecuc_sws_1020]** Values fixed by the implementation as defined in the Vendor Specific Pre-configured Configuration Description. Since the module implementation fixes those configuration parameters, those values must be included in the base ECU configuration description and shall not be changed in later editing.

  - **[ecuc_sws_1021]** Values derived from the ECU extract of the system configuration. E.g. for COM stack configuration, the system description provides configuration information for bus speed, frame definitions etc, which can be taken over into the ECU configuration description.

    E.g. The signal definitions relevant for the COM stack can be derived from the ECU extract of system configuration. One container instance with all relevant parameter values taken from the system configuration will be generated for each signal.

  - **[ecuc_sws_1022]** Values provided by the implementor in the BSWMD in the Vendor Specific Recommended Configuration Description. Implementors may provide configuration settings in the BSWMD provided with their implementation. This allows the implementor to provide the integrator with his hints which values might be most useful for his implementation of the module on a specific ECU.

  - **[ecuc_sws_1023]** Default values provided as part of the parameter definition. Since each configuration parameter is defined only once, all instances of the parameter will have the same initial value when the default values is taken as input to the base configuration.

  [ecuc_sws_1024] If no initial value can be derived from any of these sources for a parameter, the parameter will be generated without an initial value.

[ecuc_sws_4004] If an existing `ECU Configuration Description` exist and an updated `ECU Extract of System Configuration` or `BSW Module Description` is released the existing `ECU Configuration Description` must be taken into consideration when updating to a new version of `ECU Configuration Description`, i.e, the `Generate Base ECU Configuration Description` activity shall consist of a merge functionality. This functionality is optional since the first time an `ECU Configuration Description` is generated there is no existing `ECU Configuration Description`.

5.3 Rules for Configuration Editors

Chapter 2.3.4 describes the methodology for editing configuration parameters. The following rules apply for a configuration editor supporting the methodology:

- **[ecuc_sws_4001]** The ECU Configuration Editor shall be able to generate the files containing parameters affecting other parameters. That is the xml-files of type `Module Configurations` described in chapter 2.3.4.
- **[ecuc_sws_4002]** The ECU Configuration Editor shall be able to perform a simple merge of `ECU Configuration Descriptions` as described in chapter 2.3.4.
- **[ecuc_sws_4003]** The ECU Configuration Editor shall be able to work with subsets of parameters. The subset shall be any combination of pre-compile time, link-time and post-build time parameters. This feature is to avoid editing wrong kind of parameters.
- **[ecuc_sws_4005]** The ECU Configuration Editor shall be able to generate and import files describing a specific aspect of the configuration of a module. The files that shall be generated and imported are `ModuleConfigurations`. The rationale for this is to support post-build time loadable configuration from a Configuration Management perspective. See chapter 2.3.2.3.1.

Following is a list (not complete) of additional requirements which a Configuration Editor shall support:

- **[ecuc_sws_2088]** When a `longName` (`LONG-NAME` in XML) is provided for a configuration element the Configuration Editor shall display the content of the `longName` this to it's users.

A Possible Implementations for the Configuration Steps

A.1 Alternative Approaches

This chapter contains description of alternative approaches and information that is not part of the AUTOSAR, but can be helpful and give some guidance.

A.1.1 Alternative Configuration Editor Approaches

[ecuc_sws_1031] The ECUC parameter definitions and ECUC descriptions are designed to support a variety of different tooling approaches. In the following, the different approaches that have been considered during the development of the specification are introduced. These tooling approaches are supported by ECUC parameter definition and ECUC description. Other approaches might be consistent with this specification, but have not been considered explicitly.

Tool suppliers have a high degree of freedom in the approach their tools may take to ECU Configuration. ECU Configuration tools might consist of a single monolithic editor capable of manipulating all aspects of ECU Configuration, it could be a core tool framework that takes plug-in components to manipulate particular aspects of ECU Configuration, it might be a set of specialized tools each capable of configuring a particular type or subset of software modules or, probably more likely, software vendors could supply individual custom tools to configure only the code blocks that they supply (similar to microprocessor vendors providing specialized debuggers for their own micros).

Common to the different tool approaches is that each configuration editor must be capable of reading an (possibly incomplete) `ECU Configuration Description` and writing back its modified configuration results in the same format. The modification may include changed values of ECU Configuration values and added instances of containers with all included ECU Configuration Values (only for containers/parameters with variable multiplicity).

In every case, the `ECU Configuration Description` is expected to be the point of reference, the backbone of the process.

The sections below look at some possible tool forms and identify some of their strengths and weaknesses.

A.1.1.1 Custom Editors (Informative)

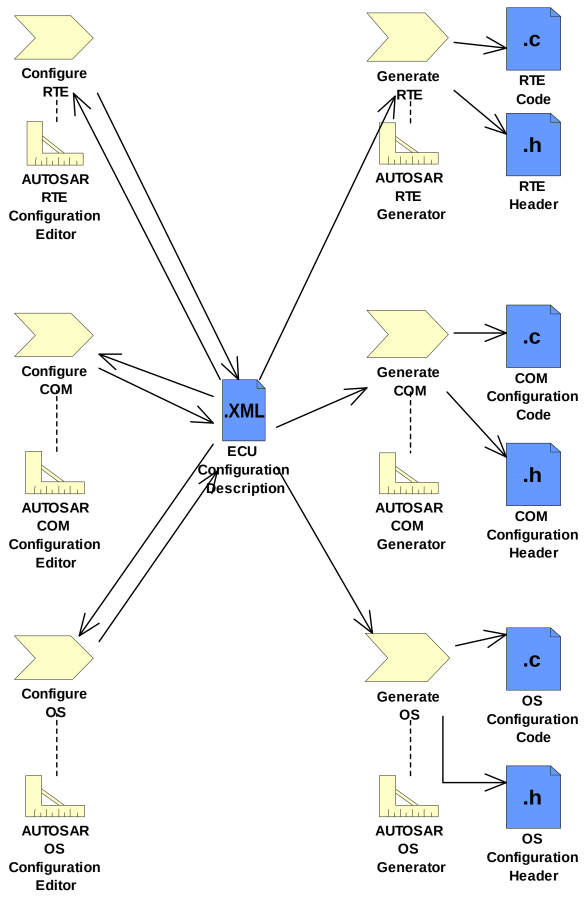

Figure A.1: Custom Editors and Generators

In the custom editors approach as shown in figure A.1, each BSW module is delivered bundled with a custom configuration editor and a generator (E.g. in figure A.1 the `AUTOSAR RTE Configuration Editor` and `AUTOSAR RTE Generator`). These tools can be optimized to the particular task of configuring one BSW module and would likely be quite powerful. The complex dependencies between the BSW module configuration and other configuration items in the `ECU Configuration Description` could be expressed and supported in the tool. Each vendor of a BSW module would need to provide a tool. System and ECU engineers would require a large number of tools to deal with the range of BSW modules. Each tool would probably have an individual look and feel and this could increase the training and experience required to become proficient.

A.1.1.2 Generic Tools (Informative)

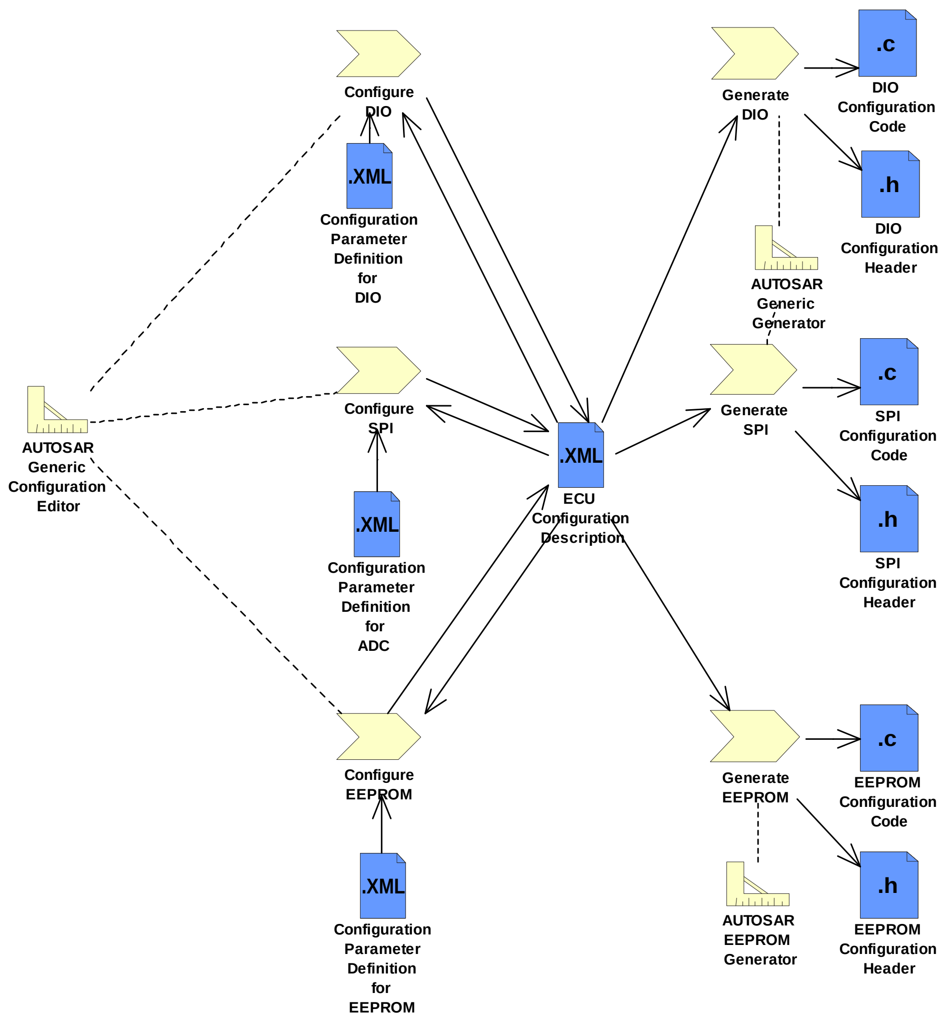
Configure DIO
.XML
Configuration Parameter Definition for DIO
Configure SPI
.XML
Configuration Parameter Definition for ADC
Configure EEPROM
.XML
Configuration Parameter Definition for EEPROM
AUTOSAR Generic Configuration Editor
.XML
ECU Configuration Description
Generate DIO
.c
DIO Configuration Code
.h
DIO Configuration Header
AUTOSAR Generic Generator
Generate SPI
.c
SPI Configuration Code
.h
SPI Configuration Header
Generate EEPROM
.c
EEPROM Configuration Code
.h
EEPROM Configuration Header
AUTOSAR EEPROM Generator

Figure A.2: Generic Configuration Editor

An `AUTOSAR Generic Configuration Editor` as shown in figure A.2 would be able to configure any parameter defined in `Configuration Parameter Definitions`. It would read those definitions and provide a generic interface to edit values for all parameters in the `ECU Configuration Description`. It would only be able to resolve the relatively simple dependencies explicitly defined in the `Configuration Parameter Definitions`. Only a limited number of editors would be required, maybe only one, and the look and feel is less likely to vary greatly between generic tools. Training and tooling costs are therefore likely to be lower. Examples of such tools that already exist or are soon to exist are tresos, GENy, DAvE and MICROSAR, although it should be noted that no ECU Configuration Editors exist at the time of writing this document. On the generation side, either a generic generator may be used, or custom generators for the individual modules.

A.1.1.3 Tools Framework (Informative)

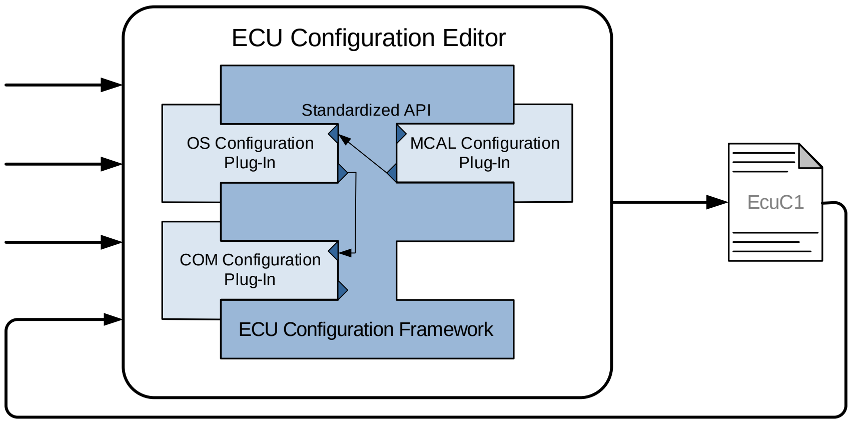
ECU Configuration Editor
Standardized API
OS Configuration Plug-In
MCAL Configuration Plug-In
COM Configuration Plug-In
ECU Configuration Framework
EcuC1

Figure A.3: Framework Tools

The tool framework as shown in figure A.3 is a cross between custom tools and generic tools where dedicated configuration steps (OS, COM, MCAL, etc.) are integrated as plug-ins into the common ECU Configuration framework. The heart of the tool would be a framework that provides certain core services such as importing and exporting data from standard file formats, maintaining standard internal data structures and providing an HMI to the user. This ensures that the `ECU Configuration Description` is read, interpreted and written in a defined way. The frame could also monitor and control the user / project work flow. It provides a low initial tooling and training investment. The power of the approach would be the ability to add plug-in modules that extend the core functionality. These would allow very sophisticated features, potentially dedicated to one BSW module, to be added by individual suppliers. Additionally, it would be possible to add generic plug-ins that addressed a specific aspect of configuration across all BSW modules. This approach relies upon a standard framework: multiple framework standards could lead to high tool costs. An example of this kind of tool is the LabVIEW test and measurement tool from National Instruments and the Eclipse framework.

A.1.2 Alternative Generation Approaches

As stated before, the `ECU Configuration Description` is the only configuration source that stores the configuration parameters for all modules of an AUTOSAR based ECU. However, for several modules such as OS, existing configuration languages have already been established. Those languages probably will in future still be used for non-AUTOSAR systems. Thus, modules that are used both for AUTOSAR and non-AUTOSAR systems must support different configuration languages in parallel. This can be implemented in different ways, as shown in figure A.4.

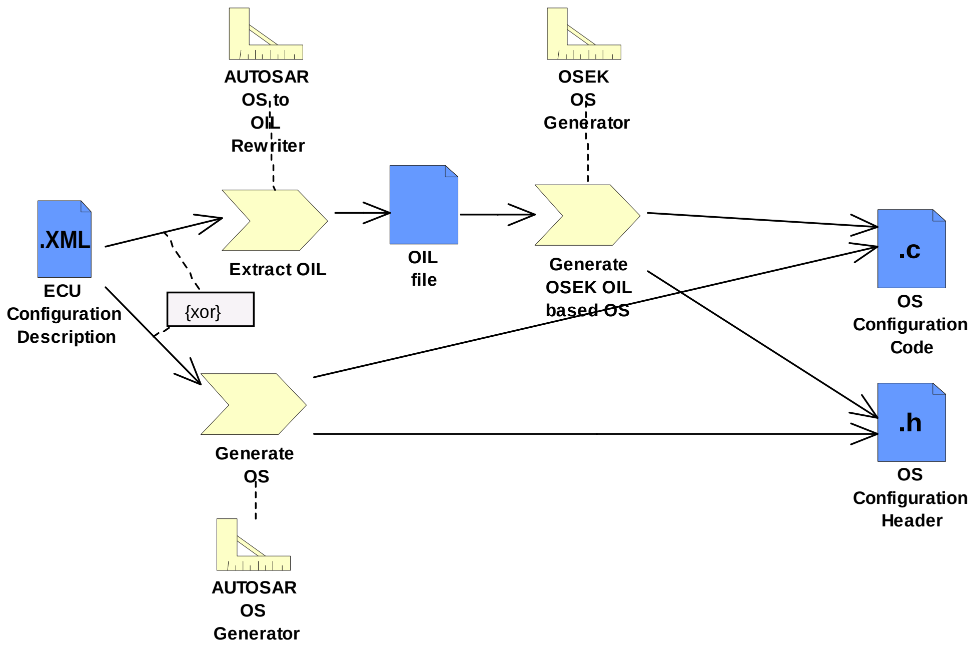

Figure A.4: Module generator implementation alternatives, example for OS

In a fully AUTOSAR based approach, the generator reads in the `ECU Configuration Description` and output the relevant source code directly in one step, supported by a `AUTOSAR OS Generator` in the example given. To support reuse of existing generator tools, this single step can be split into two steps. Then the first activity is to extract all OS configuration information from the `ECU Configuration Description` using an `AUTOSAR OS to OIL Rewriter` and to store it in the format used by the legacy tools (`OIL file` in the example chosen). The existing `OSEK OS Generator` may then read the intermediate format to generate the code. However, the intermediate format must **not** be changed by any legacy configuration tools, since those changes would possibly make the overall configuration inconsistent, and since changes would be lost with the next generation of the intermediate format.

[ecuc_sws_1025] Thus, none of the activities (extract, generate) shown in figure A.4 must include any engineering step with decisions taken by an engineer. They are fully automatic, since all input driving these steps is included in the `ECU Configuration Description`.