UML Design Guideline (SMCAL) – V6.0

September, 2016

freescale™

Confidential and Proprietary

Purpose, Doc location, Revision History

Purpose:
This document provides guidance and contains rules to be applied during SMCAL drivers UML design. The documents is used for training purposes and as input for UML Design Checklist template.
Location:
This document is placed at the below compass location:
http://compass.freescale.net/livelink/livelink?func=ll&objId=227962899&objAction=browse&viewType=1
Revision History:

Version	Date	Owner/Name	Rev. Changes
1.0 – D01	23 Sept 2012	Simona Almajan, Bogdan Andone	First draft version
1.0	1 Oct 2012	Simona Almajan, Bogdan Andone	Official version
2.0	12 July 2013	Bogdan Andone	Updates to rules, extended the rules
3.0	9 Nov 2013	Alexandra Amarandei	Added Validate Functions
4.0	10 Oct 2015	Alexandra Amarandei Eugenia Neacsa	Added examples of control flow diagrams, minor changes
5.0	April 2016	Alexandra Amarandei Eugenia Neacsa	Added new rule 7.13 Added naming convention rules: 12.1, 12.2, 13.3, 12.4, 12.5, 12.6 Peer review **48181020**

Purpose, Doc location, Revision History

Revision History – Continue

Version	Date	Owner/Name	Rev. Changes
6.0	September 2016	Alexandra Amarandei Eugenia Neacsa	Removed rule 1.1 Renumbered all other rules from section 1. Peer review 48767182

Agenda

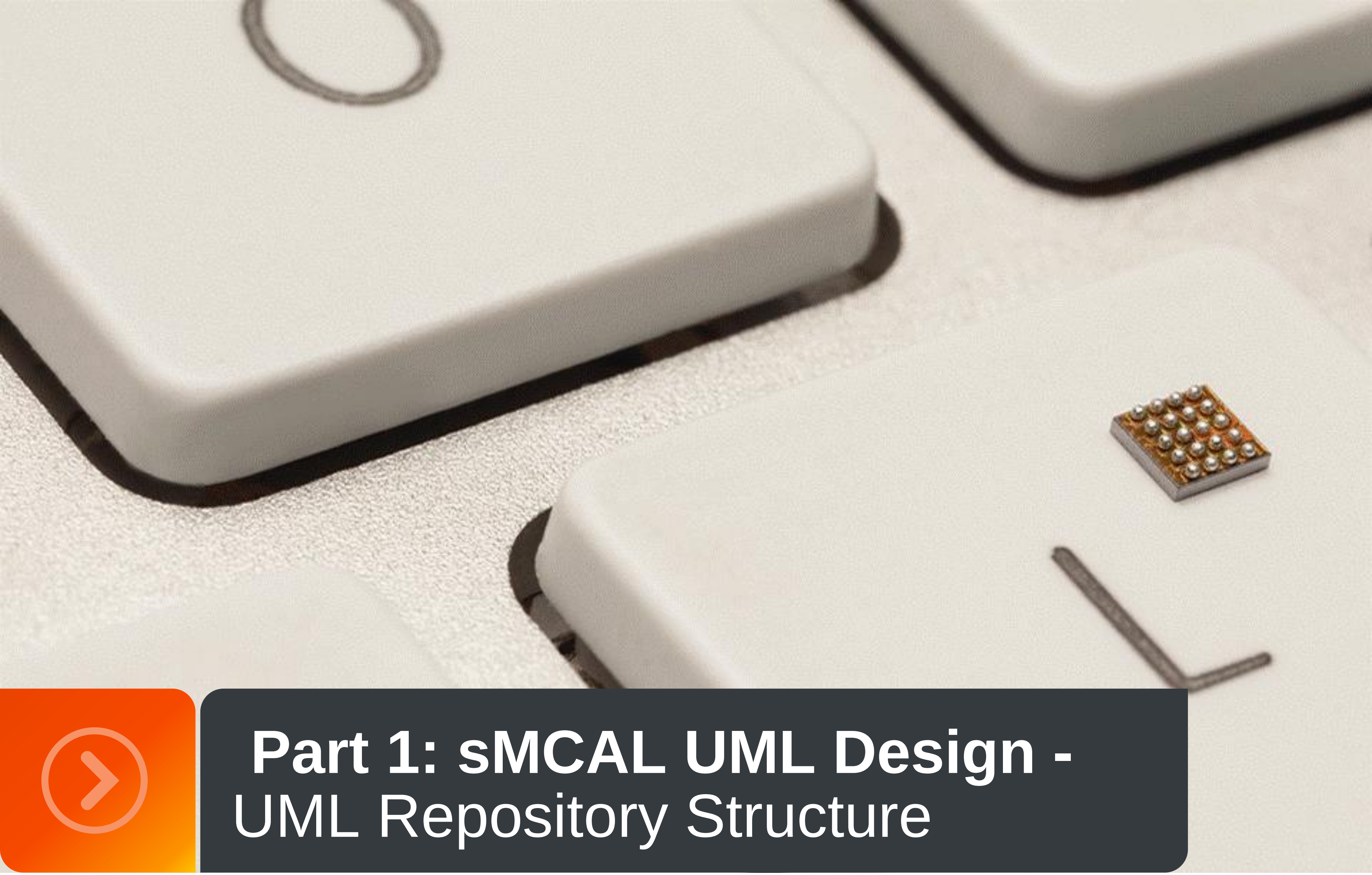
Part 1: SMCAL UML Design

- ✓sMCAL UML Design Structure
- ✓sMCAL Top Level Diagrams
- ✓<Mdl> Architecture Overview
- ✓<Mdl> Architectural Design →Static View
- ✓Unit Detailed Design (HLD, IPW, IP layers) →Static View
- ✓Unit Detailed Design (HLD, IPW, IP layers) → Behavioral View
- ✓Plugin Configuration
- ✓Generic Rules
- ✓ReqTracer Mapping
- ✓UML Design Review Checklist

(backup slides)Part 2: Generic UML Language Structures

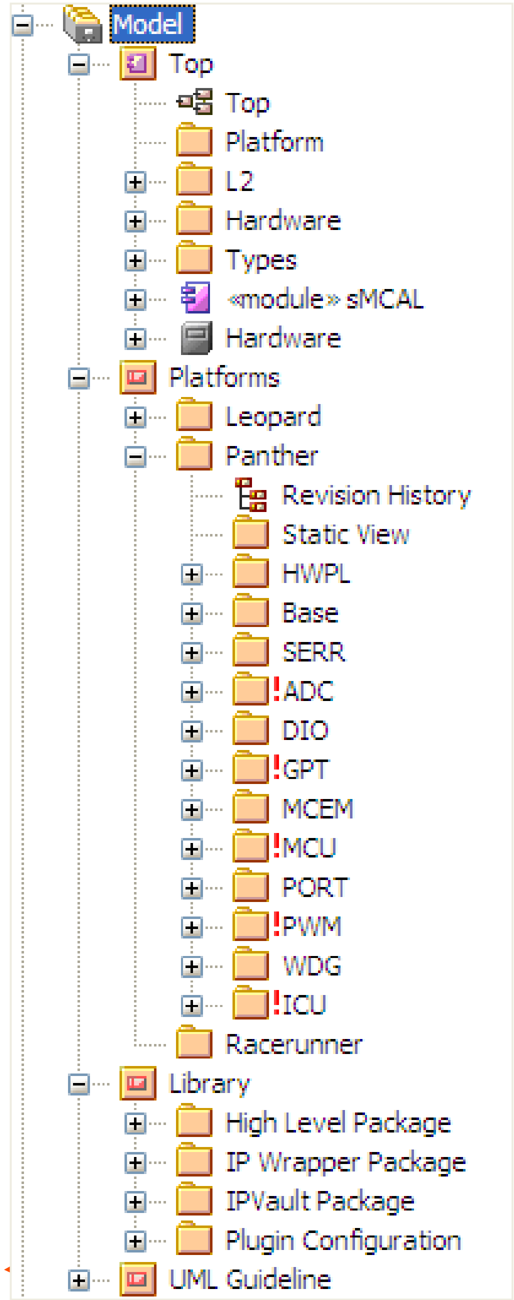
- ✓Structural Diagrams Elements and Connectors
- ✓Behavioral Diagrams Elements and Connectors

(backup slides)Part 3: ISO26262 – Part 6 SW Unit Design Requirements



Part 1: sMCAL UML Design - UML Repository Structure

UML Repository Structure



- Model
  - Top
    - Top
    - Platform
    - L2
    - Hardware
    - Types
    - «module» sMCAL
    - Hardware
  - Platforms
    - Leopard
    - Panther
      - Revision History
      - Static View
      - HWPL
      - Base
      - SERR
      - !ADC
      - DIO
      - !GPT
      - MCEM
      - !MCU
      - PORT
      - !PWM
      - WDG
      - !ICU
    - Racerunner
  - Library
    - High Level Package
    - IP Wrapper Package
    - IPVault Package
    - Plugin Configuration
  - UML Guideline

Top:

1. Top/Top Diagram - showing the SMCAL -L2 – HW Interface
2. L2 – contains application layer components designed for Safety Application (DEM, DET, EcuM, SchM)
3. Hardware – hardware IPs interface (register access/ranges interrupt handlers ..)
4. Types – ASR type definitions

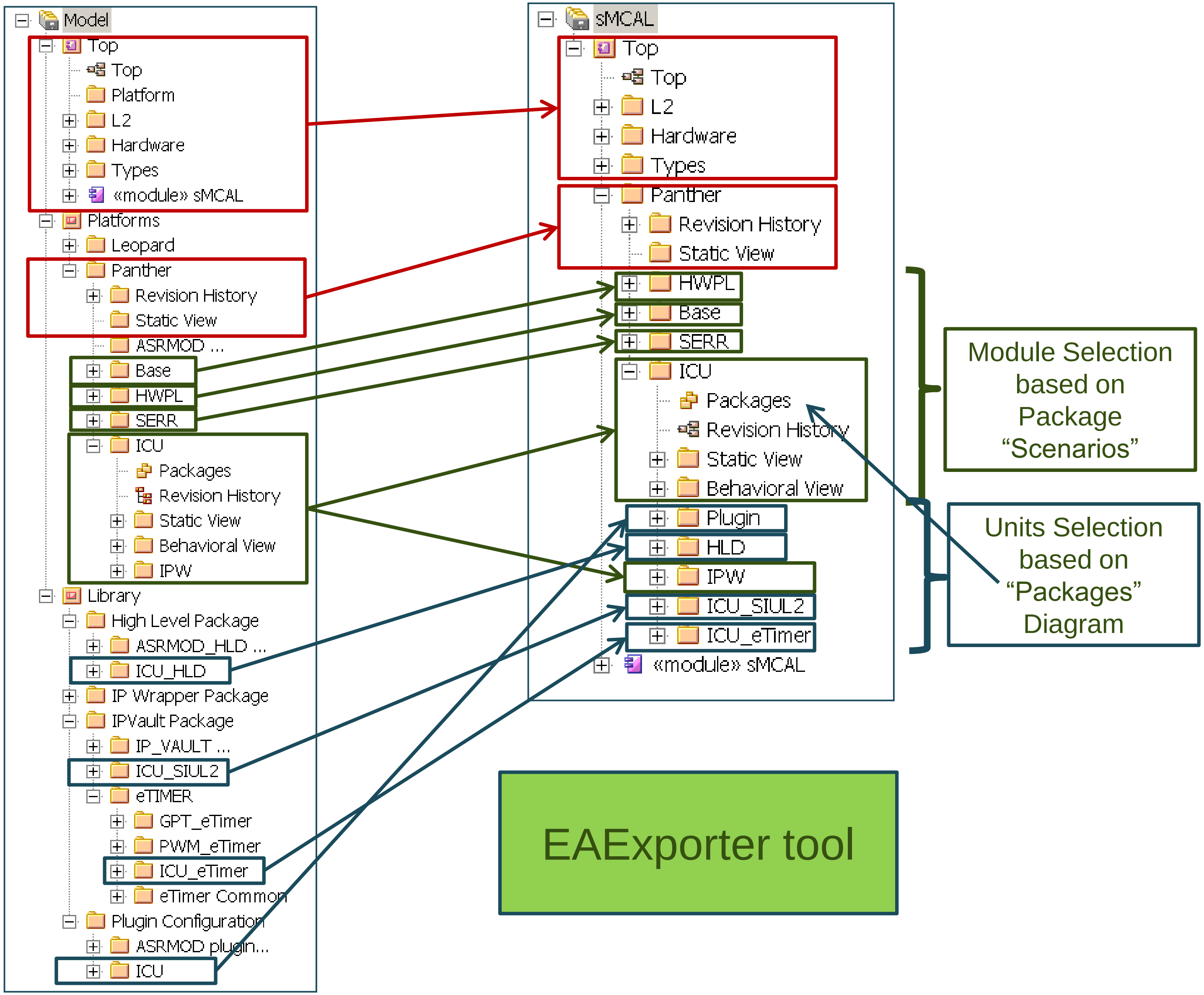
Platforms: contains the collection of UML designs for the modules provided for each HW platform

1. Static and Dynamic Views at ASR module level
2. Unit designs for HLD, IPW and IP layers
3. Plugin design

Library: contains the unit or plugin designs which can be shared between platforms

1. Static and Dynamic Views of HLD, IPW and IP layers
2. Plugin designs

Exporting the UML Design from EA DB



Module Selection based on Package “Scenarios”

Units Selection based on “Packages” Diagram

EAExporter tool

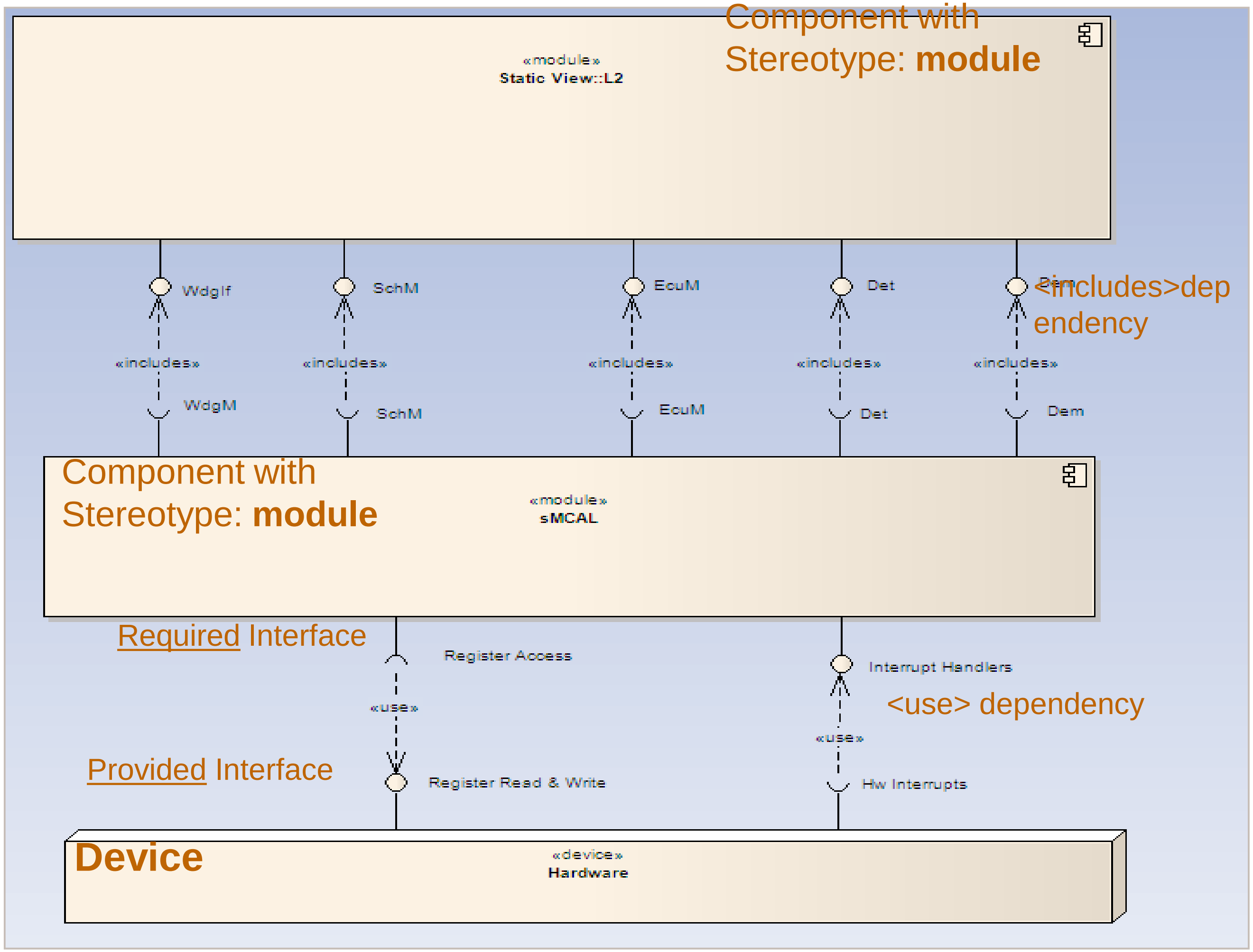
sMCAL Top Level Diagrams

- Top
- L2
- HW Platform
- ASR Types

Top Level Diagram

Rule 1.1:

A top level component diagram shall show interfaces between sMCAL external application and hardware



Component with Stereotype: **module**

<includes>dependency

Component with Stereotype: **module**

Required Interface

Provided Interface

<use> dependency

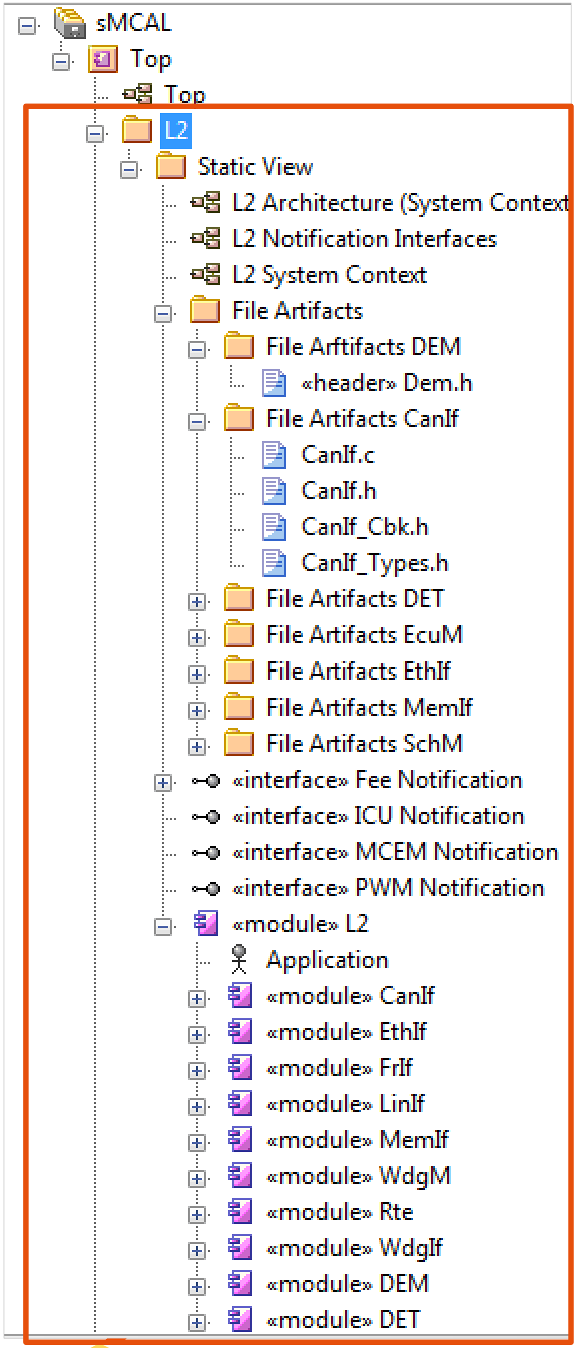
Device

- **L2**: Layers above MCAL
- **Hardware**: microcontroller hardware designed to support MCAL functionality

The three Layers are linked with **required** and **provided interfaces** (Expose Interfaces)

Component Type Diagram

L2 Package Overview



- sMCAL
  - Top
    - Top
    - L2
      - Static View
        - L2 Architecture (System Context
        - L2 Notification Interfaces
        - L2 System Context
        - File Artifacts
          - File Arftifacts DEM
            - «header» Dem.h
          - File Artifacts CanIf
            - CanIf.c
            - CanIf.h
            - CanIf_Cbk.h
            - CanIf_Types.h
          - File Artifacts DET
          - File Artifacts EcuM
          - File Artifacts EthIf
          - File Artifacts MemIf
          - File Artifacts SchM
        - «interface» Fee Notification
        - «interface» ICU Notification
        - «interface» MCEM Notification
        - «interface» PWM Notification
        - «module» L2
          - Application
          - «module» CanIf
          - «module» EthIf
          - «module» FrIf
          - «module» LinIf
          - «module» MemIf
          - «module» WdgM
          - «module» Rte
          - «module» WdgIf
          - «module» DEM
          - «module» DET

Rule 1.2:

A dedicated package (named "L2") shall encapsulate the interfaces definition of the external modules referenced by sMCAL: DEM, Det, Rte, EcuM, …

Package Content – Static View:

✓L2 Architecture – System Context

✓L2 Notification Interfaces

✓L2 System Context

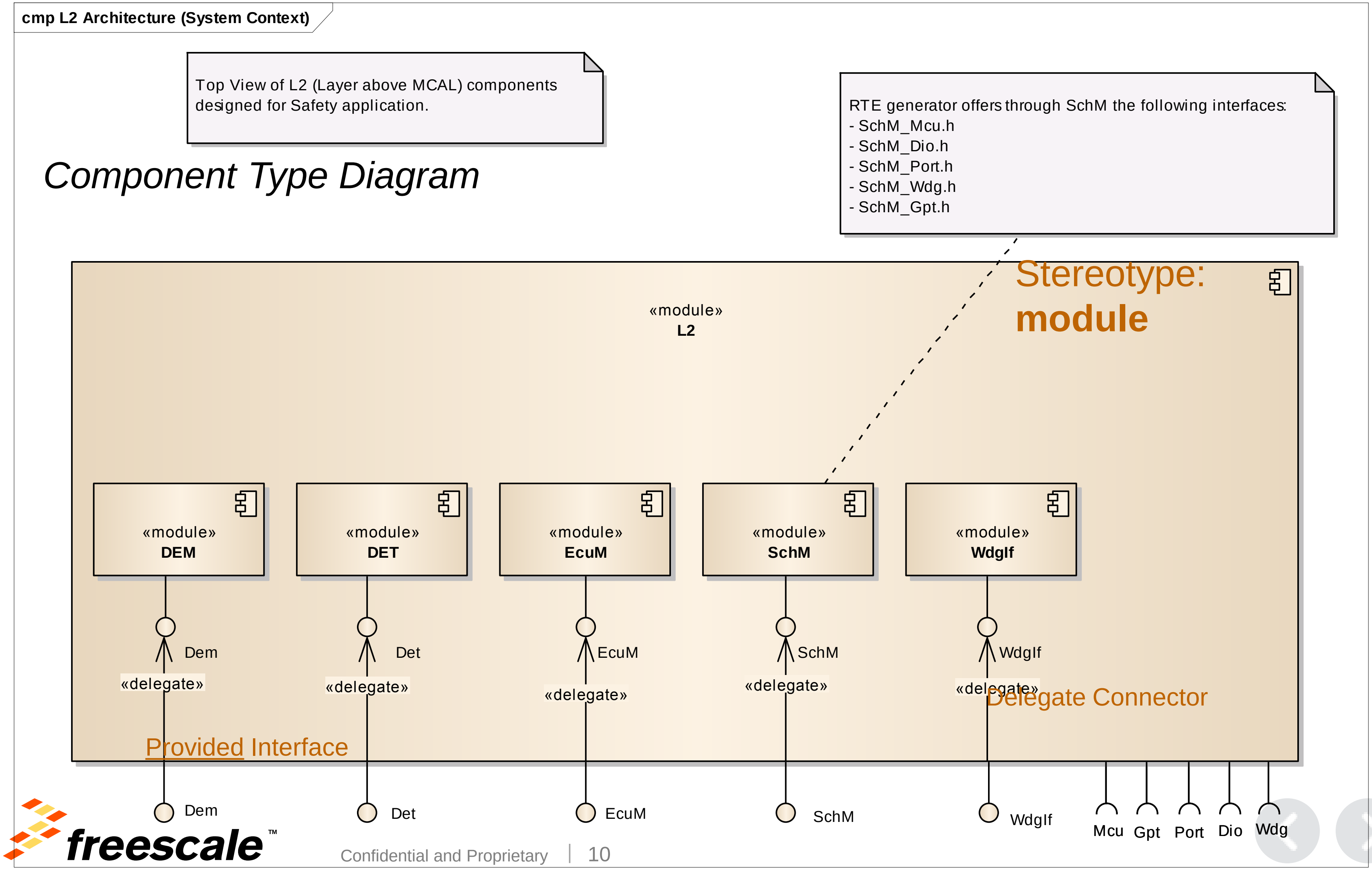
✓File Artifacts for Stubs

✓Stubs Interfaces

L2 Package: Static View: “L2 Architecture” – System Context

Rule 1.2:

A dedicated package (named "L2") shall encapsulate the interfaces definition of the external modules referenced by sMCAL: DEM, Det, Rte, EcuM, …



Top View of L2 (Layer above MCAL) components designed for Safety application.

Component Type Diagram

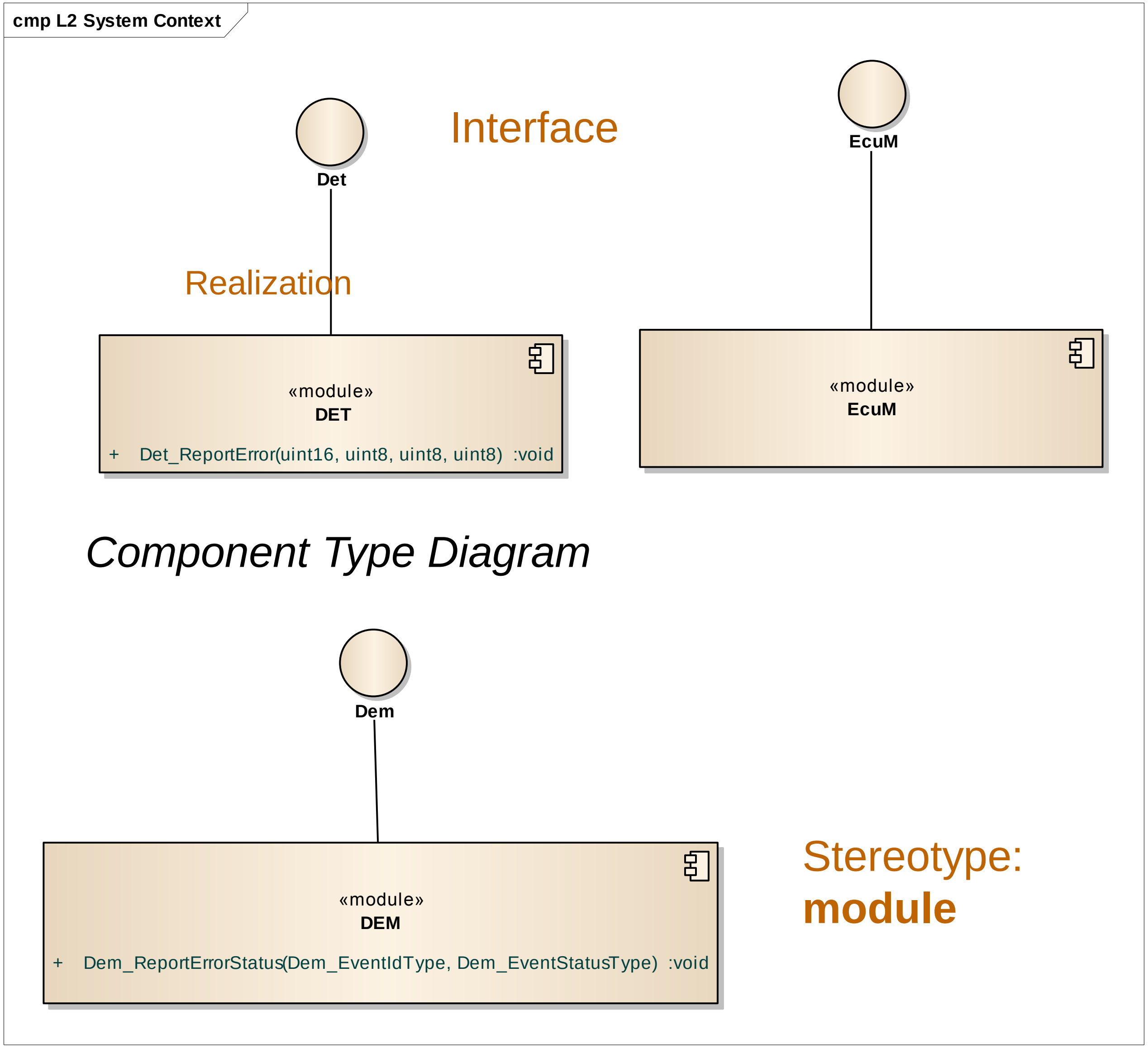
RTE generator offers through SchM the following interfaces:
- SchM_Mcu.h
- SchM_Dio.h
- SchM_Port.h
- SchM_Wdg.h
- SchM_Gpt.h

Stereotype: **module**

Delegate Connector

Provided Interface

L2 Package: Static View: “L2 System Context”



Rule 1.3:

Each external module will be designed as a UML component having the name of AUTOSAR module, and the stereotype "module“

Rule 1.4:

Each external module component have an associated interface definition defining at least the functions referenced from sMCAL; the interface will be deployed as an UML interface object (not as exposed interface)

Remark: The interface will be further used in static views of module specific diagrams

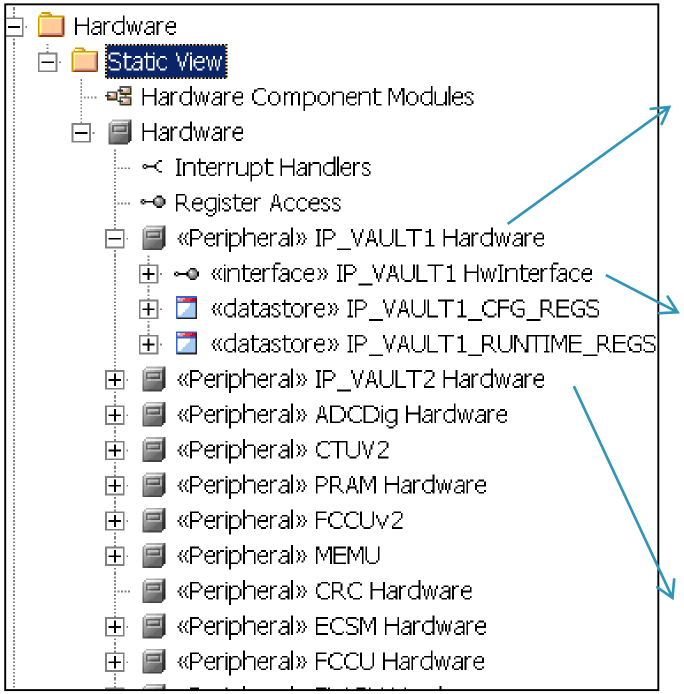
“Hardware” Package Overview

Rule 1.5:

A dedicated package (named "Hardware") shall encapsulate the interface definition of hardware IPs used by sMCAL.

Covers:

✓Interrupt Handlers and Registers Access for HW Peripherals



- Hardware
  - Static View
    - Hardware Component Modules
    - Hardware
      - Interrupt Handlers
      - Register Access
      - «Peripheral» IP_VAULT1 Hardware
        - «interface» IP_VAULT1 HwInterface
        - «datastore» IP_VAULT1_CFG_REGS
        - «datastore» IP_VAULT1_RUNTIME_REGS
      - «Peripheral» IP_VAULT2 Hardware
      - «Peripheral» ADCDig Hardware
      - «Peripheral» CTUV2
      - «Peripheral» PRAM Hardware
      - «Peripheral» FCCUv2
      - «Peripheral» MEMU
      - «Peripheral» CRC Hardware
      - «Peripheral» ECSM Hardware
      - «Peripheral» FCCU Hardware

Rule 1.6:

Each IP shall be deployed as an UML Device element having the name of the Hardware IP and the stereotype "Peripheral“

Rule 1.7:

Each hardware IP shall have a generic HW interface, designed as an UML interface object (not as exposed interface) with the name "<IP_NAME> HwInterface“

Remark: The interface will be further used in static views of module specific diagrams

Rule 1.8:

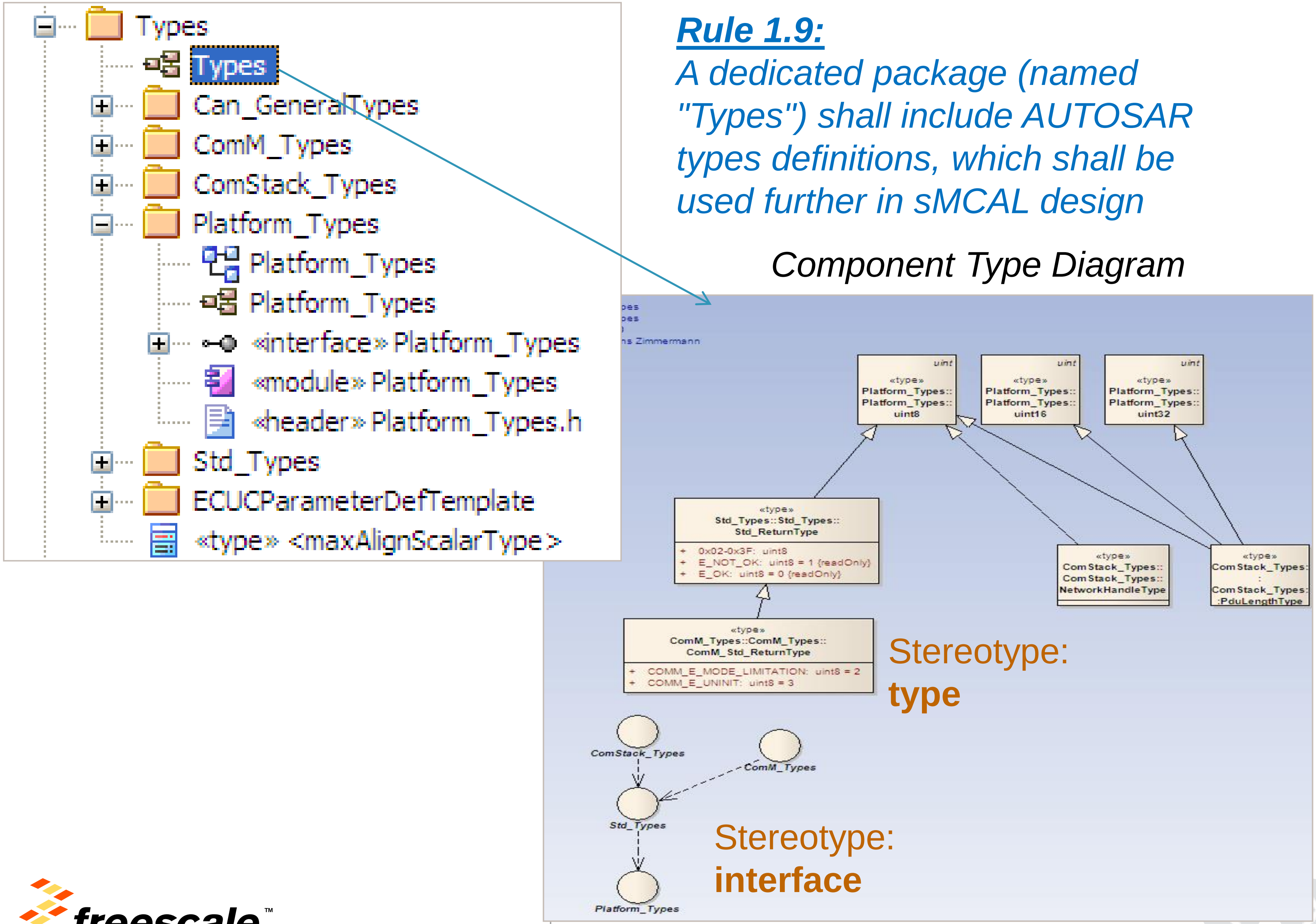
Each hardware IP shall have embedded datastores for the registers or registers ranges/partitions which are used for controling its behavior from sMCAL; the granularity of the datastores definition (register level, registers category level, ...) is module specific, depending of register sharing scenarios.

Remark: The data-stores will be further used in activity diagrams

Autosar Types

Rule 1.9:

A dedicated package (named "Types") shall include AUTOSAR types definitions, which shall be used further in sMCAL design



Types
Types
Can_GeneralTypes
ComM_Types
ComStack_Types
Platform_Types
Platform_Types
Platform_Types
«interface» Platform_Types
«module» Platform_Types
«header» Platform_Types.h
Std_Types
ECUCParameterDefTemplate
«type» <maxAlignScalarType>
pes
pes
)
ns Zimmermann
uint
«type» Platform_Types:: Platform_Types:: uint8
uint
«type» Platform_Types:: Platform_Types:: uint16
uint
«type» Platform_Types:: Platform_Types:: uint32
«type» Std_Types::Std_Types:: Std_ReturnType
+ 0x02-0x3F: uint8
+ E_NOT_OK: uint8 = 1 {readOnly}
+ E_OK: uint8 = 0 {readOnly}
«type» ComStack_Types:: ComStack_Types:: NetworkHandleType
«type» ComStack_Types: : ComStack_Types: :PduLengthType
«type» ComM_Types::ComM_Types:: ComM_Std_ReturnType
+ COMM_E_MODE_LIMITATION: uint8 = 2
+ COMM_E_UNINIT: uint8 = 3
ComStack_Types
ComM_Types
Std_Types
Platform_Types

Component Type Diagram

Stereotype: **type**

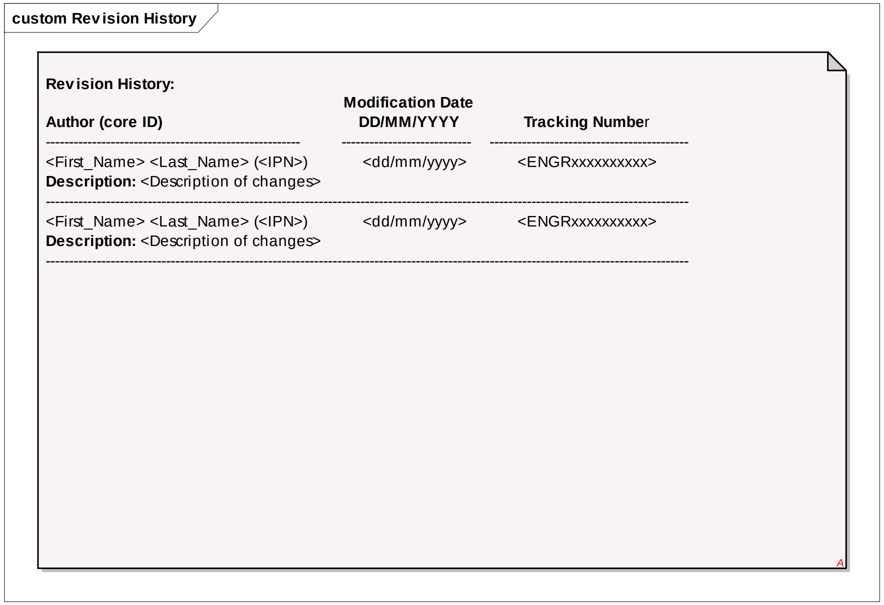
Stereotype: **interface**

<Mdl> Architecture Overview

Revision History Rules

Rule 2.1:

The Revision History diagram shall be included at module level:
Top/<Platform>/<MLD>/Revision History
The Revision History shall be up to date.
Remark: See UML template



custom Revision History

Revision History:

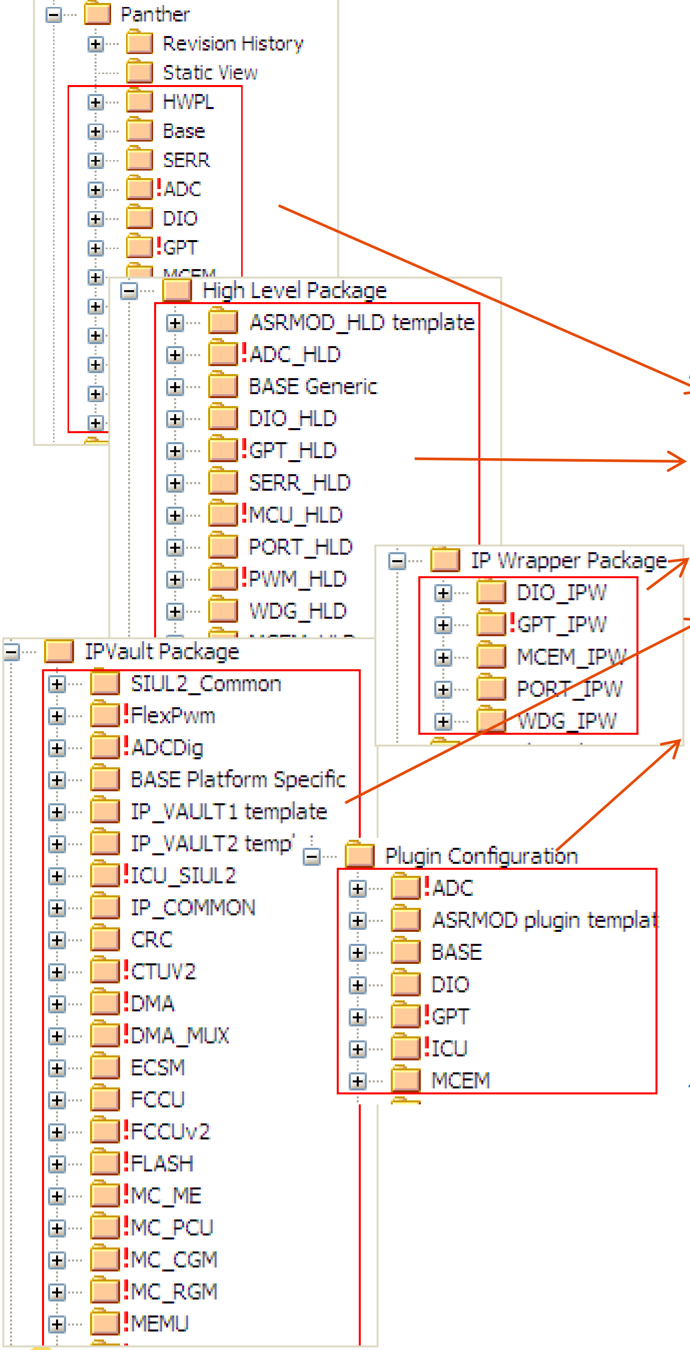
Author (core ID)	**Modification Date DD/MM/YYYY**	**Tracking Number**
<First_Name> <Last_Name> (<IPN>) **Description:** <Description of changes>	<dd/mm/yyyy>	<ENGRxxxxxxxxxxx>
<First_Name> <Last_Name> (<IPN>) **Description:** <Description of changes>	<dd/mm/yyyy>	<ENGRxxxxxxxxxxx>

Rule 5.2:

The Revision History diagram shall be included at unit level:
Top/<Platform>/<MLD>/<Mdl Unit>/Revision History
The Revision History shall be up to date.
Remark: See UML template

Note: Plugin also shall contain revision history, the same format

<Mdl> Architecture Overview



***Rule 2.2**:*

Each sMCAL UML design shall be split in the following packages:

***-<MDL>:** architectural design of an ASR module*

***-<MDL>\<MDL>_HLD:** detailed design of High Level Driver layer*

***-<MDL>\<MDL>_IPW:** detailed design of IP Wrapper layer*

***-<MDL>\<IP_VAULT>:** detailed design of the low level driver layer*

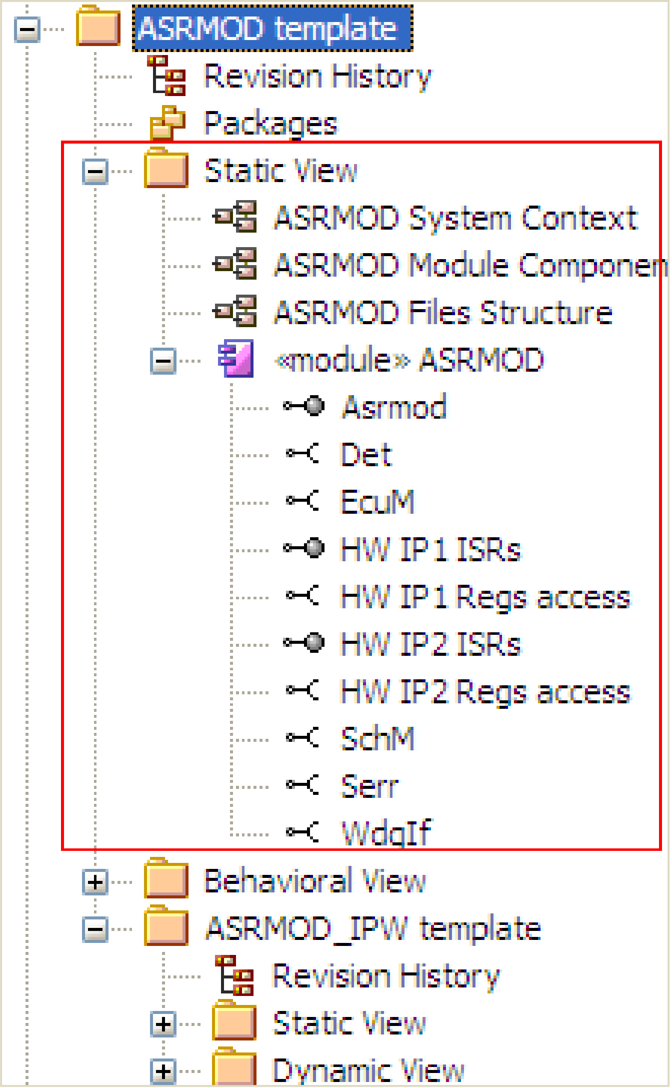
***-<MDL>\<MDL>_Plugin:** static views of plugin definition for the module*

Rule 2.3:

Architectural Design of a module shall have
Static View containing the static diagrams of the module

<Mdl> Architectural Design – Static View

<Mdl> Static View



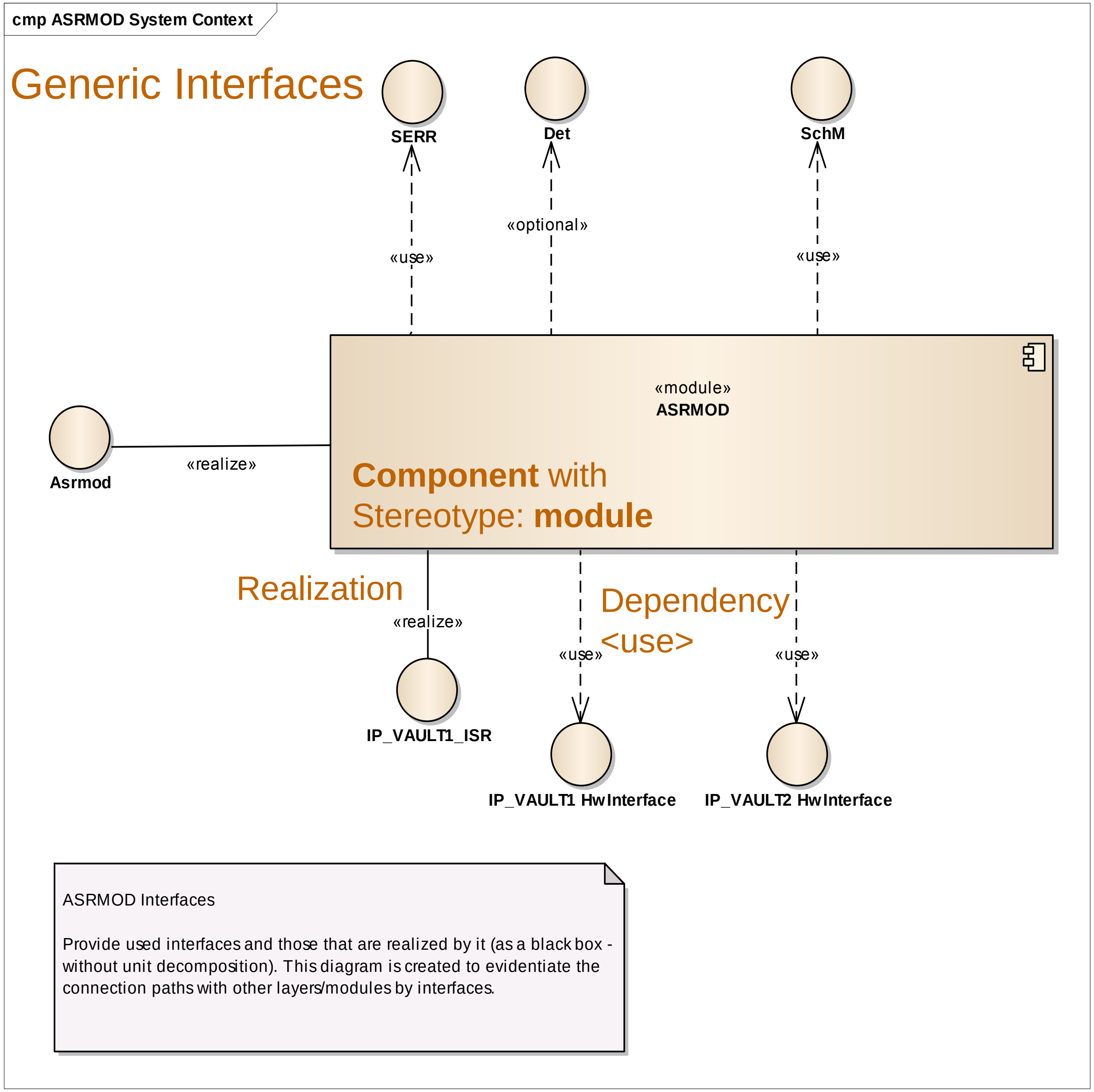
- ASRMOD template
  - Revision History
  - Packages
  - Static View
    - ASRMOD System Context
    - ASRMOD Module Componen
    - ASRMOD Files Structure
    - «module» ASRMOD
      - Asrmod
      - Det
      - EcuM
      - HW IP1 ISRs
      - HW IP1 Regs access
      - HW IP2 ISRs
      - HW IP2 Regs access
      - SchM
      - Serr
      - WdgIf
  - Behavioral View
  - ASRMOD_IPW template
    - Revision History
    - Static View
    - Dynamic View

Rule 3.1:

The sMCAL module will be designed as a UML component having the name of the AUTOSAR module, and the stereotype "module"

Remark: See the UML Template

<Mdl> System Context Diagram



ASRMOD Interfaces

Provide used interfaces and those that are realized by it (as a black box - without unit decomposition). This diagram is created to evidentiate the connection paths with other layers/modules by interfaces.

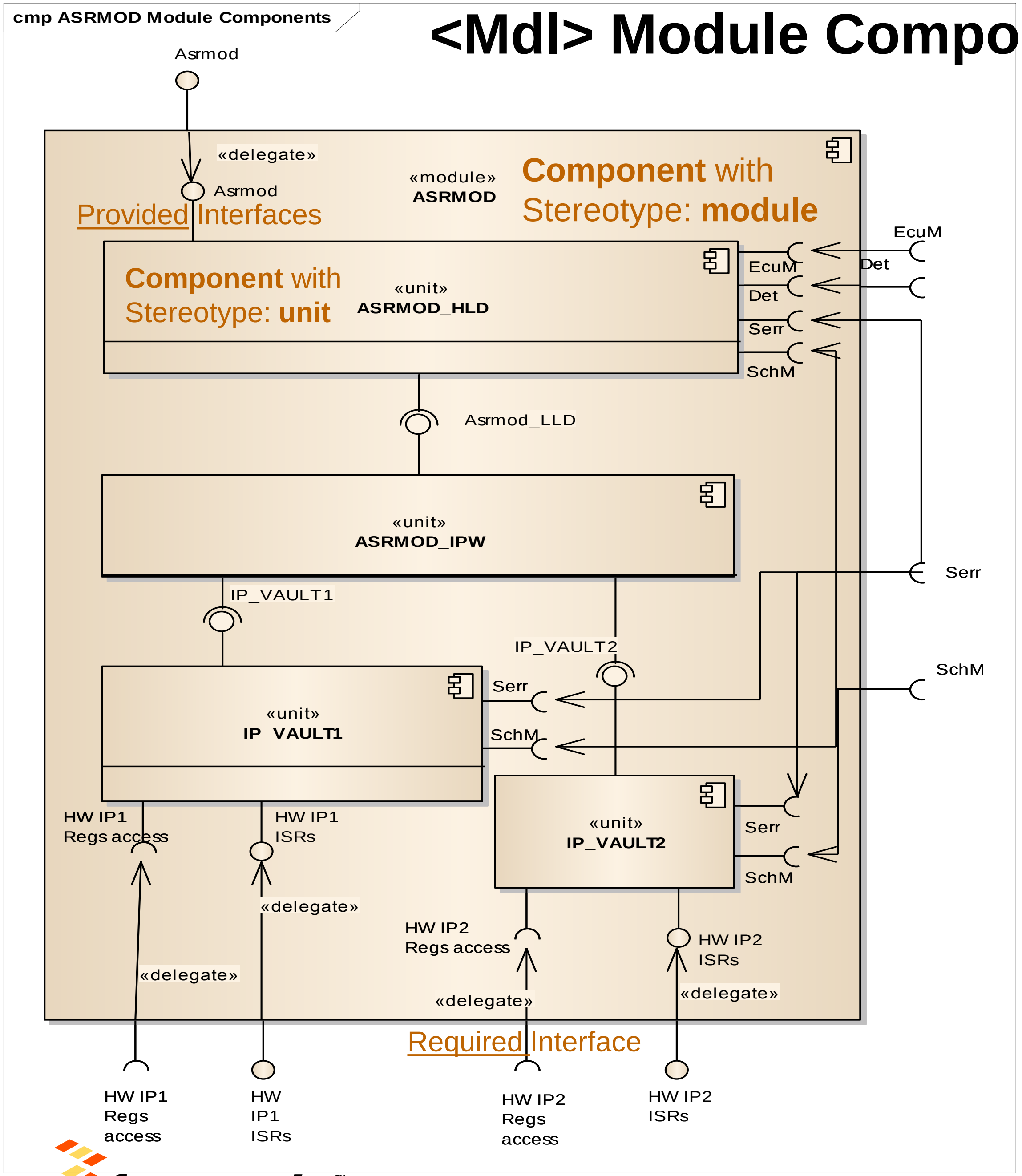
Rule 3.2:

A component diagram (named "<MDL> System Context") will show the interfaces of the sMCAL module with the external software application and HW peripherals..

Remark: See the UML Template

Component Type Diagram

<Mdl> Module Components Diagram - Layers



Rule 3.3:

A component diagram (named "<MDL> Module Components") will show the module decomposition in layers and the interfaces between these layers.

Goal:

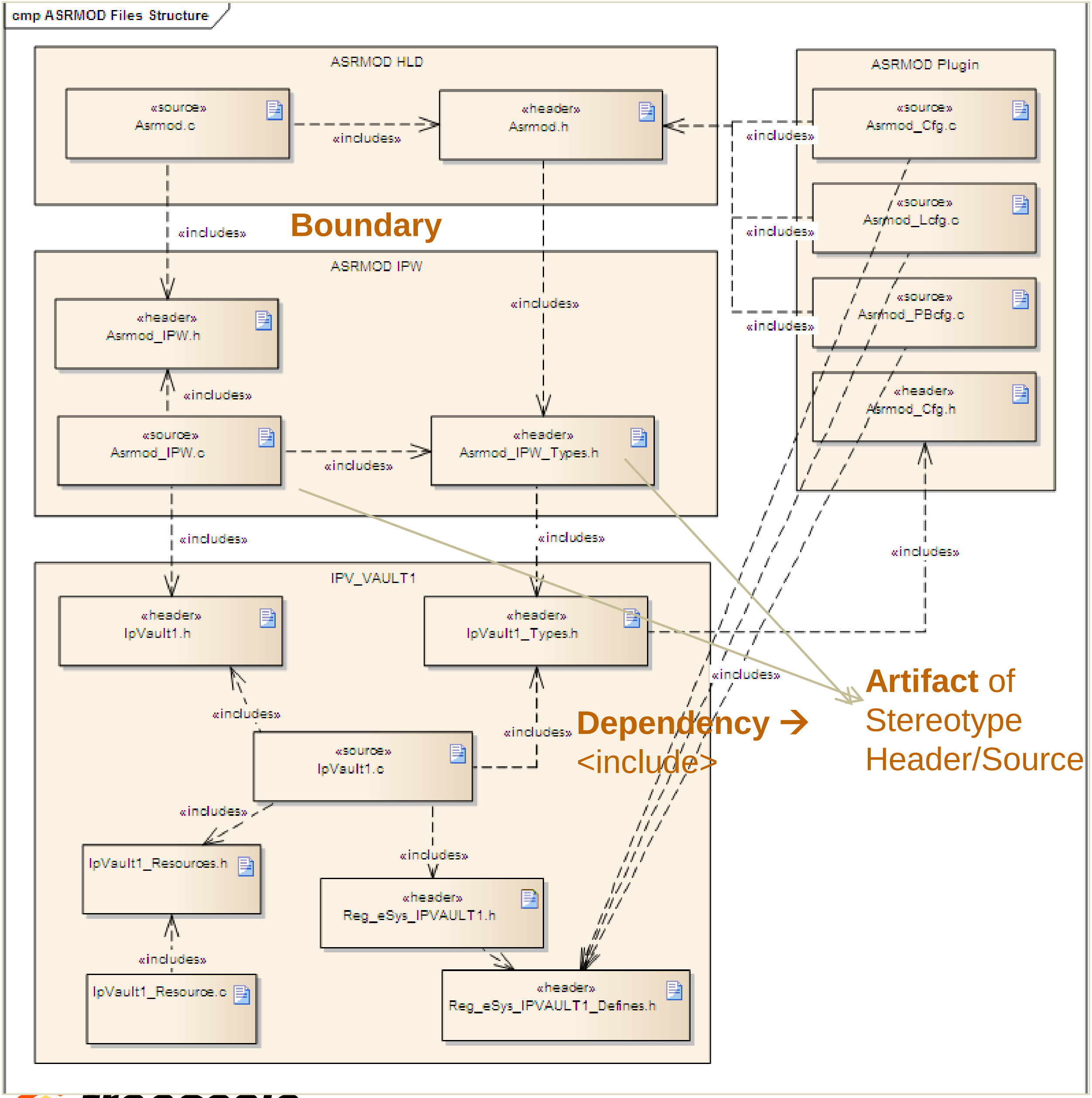
Understanding how the a module is decomposed in units and how they are interconnected

Remark: The module component diagram from the UML template is provided as a guidance. For enhanced readability and maintainability, it can be deployed in a simplified way for specific modules.

Component Type Diagram

<Mdl> Files Structure Diagram

Component Type Diagram



Boundary

Dependency → <include>

Artifact of Stereotype Header/Source

HLD Layer

IPW Layer

+ Driver Plugin

HW IP Layer

Rule 3.4 :

A component diagram (named "<MDL> Files Structure") shall show the files organization and dependencies at module level

<Mdl> Files Structure Diagram - Additional Rules

Goals:

- Have a big picture on the way the units are deployed in implementation files
- Understanding the deployment of interfaces in header files; check for circular dependencies
- Validate layers isolation

Rule 4.1

In a Files Structure diagram the files are designed as artifacts, with the stereotypes "header" or "source“

Rule 4.2

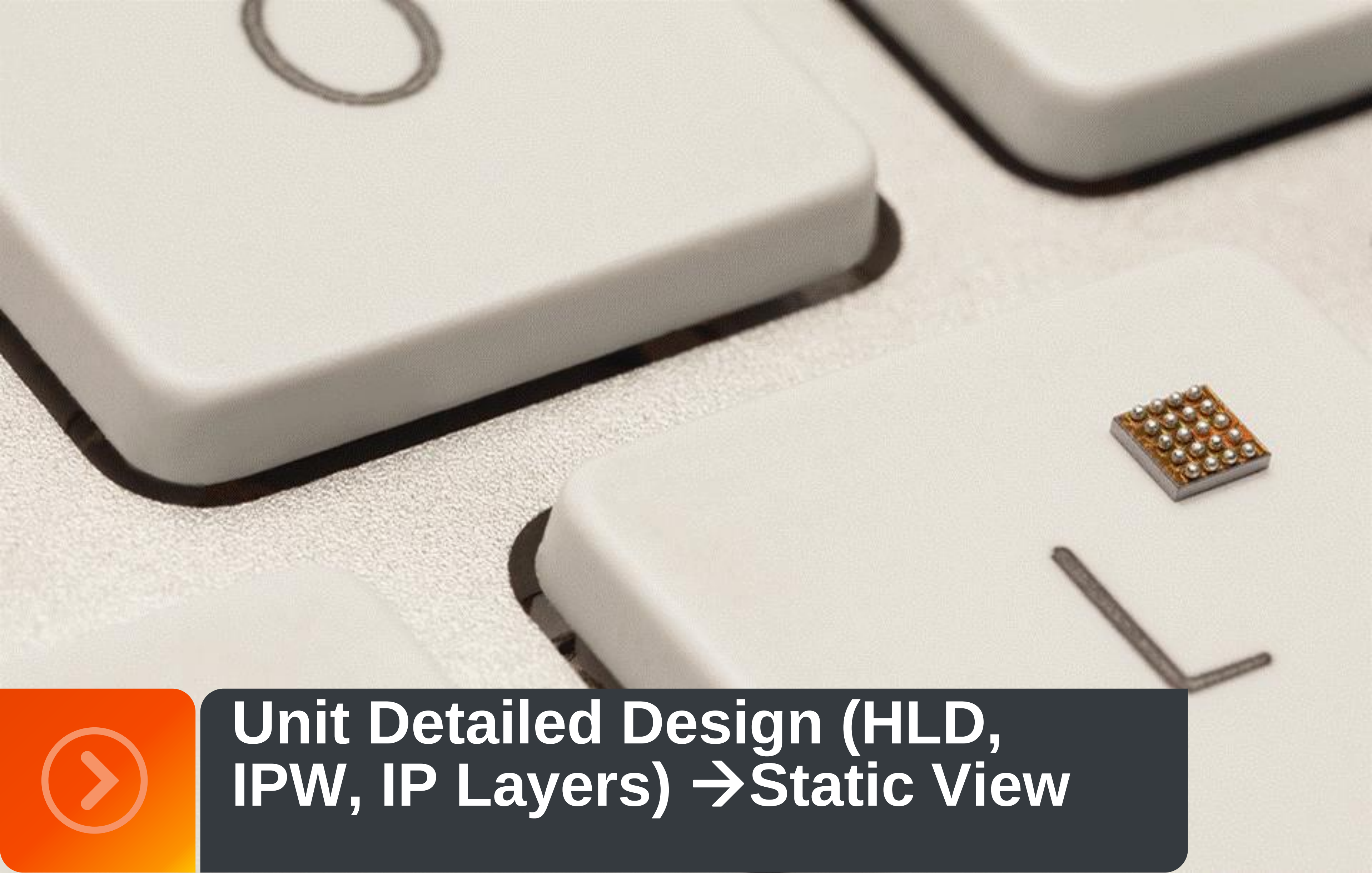
The file names shall be consistent with the established naming conventions

Rule 4.3

In a Files Structure diagram the files belonging to a given unit shall be isolated by boundaries

Rule 11.1

In a Files Structure diagram, the including dependencies between units shall follow the layering approach:
- no cyclic inclusions
- include dependencies shall be only to the layers bellow or above (avoid skipping layers)
- specific configuration header files for each layer/unit



Unit Detailed Design (HLD, IPW, IP Layers) →Static View

Generic Unit Level Design Rules 1/4

- ➢These are generic rules applicable for all three Layers/Units

Rule 5.1: *Unit Level packages contain the detailed design of ASR layers*

Rule 5.2**:** *The Revision History diagram shall be included at unit level: Top/<Platform>/<MLD>/<Mdl Unit>/Revision History*
The Revision History shall be up to date.
Remark: See UML template

Rule 5.3: *Unit Level Design of a module shall be split in the following packages:*
- Static View: contains the static diagrams of the module
- Behavioral View: contains the dynamic diagrams of the module

Generic Unit Level Design Rules 2/4

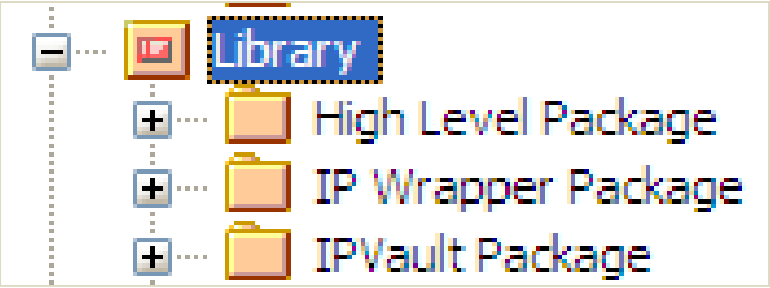
Rule 6.1: *The sMCAL unit will be designed as a UML component having the stereotype "unit"; the unit name is defined as follows:*

***- <MDL>_HLD** for High Level Driver layer*

***-<MDL>_IPW** for IP Wrapper layer*

***- <IP_VAULT>** for IP_VAULT layer*

Remark: See the UML Template



Library

High Level Package

IP Wrapper Package

IPVault Package

Generic Unit Level Design Rules 3/4

Rule 6.2: *A package named "**File Artifacts**" will contain the artifacts elements corresponding to the files used for deploying the current unit*
Remark: This artifact elements are used in Files Structure diagram

Rule 6.3: *A dedicated package (named **“Data Types**") shall contain unit specific types definition. Additional sub-packages for splitting data types in public, local or configuration data types can be added if relevant.*
Remark: See the UML Template

Rule 6.4: *A dedicated package (named "**Data-stores**") shall contain unit specific global data, designed as data-stores.*
Remark: See the UML Template

Rule 6.5:Unit specific configuration **Data Types** shall be documented in the static view package of the concerned unit.*
Remark: See the UML Template

Generic Unit Level Design Rules 4/4

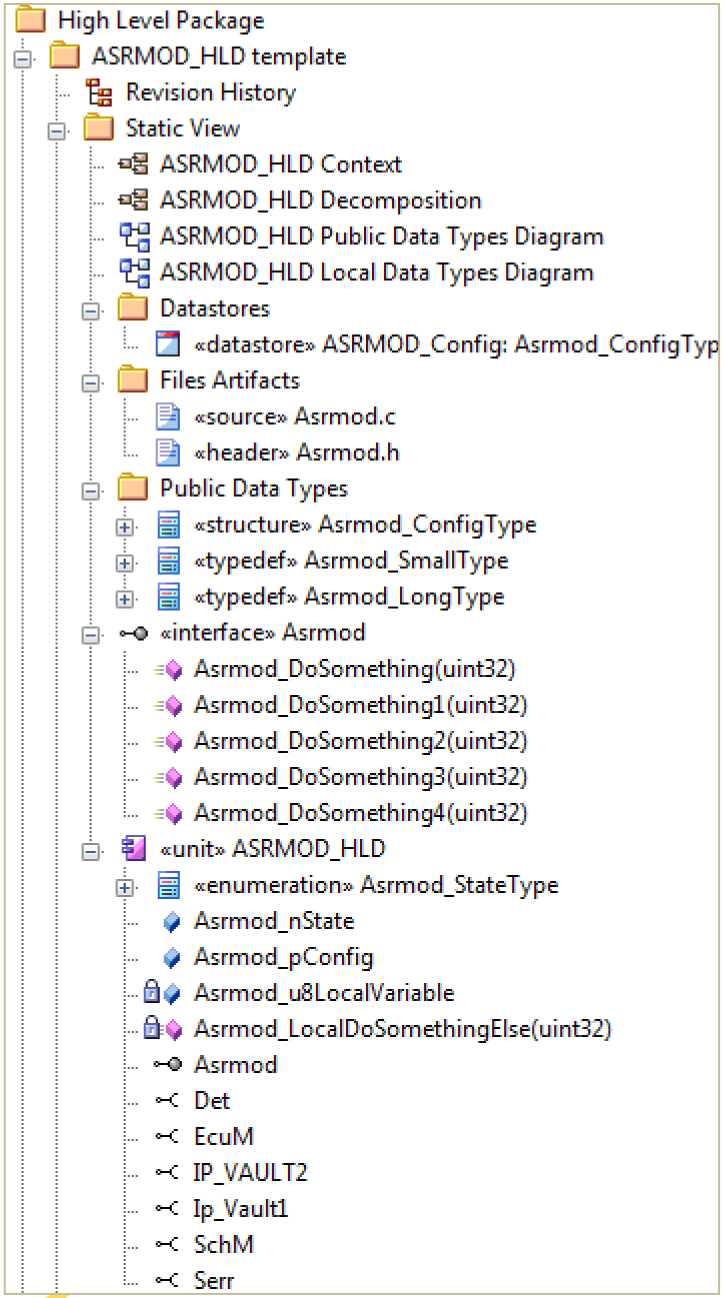
Rule 6.6: *The definition of the public functions exported by the unit shall de designed in a dedicated UML interface object; (to be defined if it is embedded in the unit component or not)*
Remarks: See the UML Template. The interfaces are used in static views of different modules

Rule 6.7: *A component diagram (named "<Unit_Name> Context") shall reflect the interface of the sMCAL unit with the external environment*
Remark: See the UML Template

Rule 6.8: *Dedicated class diagrams shall show in a visual form the unit specific data types. If relevant, data-store instantiations can be shown.*
Remark: splitting in local types, public types, … is module dependent

***Note**: Public data types exported by an unit **shall not** be modeled in interfaces; use “Public Data Types Diagram” for highlighting the exported data types*

HLD Layer Context → Static View



- High Level Package
  - ASRMOD_HLD template
    - Revision History
    - Static View
      - ASRMOD_HLD Context
      - ASRMOD_HLD Decomposition
      - ASRMOD_HLD Public Data Types Diagram
      - ASRMOD_HLD Local Data Types Diagram
      - Datastores
        - «datastore» ASRMOD_Config: Asrmod_ConfigTyp
      - Files Artifacts
        - «source» Asrmod.c
        - «header» Asrmod.h
      - Public Data Types
        - «structure» Asrmod_ConfigType
        - «typedef» Asrmod_SmallType
        - «typedef» Asrmod_LongType
      - «interface» Asrmod
        - Asrmod_DoSomething(uint32)
        - Asrmod_DoSomething1(uint32)
        - Asrmod_DoSomething2(uint32)
        - Asrmod_DoSomething3(uint32)
        - Asrmod_DoSomething4(uint32)
      - «unit» ASRMOD_HLD
        - «enumeration» Asrmod_StateType
        - Asrmod_nState
        - Asrmod_pConfig
        - Asrmod_u8LocalVariable
        - Asrmod_LocalDoSomethingElse(uint32)
        - Asrmod
        - Det
        - EcuM
        - IP_VAULT2
        - Ip_Vault1
        - SchM
        - Serr

Contains:

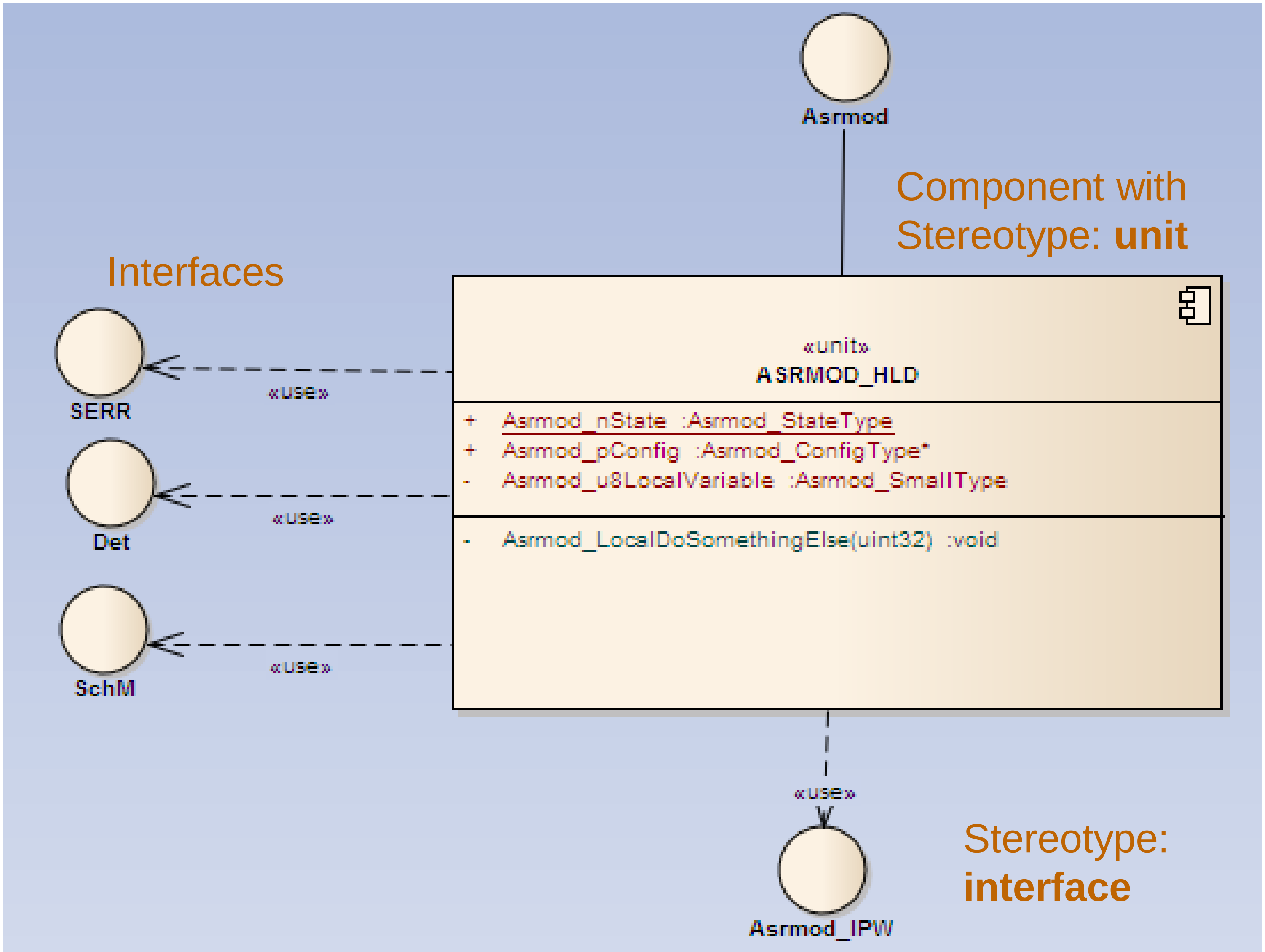
- ASRMOD HLD Context Diagram – HLD interface with external environment (SERR, DEM, SchM)
- ASRMOD HLD Decomposition Diagram – File used for deploying the HLD unit
- ASRMOD HLD Public Data Types Diagram – HLD specific public data types
- ASRMOD HLD Local Data Types Diagram – HLD specific local data types

+

Data-stores
Files Artifacts
Public Data Types
Interfaces
Unit types

HLD Layer Context Diagram

HLD Context Static Diagram
(Component Type Diagram)


Asrmod
«unit»
ASRMOD_HLD
+ Asrmod_nState :Asrmod_StateType
+ Asrmod_pConfig :Asrmod_ConfigType*
- Asrmod_u8LocalVariable :Asrmod_SmallType
- Asrmod_LocalDoSomethingElse(uint32) :void
SERR
«use»
Det
«use»
SchM
«use»
«use»
Asrmod_IPW

Interfaces

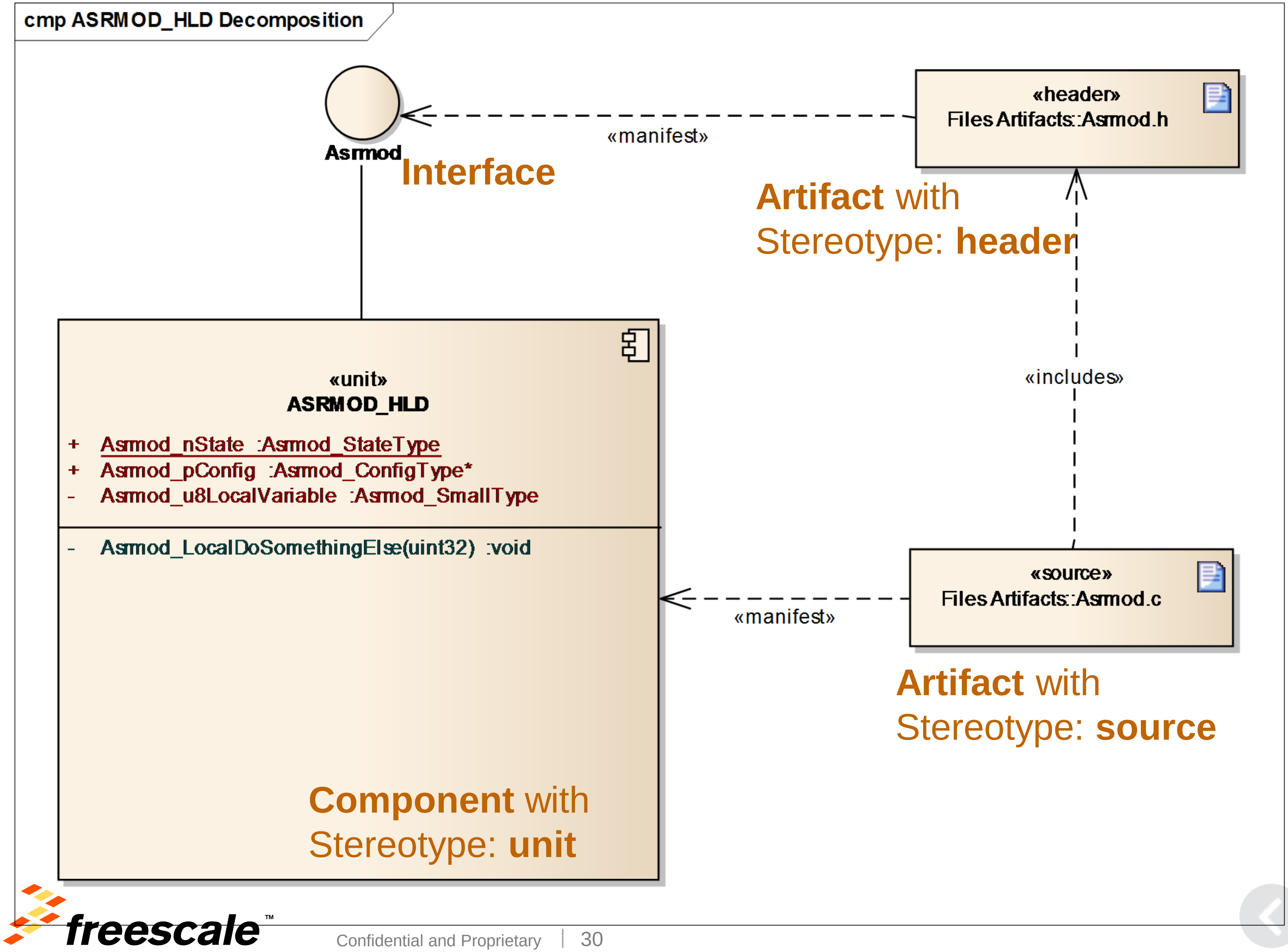
Component with Stereotype: **unit**

Stereotype: **interface**

Shows the module interface with external environment

HLD Deployment Static Diagram
(Component Diagram)

HLD Layer → Deployment Static Diagram


cmp ASRMOD_HLD Decomposition
Asmod
«manifest»
«header»
Files Artifacts::Asmod.h
«includes»
«unit»
ASRMOD_HLD
+ Asmod_nState :Asmod_StateType
+ Asmod_pConfig :Asmod_ConfigType*
- Asmod_u8LocalVariable :Asmod_SmallType
- Asmod_LocalDoSomethingElse(uint32) :void
«manifest»
«source»
Files Artifacts::Asmod.c

Interface

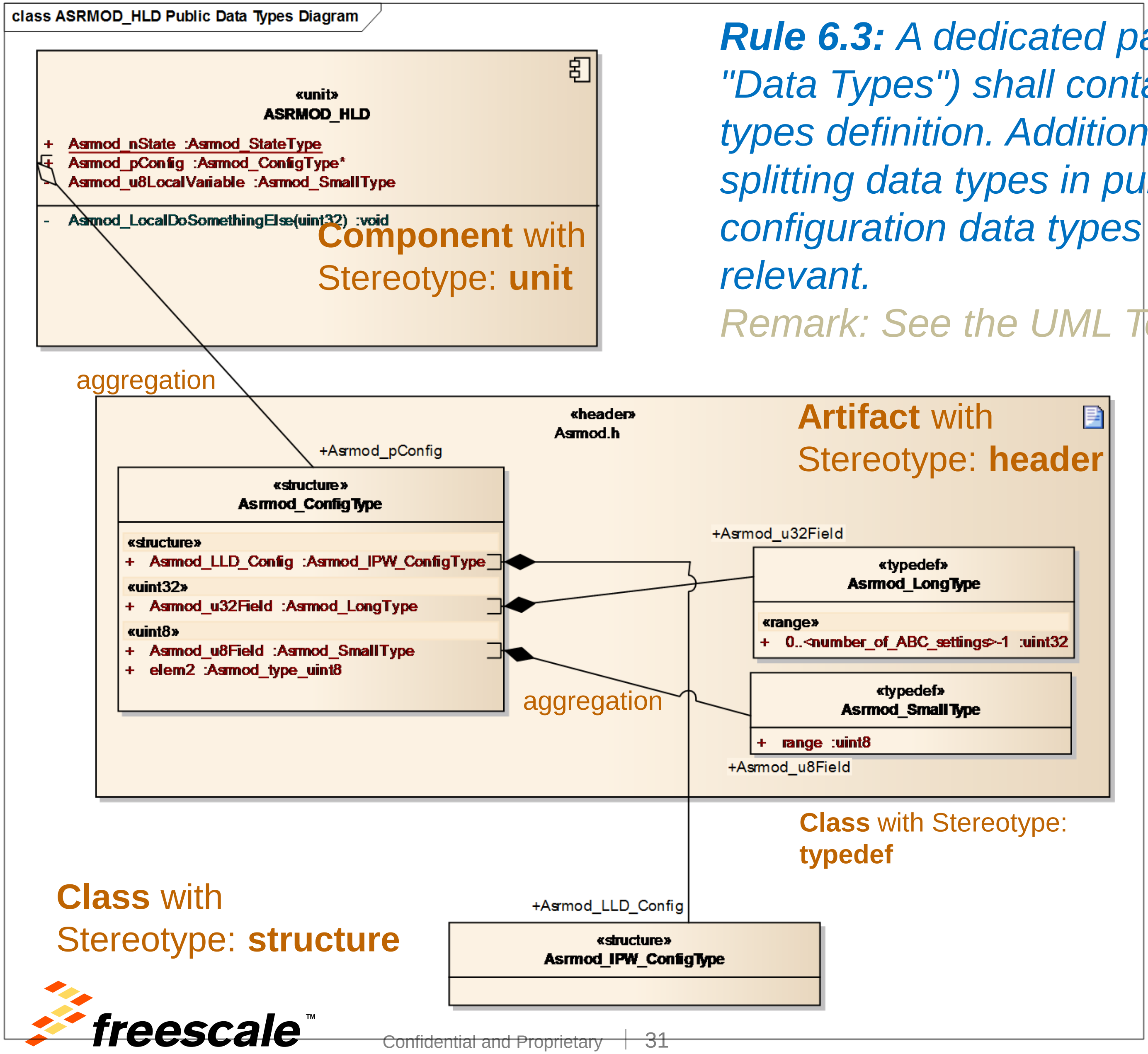
Artifact with
Stereotype: **header**

Artifact with
Stereotype: **source**

Component with
Stereotype: **unit**

Public Data Types Diagram

HLD Static View – Public Data Types


class ASRMOD_HLD Public Data Types Diagram
«unit»
ASRMOD_HLD
+ Asrmod_nState :Asrmod_StateType
+ Asrmod_pConfig :Asrmod_ConfigType*
- Asrmod_u8LocalVariable :Asrmod_SmallType
- Asrmod_LocalDoSomethingElse(uint32) :void
«header»
Asrmod.h
+Asrmod_pConfig
«structure»
Asrmod_ConfigType
«structure»
+ Asrmod_LLD_Config :Asrmod_IPW_ConfigType
«uint32»
+ Asrmod_u32Field :Asrmod_LongType
«uint8»
+ Asrmod_u8Field :Asrmod_SmallType
+ elem2 :Asrmod_type_uint8
+Asrmod_u32Field
«typedef»
Asrmod_LongType
«range»
+ 0..<number_of_ABC_settings>-1 :uint32
«typedef»
Asrmod_SmallType
+ range :uint8
+Asrmod_u8Field
+Asrmod_LLD_Config
«structure»
Asrmod_IPW_ConfigType

Component with Stereotype: **unit**

aggregation

Artifact with Stereotype: **header**

aggregation

Class with Stereotype: **typedef**

Class with Stereotype: **structure**

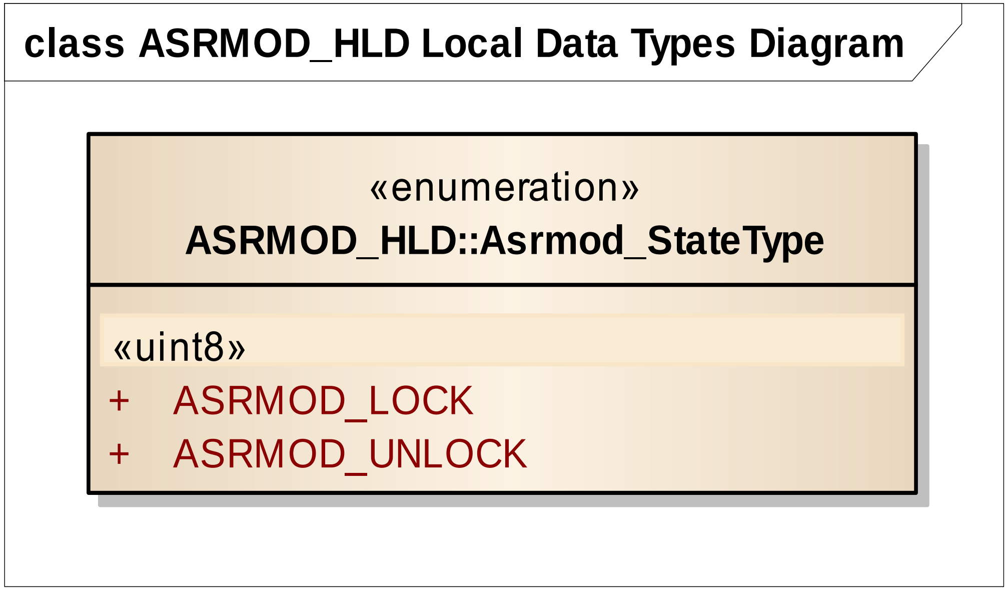
Rule 6.3: *A dedicated package (named "Data Types") shall contain unit specific types definition. Additional subpackages for splitting data types in public, local or configuration data types can be added if relevant.*

Remark: See the UML Template

Class *Type Diagram*

HLD Static View – Local Data Types

Local Data Types Diagram



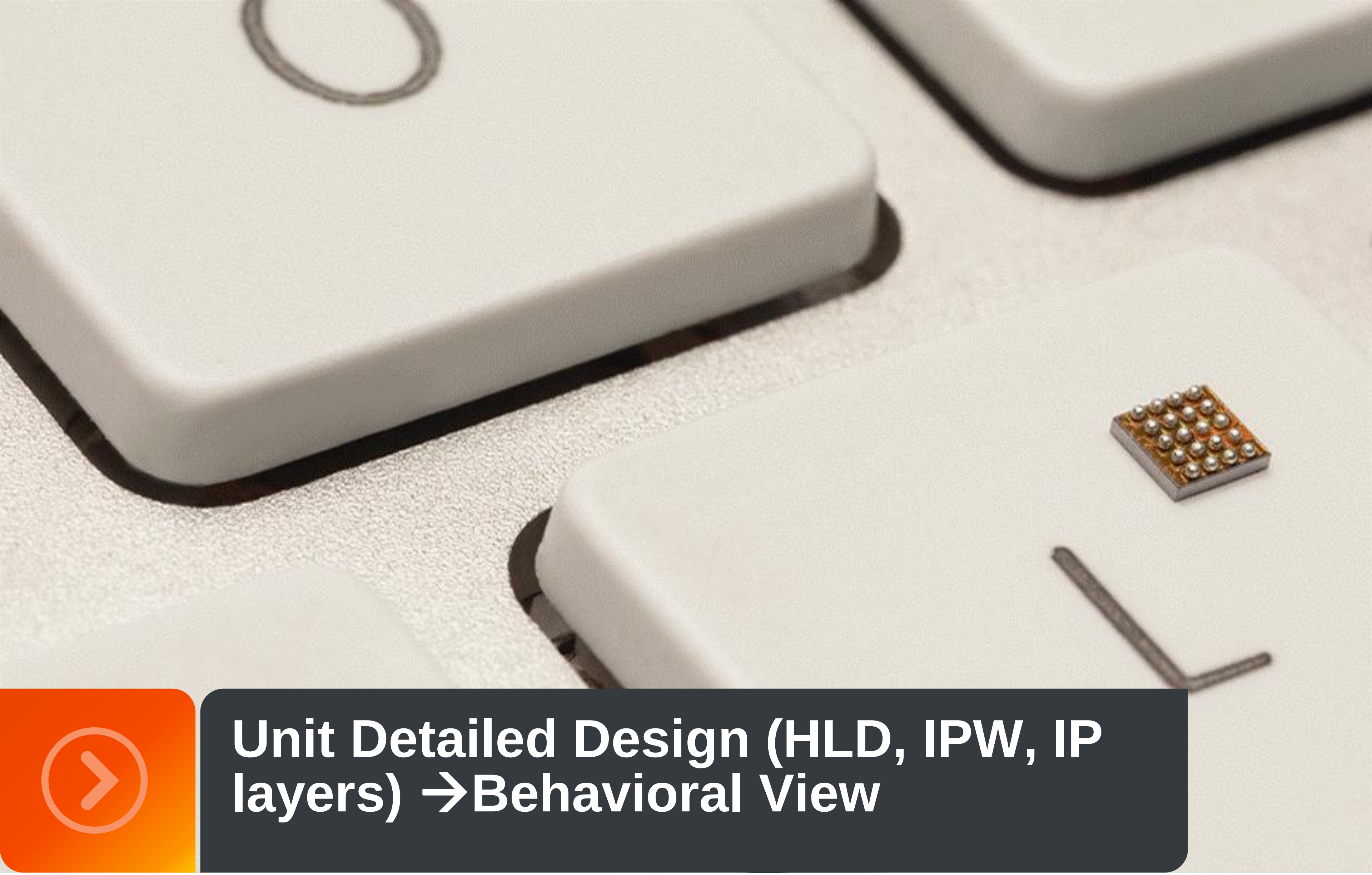
class ASRMOD_HLD Local Data Types Diagram

«enumeration»
ASRMOD_HLD::Asrmod_StateType

«uint8»
+ ASRMOD_LOCK
+ ASRMOD_UNLOCK

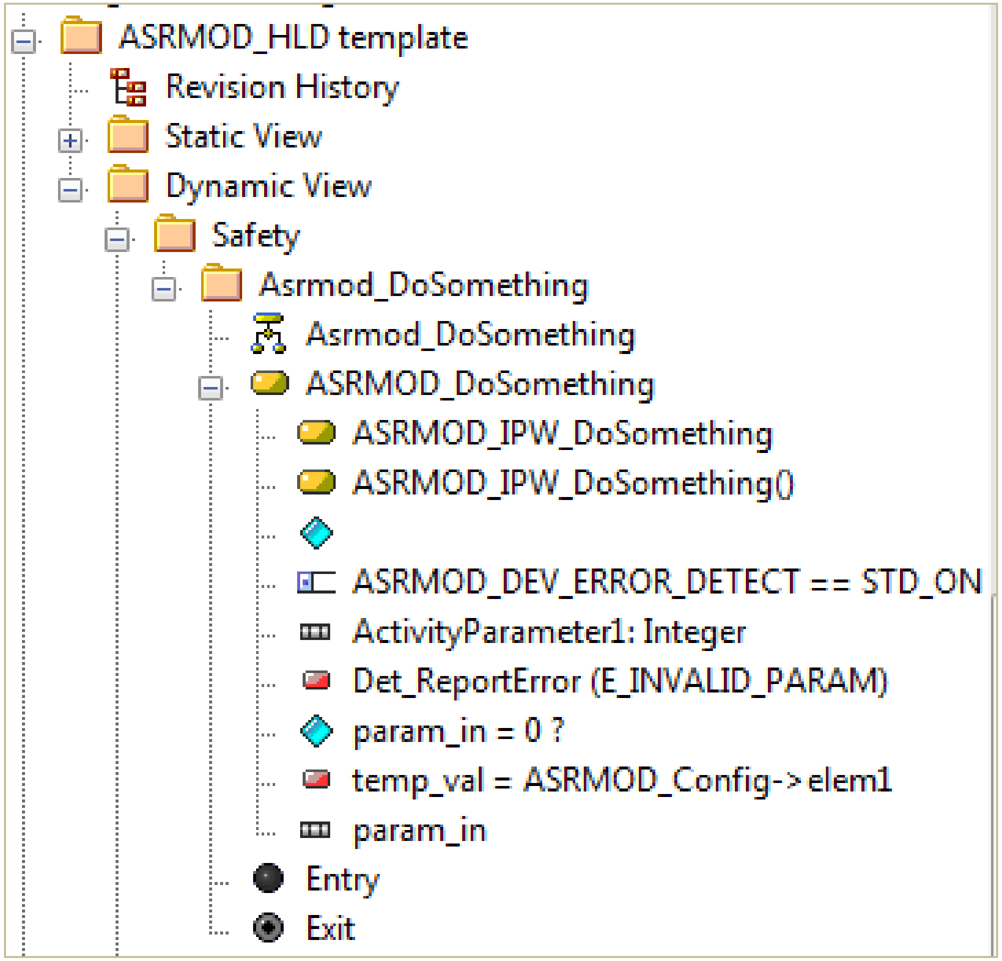
Class *Type Diagram*

Rule 6.8 *Dedicated class diagrams shall show in a visual form the unit specific data types.*
If relevant, data-store instantiations can be shown.
Remark: splitting in local types, public types, … is module dependent



Unit Detailed Design (HLD, IPW, IP layers) →Behavioral View

HLD Dynamic View Content



ASRMOD_HLD template
Revision History
Static View
Dynamic View
Safety
Asrmod_DoSomething
Asrmod_DoSomething
ASRMOD_DoSomething
ASRMOD_IPW_DoSomething
ASRMOD_IPW_DoSomething()
ASRMOD_DEV_ERROR_DETECT == STD_ON
ActivityParameter1: Integer
Det_ReportError (E_INVALID_PARAM)
param_in = 0 ?
temp_val = ASRMOD_Config->elem1
param_in
Entry
Exit

- Each function exported by a unit shall be modeled in this way;

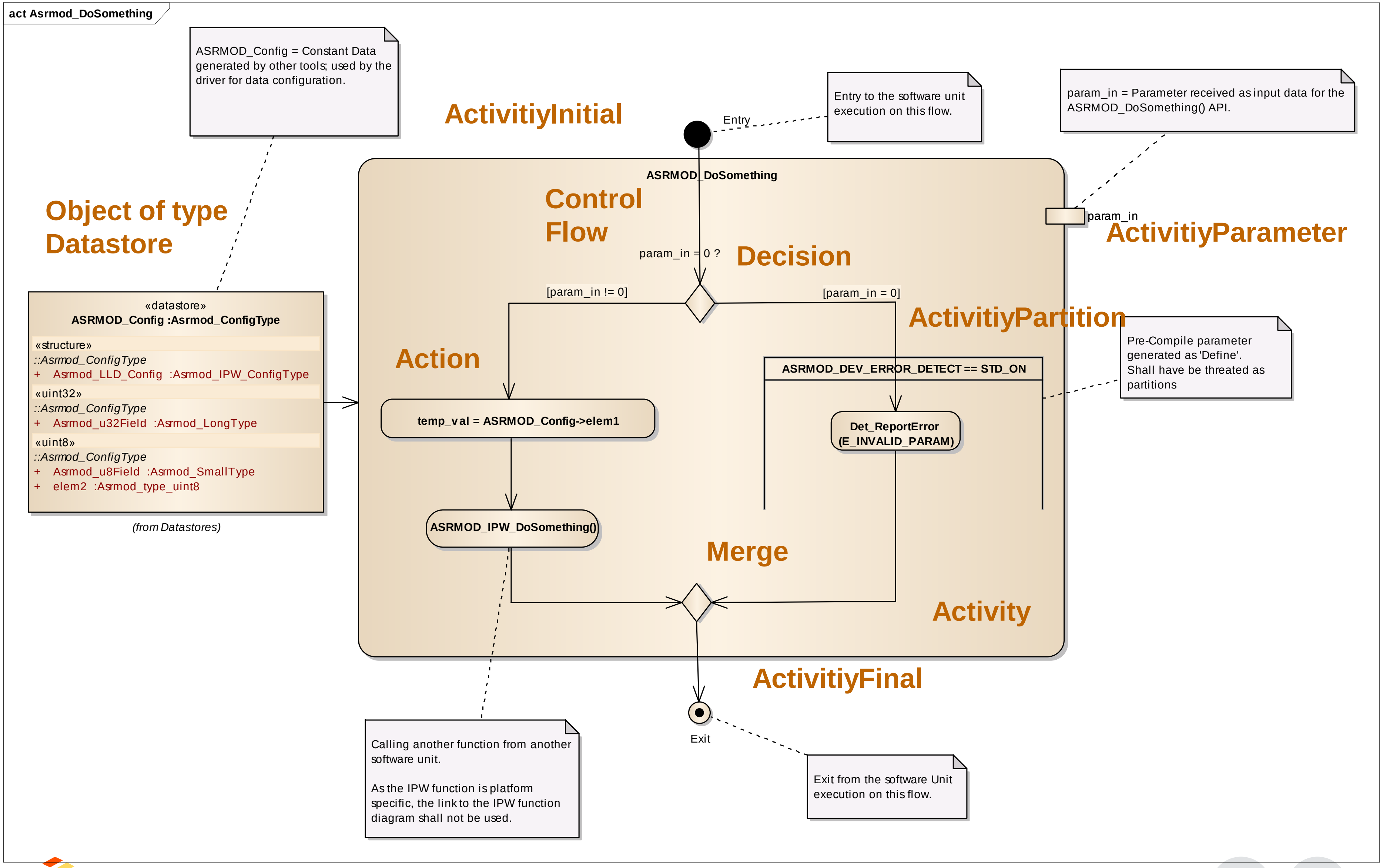
- We will not create detailed activity diagrams for IP functions, will have one block per function containing a brief description;

Rule 7.1:

For each public unit function a package having the function name shall be defined; this package shall contain activity diagrams showing the logic of the given functionality.

Activity Diagram

Example of ASRMOD HLD Function



ASRMOD_Config = Constant Data generated by other tools; used by the driver for data configuration.

Entry to the software unit execution on this flow.

param_in = Parameter received as input data for the ASRMOD_DoSomething() API.

«datastore»
ASRMOD_Config :Asrmod_ConfigType

«structure»
::Asrmod_ConfigType
+ Asrmod_LLD_Config :Asrmod_IPW_ConfigType

«uint32»
::Asrmod_ConfigType
+ Asrmod_u32Field :Asrmod_LongType

«uint8»
::Asrmod_ConfigType
+ Asrmod_u8Field :Asrmod_SmallType
+ elem2 :Asrmod_type_uint8

(from Datastores)

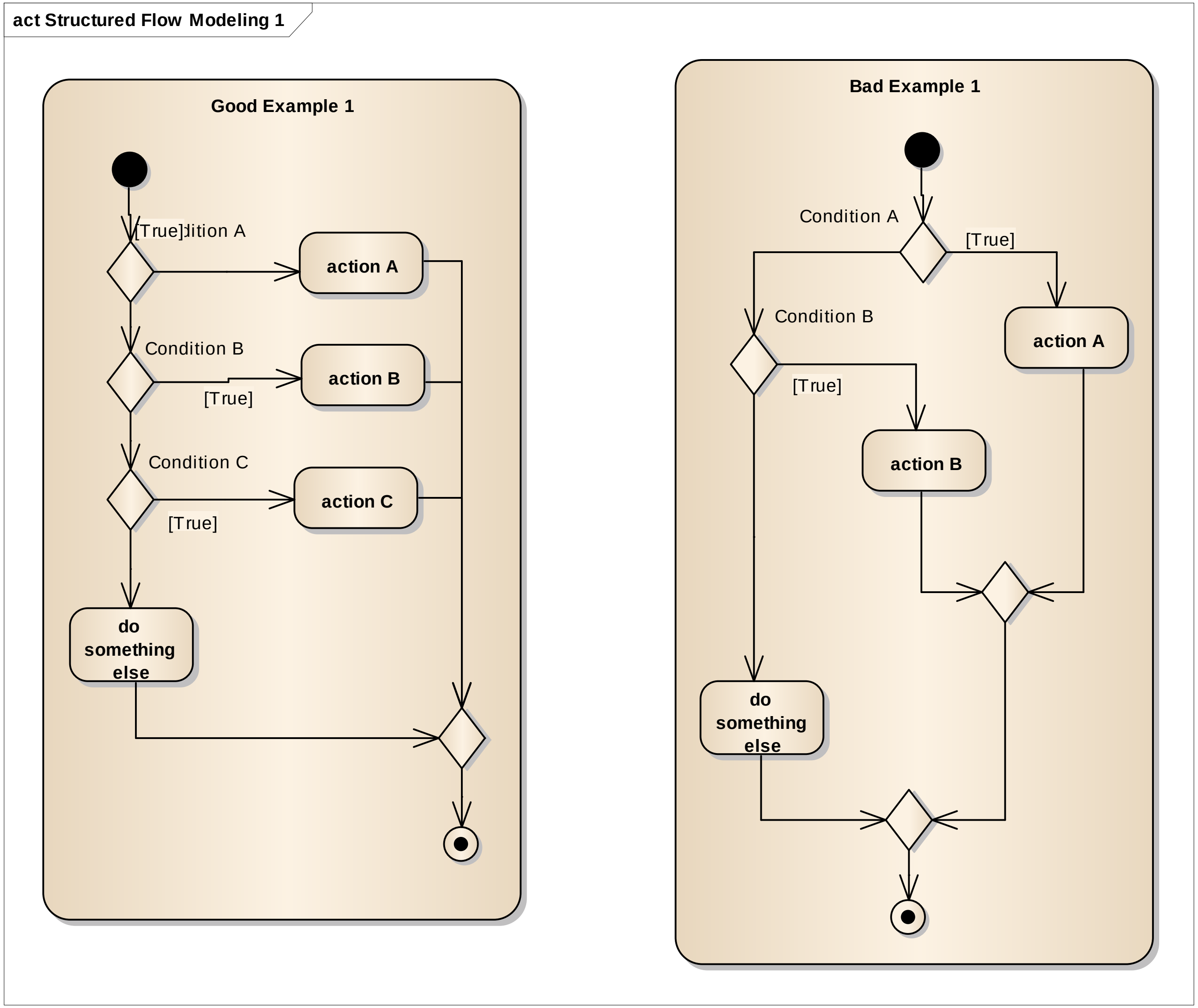
Pre-Compile parameter generated as 'Define'. Shall have be threated as partitions

Calling another function from another software unit.

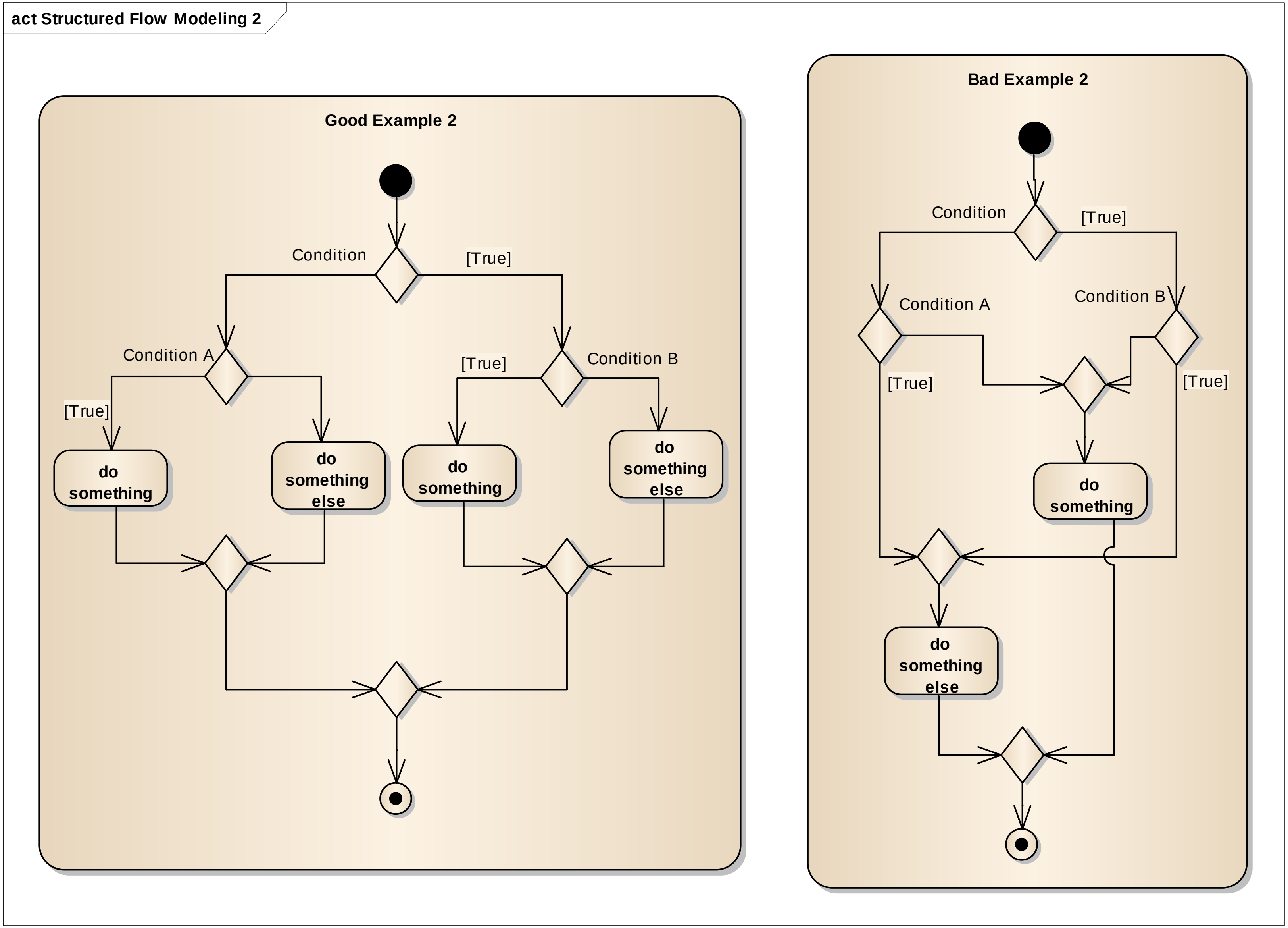
As the IPW function is platform specific, the link to the IPW function diagram shall not be used.

Exit from the software Unit execution on this flow.

Structured Control Flow in Activity Diagrams (1/2)



Structured Control Flow in Activity Diagrams (2/2)



Activity Diagrams Rules

Rule 7.2:

An activity diagram shall reflect the logic of the function, which might not be identically reflected at code level.

Remark: it shall reflect the code (broadly), at IP level the activity shall be a brief description of the function

Rule 7.3:

In an activity diagram, the analyzed function will be designed as an activity element embedding all the other detailed items.

Rule 7.4:

Function parameters and returned values shall be designed as parameters of the outer activity item.

Remark: See the UML Template

Rule 7.5

Internal actions deployed in an activity diagram shall be designed as "action" elements.

Remark: See the UML Template

Activity Diagrams Rules

Rule 7.6:

Invocations of external functions shall be designed as "activity" elements and function parameters shall be mentioned between brackets.

Remarks: Activity Linkage shall be carefully used, as their might be more than one activity diagrams per function.

Rule 7.7:

Global resources (global memory/HW registers) accessed from a function shall be represented as data-stores in the function activity diagrams; these data-stores can be:

- instances already defined in static views packages of the same units

- peripheral registers (only for IP units).

Remark: abstract design shall be used. Ex: enable the counter -instead of showing exactly what register is written

Rule 7.8:

Data-stores shall be shown outside of the function top level activity item.

Remark: data-stores internal to function activity corresponds to local variables - we do not design local variables

Activity Diagrams Rules

Rule 7.9:

A data flow relationship shall be drawn between a data-store and the activity corresponding to the entire function.

Remarks: even if standard UML, it helps in identifying functions accessing the same data-store.

Rule 7.10

Different functional behavior due to safety measures or pre-compile configurations can be shown:

- in different activity diagrams (increased readability)

- only one activity diagram, where the different safety measures or pre-compile options are reflected by activity partitions. (accepted for efficiency purposes)

Remark: A single activity diagram can be used only if the diagram readability is preserved.

Rule 7.11

Activity partitions are labeled as follows:

- predefined pre-compile option or safety measure name

-logical expression of pre-compile options and safety measure names.

Rule 7.12

Branch merges shall be shown as merging diamonds, and shall be designed in a structured way (each decision node shall have a corresponding merge node).

Activity Diagrams Rules

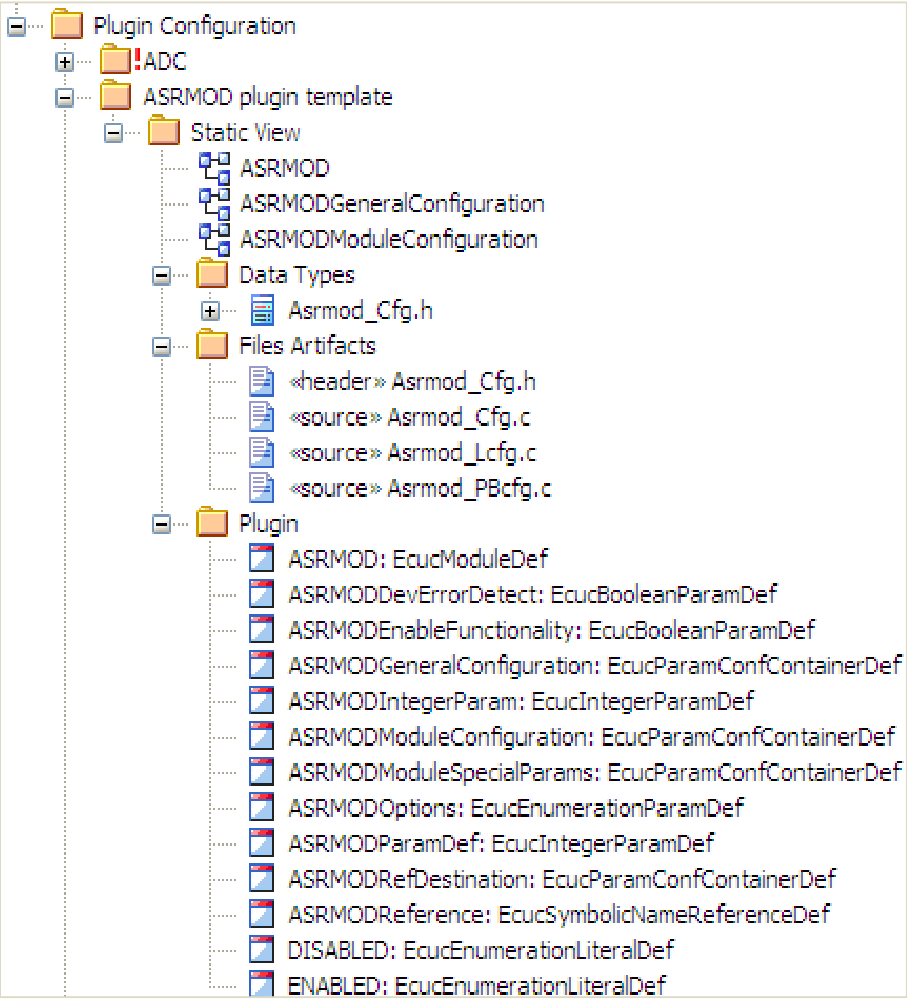
Rule 7.13

State diagrams shall be added in the <Driver>_Hld/Behavioral View.

Note: *Applicable only for HLD level.*

Plugin Configuration

Plugin Static View Structure



- Plugin Configuration
  - !ADC
  - ASRMOD plugin template
    - Static View
      - ASRMOD
      - ASRMODGeneralConfiguration
      - ASRMODModuleConfiguration
    - Data Types
      - Asrmod_Cfg.h
    - Files Artifacts
      - «header» Asrmod_Cfg.h
      - «source» Asrmod_Cfg.c
      - «source» Asrmod_Lcfg.c
      - «source» Asrmod_PBcfg.c
    - Plugin
      - ASRMOD: EcucModuleDef
      - ASRMODDevErrorDetect: EcucBooleanParamDef
      - ASRMODEnableFunctionality: EcucBooleanParamDef
      - ASRMODGeneralConfiguration: EcucParamConfContainerDef
      - ASRMODIntegerParam: EcucIntegerParamDef
      - ASRMODModuleConfiguration: EcucParamConfContainerDef
      - ASRMODModuleSpecialParams: EcucParamConfContainerDef
      - ASRMODOptions: EcucEnumerationParamDef
      - ASRMODParamDef: EcucIntegerParamDef
      - ASRMODRefDestination: EcucParamConfContainerDef
      - ASRMODReference: EcucSymbolicNameReferenceDef
      - DISABLED: EcucEnumerationLiteralDef
      - ENABLED: EcucEnumerationLiteralDef

Static View Content:

➢**ASRMOD** Diagram – top level plugin structure: contains top level containers and subcontainers

➢Dedicated Packages and diagrams are defined for each container to detail its decomposition in subcontainers and the parameter definition

Plugin Rules

Rule 8.1:

Informations about the tresos plugin configuration structure are defined in a dedicated package named "Plugin Configuration" for each sMCAL module.

Rule 8.2:

Configuration fields shall be defined as ObjectInstance elements, using AUTOSAR provided template entities

Note: *At least configuration fields requested by Autosar or Internal requirements shall be designed*

Rule 8.3:

Configuration generated defines shall de defined at plugin level in a dedicated package named "Data Types“

Naming Convention Rules

Naming Convention Rules

Rule 12.1:

File names shall represent the content or role of the file and shall follow this format:

<Msn>_<Name><.Ext>

where:

<Msn> *Module Short Name*

<Name> *File Descriptive Name (i.e.: Irq, Cfg)*

<.Ext> *File Extension*

Rule 12.2:

Data types shall respect the following naming convention:

<Msn>_<TypeName>Type

Where ***<TypeName>*** *shall follow the so called CamelCase convention (first letter of each word is uppercase, consequent letters are lowercase).*

Rule 12.3:

Global variables shall respect the following naming convention:

<Msn>_<VariableName>

Where ***<VariableName>*** *shall follow the CamelCase convention.*

Naming Convention Rules

Rule 12.4:

The APIs shall respect the following naming convention:
<Msn>_<ApiName>()
Where **<ApiName>** shall follow the CamelCase convention.

Rule 12.5:

Internal module function shall respect the following naming convention:
- **HLD function**: <Msn>_Function() (ex: Wdg_Init())
- **IPW function**: <Msn>_Ipw_<Function>() (ex: Wdg_Ipw_Init(),Icu_Ipw_StartSignalMeasurement())
- **IP function**: <Msn>_<Ip>_<Function>()(ex:Wdg_Swt_Init(),Icu_Etimer_StartSignalMeasurement())

Where first letter of each word should be in upper case and consecutive letters in lower case (see BSW00310 [3]).
Exception: For module shared functions (like eTimer functions) the format shall be <Ip>_<Function>().

Naming Convention Rules

Rule 12.6:

The global variables shall use following naming convention

<Msn>_[<Ip>_][<PrefixType>]<VarName>

where <PrefixType> might be:

- ***p**: pointer to data*
- ***pf**: pointer to a function*
- ***u** (for unsigned), s (for signed): integers for which size is not relevant or is unknown (usually HLD related data)*
- ***e**: enum variables*
- ***u8, u16, u32, s8, s16, s32**: sized integers, for which size is relevant (usually for the register values)*
- ***b**: Boolean variables to be compared against TRUE/FALSE*
- ***a**: arrays*
- *nothing: for anything else*

Generic Rules

Style Rules

Rule 9.1:

Diagrams shall be simple, clear and readable

Rule 9.2:

The level of details of the diagrams shall be driven by the need of understanding how addressed requirements are implemented by SW

Cross Checks

Rule 10.1:
There is consistency between:
- function definition at interface level
- function parameters described in activity diagrams

Generic Architectural Rules

Rule 11.1:

In a Files Structure diagram, the including dependencies between units shall follow the layering approach:
- no cyclic inclusions
- include dependencies shall be only to the layers bellow or above (avoid skipping layers)
-specific configuration header files for each layer/unit

Rule 11.2:

A function shall access data-stores in its own unit; if access to data-stores from other units is needed, it shall be done through dedicated functions publisher by the owning unit; the only exception is peripheral related data-stores accessed from IP related unit functions.

Generic Architectural Rules

Rule 11.3:

HLD Layer behavior shall be independent of the HW IP logic.

***Note:** It shall implement mainly the AUTOSAR related logic*

Rule 11.4:

IPW Layer behavior shall be only the wrapping between HLD abstraction and specific IP interface. No module dependent logic shall be deployed at IPW level.

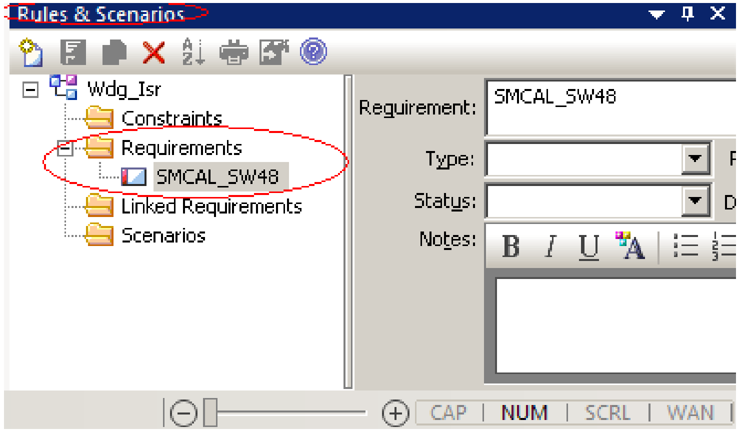
Rule 11.5:

IP Layer behavior shall be made independent of HLD logic. Implementing AUTOSAR related behavior at IP level should be avoided.

Req-Design Mapping and ReqTracer

How to add requirements in the design

- The rules of defining the requirements are specific for each document:
  - For the **requirements documents** – every entry in the table that is not marked as N/A, Explanation or Heading is a requirement
  - For the **design document** the requirements that are fulfilled by a design element shall be added under the Requirements tab



Requirement: SMCAL_SW48

Tip and trick: to make the Rules&Scenarios window more accessible the option >More Windows>Rules and Scenarios can be used

- The elements that may have requirements (in this current version) are: activity diagrams, interfaces, classes, actions, objects, components

Review Checklist

Brief Refresh …

Follow the Peer Review Process for Walkthrough/Mini-Walkthrough

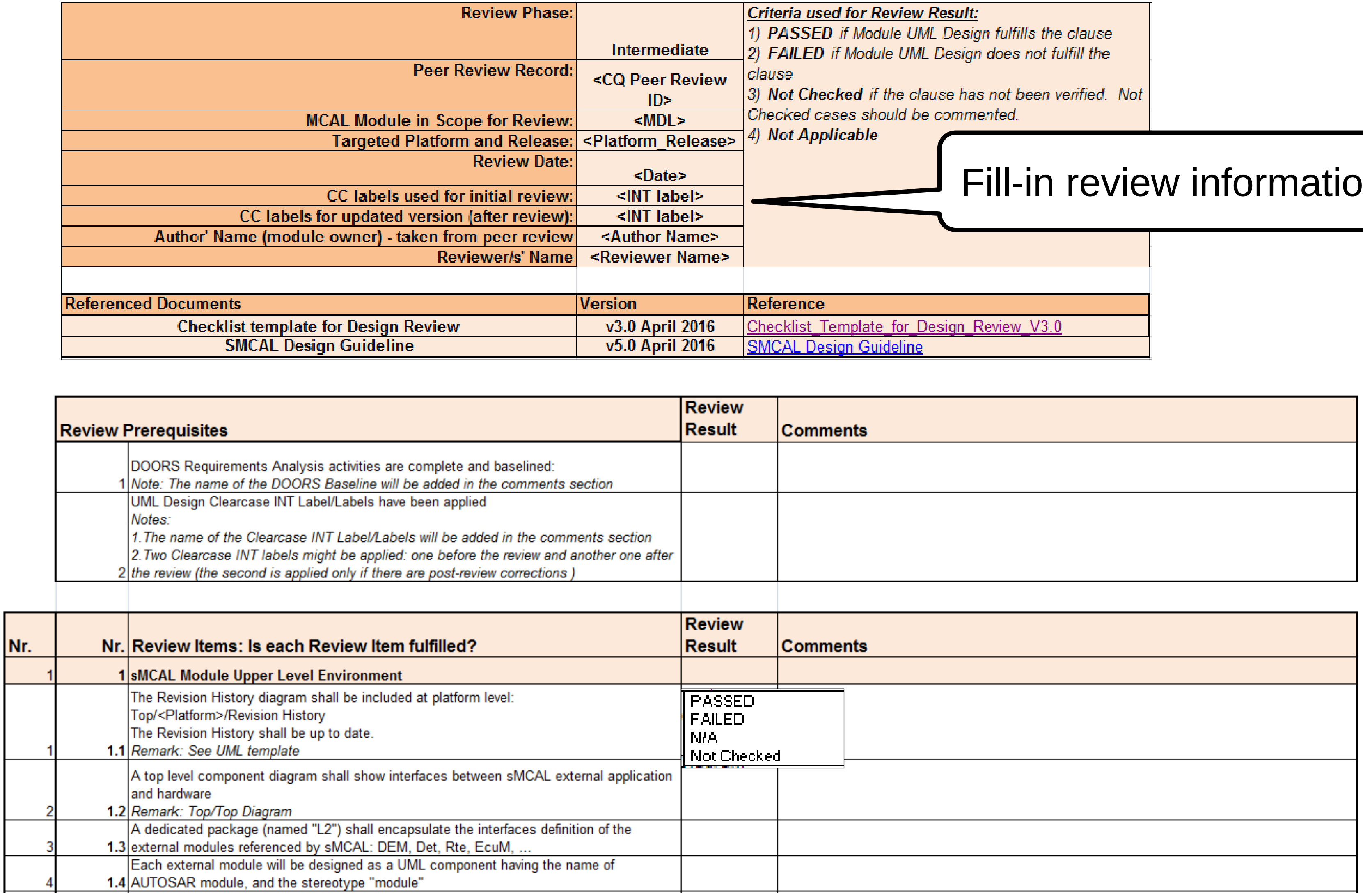
➢Reviewer has to review the work-product from finding faults perspective prior to the review meeting and send the review findings to the Author in advance, via email

➢ Reviewer has to review the work-products along with the Review Checklist, has to fill-out the checklist prior to the meeting

➢During the review meeting, the findings are discussed, the checklist is updated – version ‘Intermediate’ saved in CQ as an attachment to Review Record

➢After the issues have been corrected and the Reviewer/Moderator has done the final check and if everything is ok → review is closed with Accepted, the checklist is having the version “Final” saved to the CQ review record

Review Checklist



Review Phase:	Intermediate	***Criteria used for Review Result:*** 1) ***PASSED*** *if Module UML Design fulfills the clause* 2) ***FAILED*** *if Module UML Design does not fulfill the clause* 3) ***Not Checked*** *if the clause has not been verified. Not Checked cases should be commented.* 4) ***Not Applicable***
Peer Review Record:	<CQ Peer Review ID>	
MCAL Module in Scope for Review:	<MDL>	
Targeted Platform and Release:	<Platform_Release>	
Review Date:	<Date>	
CC labels used for initial review:	<INT label>	
CC labels for updated version (after review):	<INT label>	
Author' Name (module owner) - taken from peer review	<Author Name>	
Reviewer/s' Name	<Reviewer Name>	

Referenced Documents	Version	Reference
Checklist template for Design Review	v3.0 April 2016	Checklist_Template_for_Design_Review_V3.0
SMCAL Design Guideline	v5.0 April 2016	SMCAL Design Guideline

Review Prerequisites		Review Result	Comments
1	DOORS Requirements Analysis activities are complete and baselined: *Note: The name of the DOORS Baseline will be added in the comments section*		
2	UML Design Clearcase INT Label/Labels have been applied *Notes:* *1.The name of the Clearcase INT Label/Labels will be added in the comments section* *2.Two Clearcase INT labels might be applied: one before the review and another one after the review (the second is applied only if there are post-review corrections)*		

Nr.	Nr.	Review Items: Is each Review Item fulfilled?	Review Result	Comments
1	1	**sMCAL Module Upper Level Environment**		
1	1.1	The Revision History diagram shall be included at platform level: Top/<Platform>/Revision History The Revision History shall be up to date. *Remark: See UML template*	PASSED FAILED N/A Not Checked	
2	1.2	A top level component diagram shall show interfaces between sMCAL external application and hardware *Remark: Top/Top Diagram*		
3	1.3	A dedicated package (named "L2") shall encapsulate the interfaces definition of the external modules referenced by sMCAL: DEM, Det, Rte, EcuM, ...		
4	1.4	Each external module will be designed as a UML component having the name of AUTOSAR module, and the stereotype "module"		

Fill-in review information

Backup

Part 2: Generic UML Language Structures

- Structural Diagrams Elements and Connectors
- Behavioral Diagrams Elements and Connectors

Diagrams

1. UML **Structural** Diagrams
   - Component Diagrams
   - Package Diagrams
   - Class Diagrams

2. UML **Behavior** Diagrams
   - Activity Diagrams
   - State Machine Diagrams

Component Diagrams Elements and Connectors

- **Component Diagrams** are building blocks used to illustrate pieces of software, modules, and such make up a system. Shows also their organization and interfaces.

Note: As smaller Components come together to create bigger Components, the eventual system can be modeled, building-block style, in Component diagrams

- **E**

Elements	Connectors
Components	Assembly
Interfaces	Delegate
Expose Interfaces (Provided, Required)	Associate
Artifacts	Realize
Boundary (System)	Dependency
	Use

Examples of Elements and Connectors

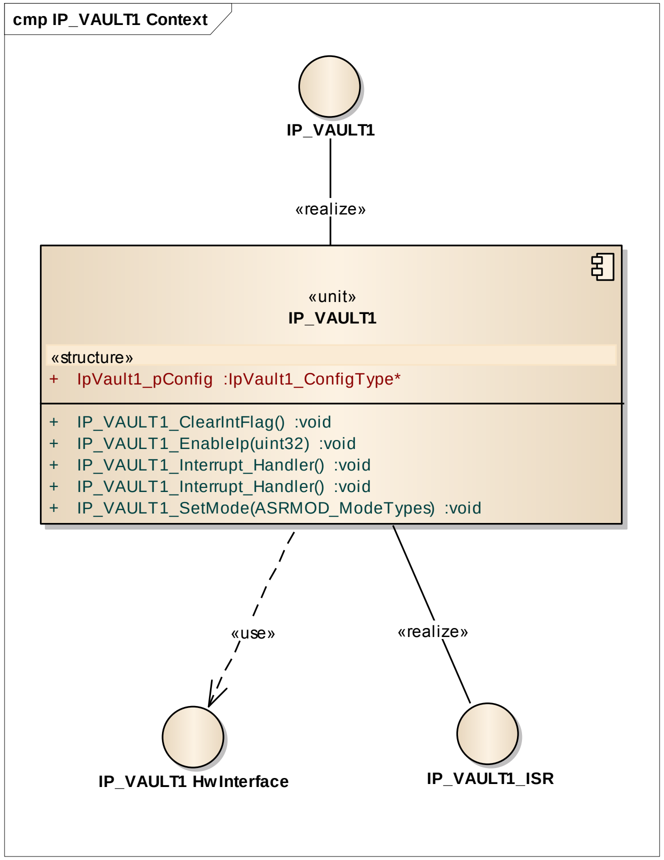
- **Elements of type:**
  - ✓ **Components:** ASR Modules, HLDs, IPVault, IPW, L2 Components, SMCAL,
  - ✓ **Artifacts:** .c/.h files
  - ✓ **Boundary:** used to group the files part of the same layer
  - ✓ **Interfaces:** used to show the connection path between layers/modules (used/realized interfaces)
  - ✓ **Expose Interfaces:**
    - ❖ Required: IPV Drivers require access to HW registers
    - ❖ Provided: IPV Drivers provide Interrupt Service Routines
  - ✓ **Devices:** used to model the HW Peripherals

- **Connectors of type:**
  - ✓ **Delegate**: link the expose interfaces
  - ✓ **Associate**: ???
  - ✓ **Assembly**: show the connection between units
  - ✓ **Realization → <realize>**: between ASR/Stub Modules and their interfaces
  - ✓ **Dependency → <includes> –** eg: files includes
  - ✓ **Dependency → <use>** – eg: ASR Modules are using the HW Interfaces
  - ✓ **Dependency → <optional>** – eg: between ASRMOD and stubs
  - ✓ **Dependency → <manifest>** – e.g.: between a source file (artifact) and a component of type unit
  - ✓ **Dependency → <declares>** – e.g.: between a generic interface (IPVault) and a IPV.h file

Component Diagrams SMCAL Examples

- ✓ASRMOD System Context
- ✓ASRMOD Module Components
- ✓ASRMOD Files Structure
- ✓ASRMOD HLD Context and Decomposition
- ✓ASRMOD IPW Context and Decomposition
- ✓IPVAULT Context and Decomposition
- ✓(HW Components)

e.g. IPVAULT Context


cmp IP_VAULT1 Context
IP_VAULT1
«realize»
«unit»
IP_VAULT1
«structure»
+ IpVault1_pConfig :IpVault1_ConfigType*
+ IP_VAULT1_ClearIntFlag() :void
+ IP_VAULT1_EnableIp(uint32) :void
+ IP_VAULT1_Interrupt_Handler() :void
+ IP_VAULT1_Interrupt_Handler() :void
+ IP_VAULT1_SetMode(ASRMOD_ModeTypes) :void
«use»
«realize»
IP_VAULT1 HwInterface
IP_VAULT1_ISR

Package Diagrams Elements and Connectors

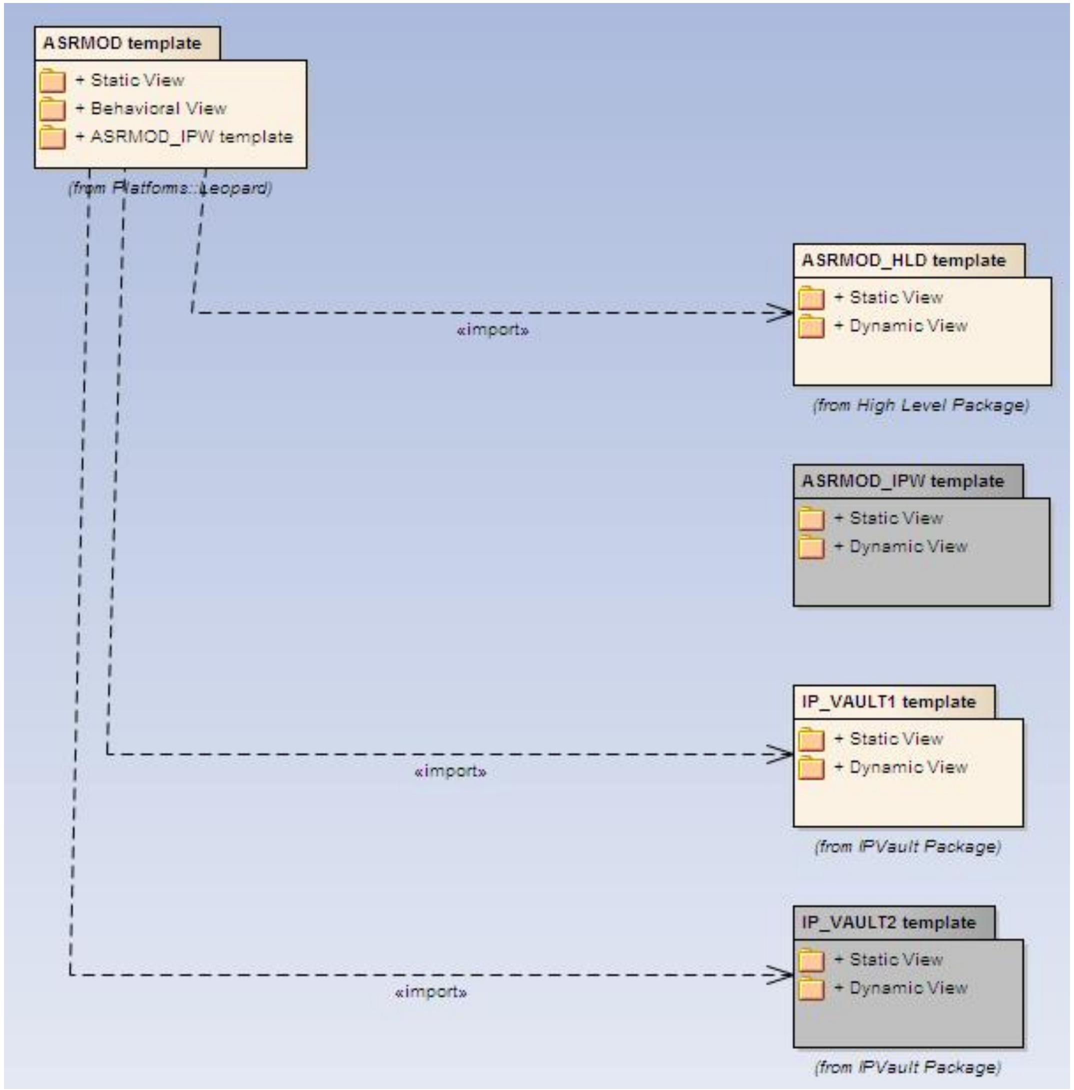
- **Package Diagrams** are used to:
  - ✓ Show <import> dependency between packages (ASRMOD →HLD, ASRMOD →IPV, ASRMOD → Plugin)
- **Elements and Connectors used in Package Diagrams**

Elements	Connectors
Packages	Package Import

Package Diagrams SMCAL Examples

- ✓ASRMOD Package Diagram

e.g. ASRMOD package template



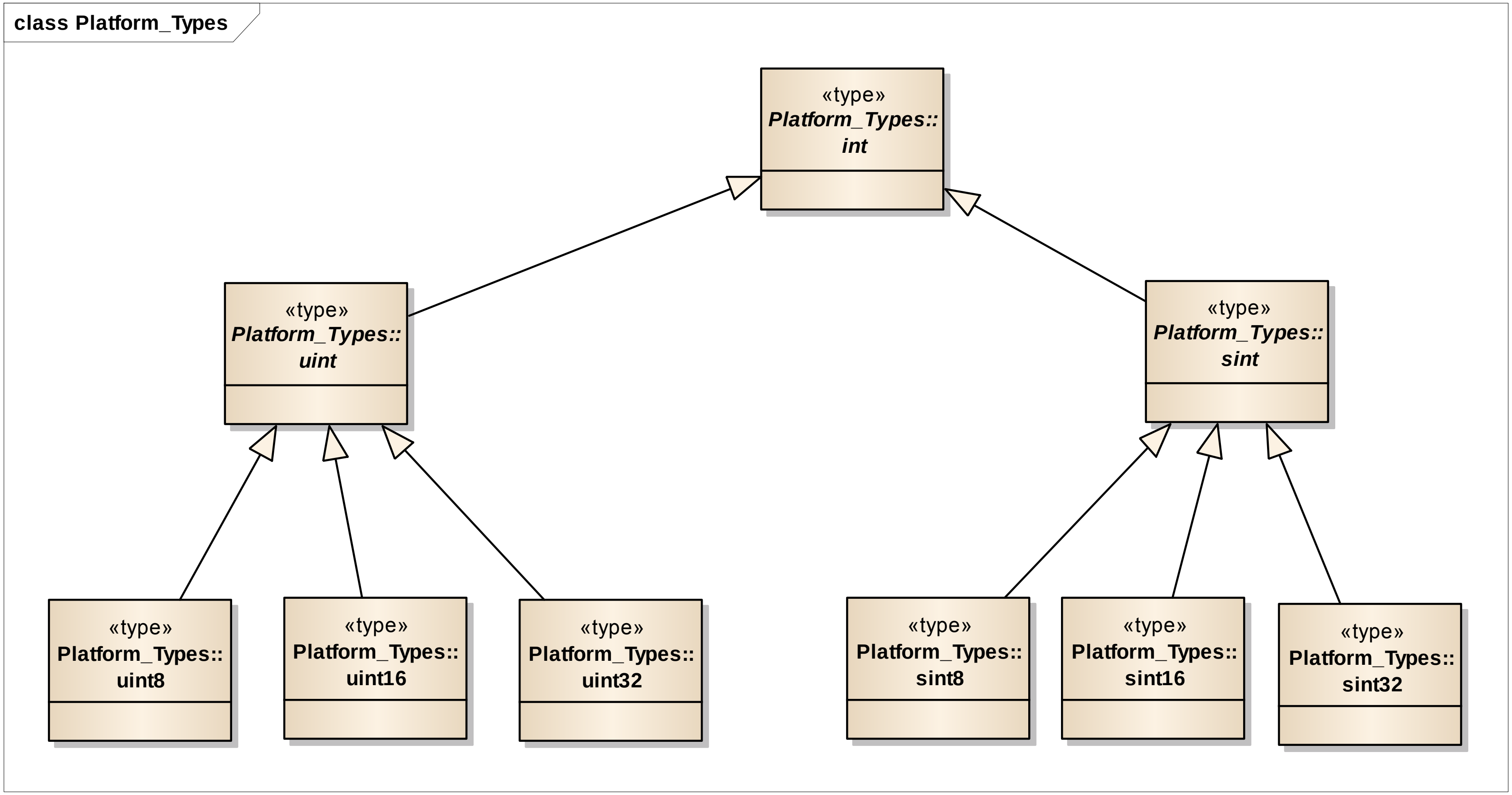
Class Diagrams Elements and Connectors

- **Class Diagrams** are used to:
  - ✓Model the ASR types (e.g. Platform Types)
  - ✓Inside Units (HLD, IPW, IPVs) to model the public and local data types
  - ✓Model the Plugin (elements of type ‘object’)
- **Elements and Connectors used in Class Diagrams**

Elements	Connectors
Classes	Generalization
Interfaces	Dependency
Components	Aggregation
Artifacts	Compose ??
Objects	
Expose Interfaces	

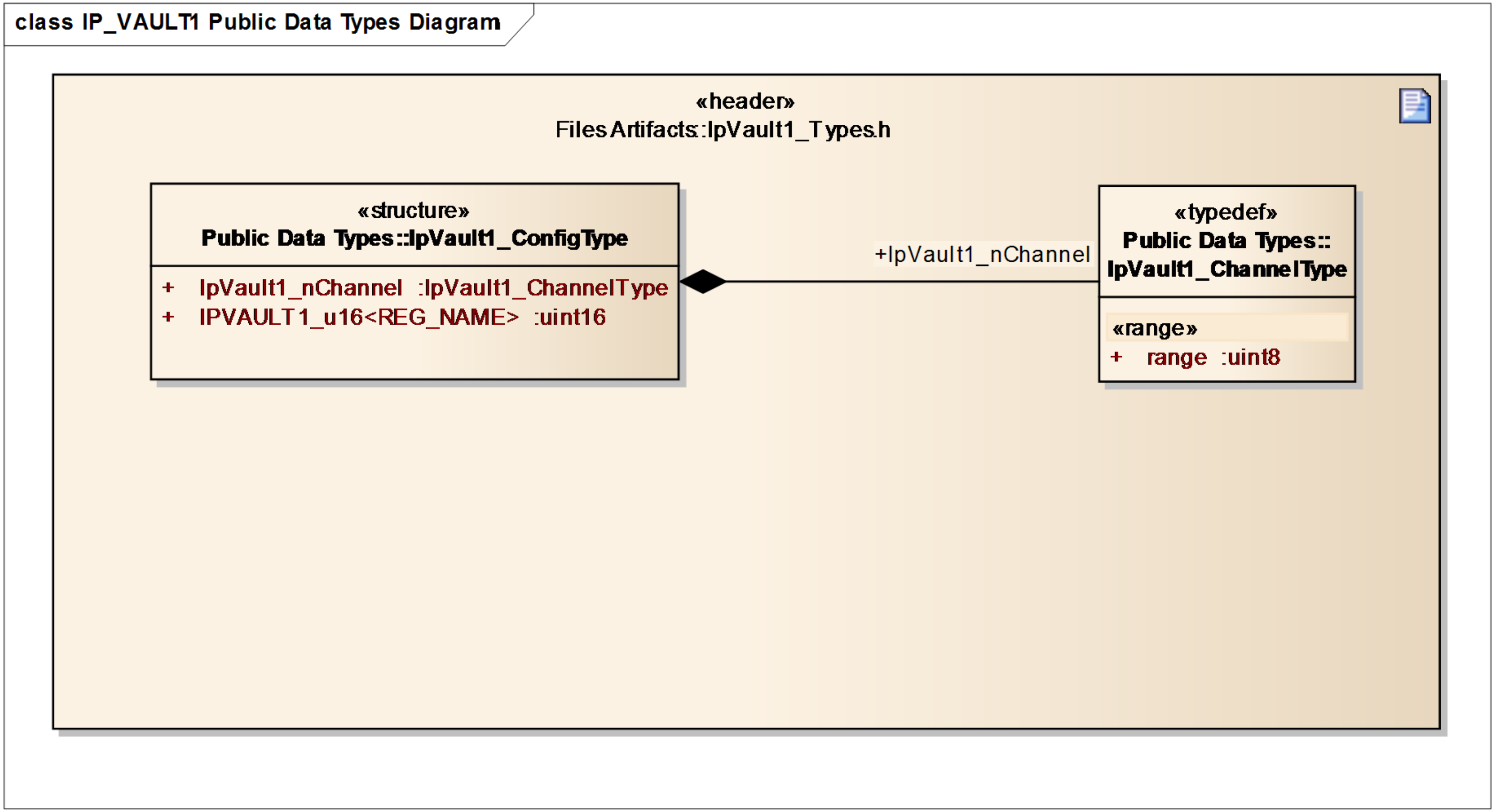
Class Diagrams SMCAL Examples

✓Platform Types



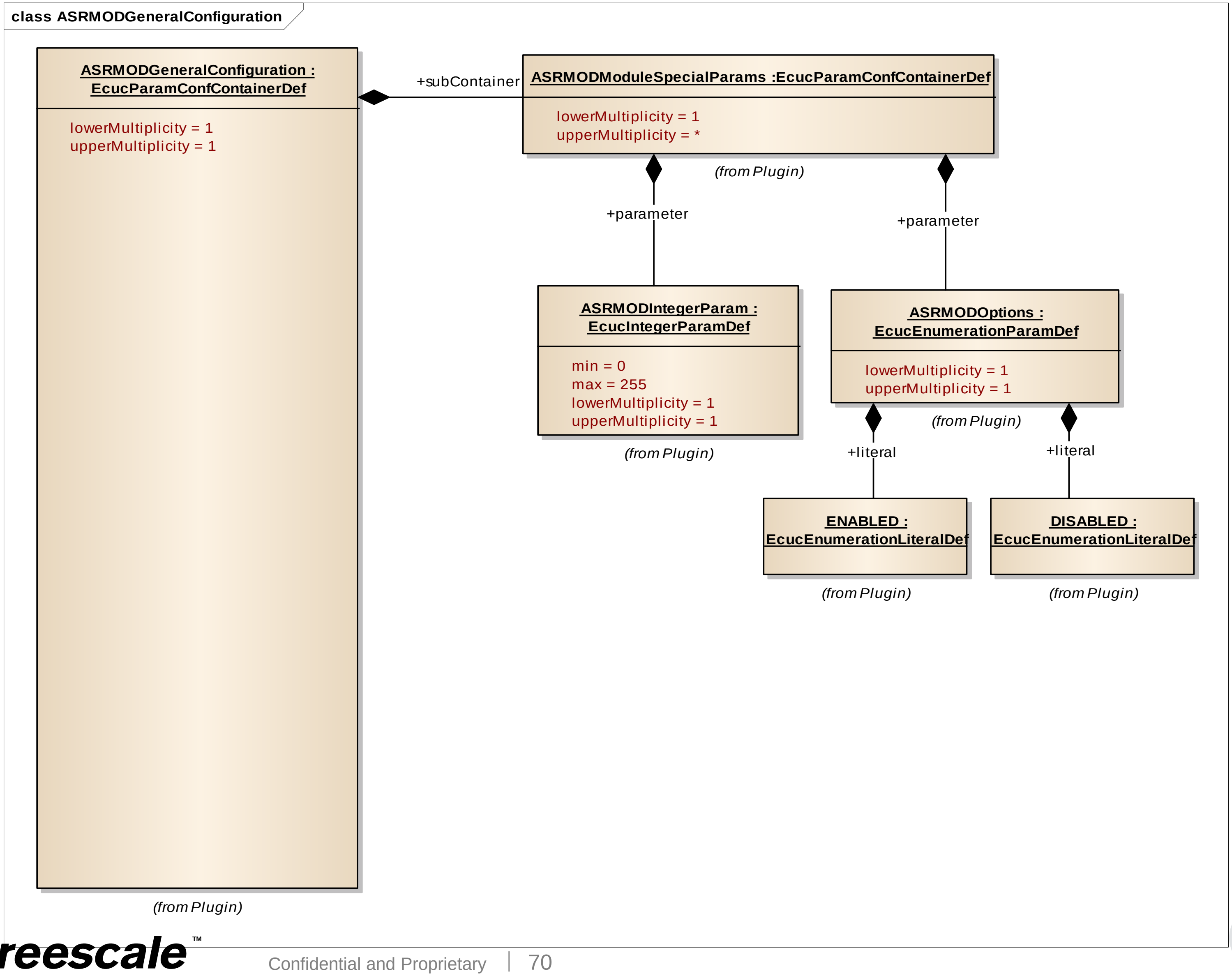
Class Diagrams SMCAL Examples

✓IPV Public Data Types


class IP_VAULT1 Public Data Types Diagram
«header»
Files Artifacts::IpVault1_Types.h
«structure»
Public Data Types::IpVault1_ConfigType
+ IpVault1_nChannel :IpVault1_ChannelType
+ IPVAULT1_u16<REG_NAME> :uint16
+IpVault1_nChannel
«typedef»
Public Data Types::
IpVault1_ChannelType
«range»
+ range :uint8

Class Diagrams SMCAL Examples

✓Plugin Diagram



Activity Diagrams Elements and Connectors

- **Activity Diagrams** are used to model the behaviors of a system, and the way in which these behaviors are related in an overall flow of the system. The logical paths a process follows, based on various conditions, concurrent processing, data access, interruptions and other logical path distinctions, are all used to construct a process, system or procedure.

Note: Are used to model system behavior and to model dynamic element interactions

Activity Diagrams Elements and Connectors and SMCAL Usage

Elements	Connectors
Activity	Control flow
Action	Object flow
ActivityInitial	
ActivityFinal	
Object	
ActivityPartition	
ActivityParameter	
Boundary	
Decision	

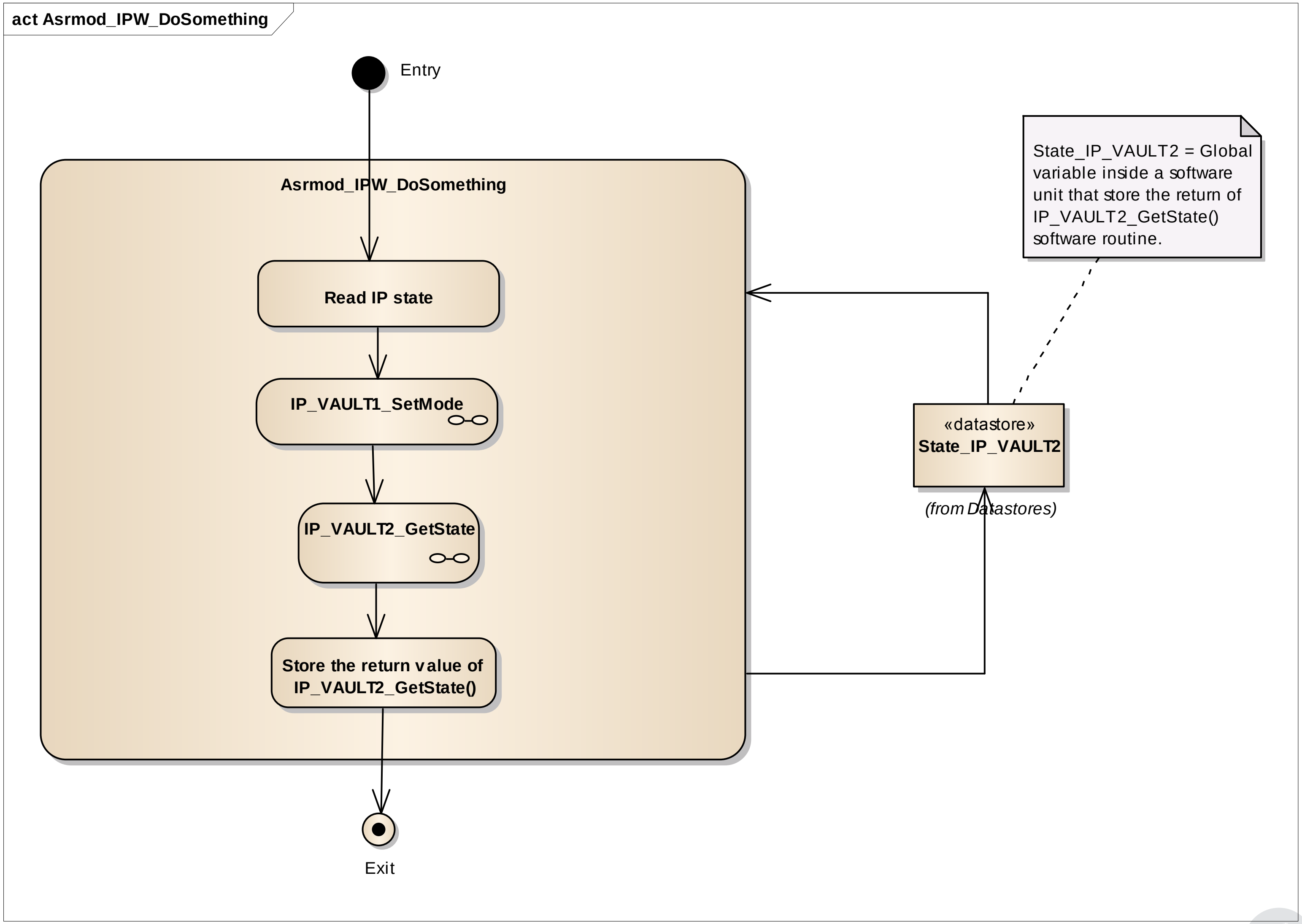
✓ASRMOD_HLD Functions

✓IPW Public Functions

✓IPVault Functions

Activity Diagrams SMCAL Examples

✓IPVault Function


act Asrmod_IPW_DoSomething
Entry
Asrmod_IPW_DoSomething
Read IP state
IP_VAULT1_SetMode
IP_VAULT2_GetState
Store the return value of IP_VAULT2_GetState()
Exit
«datastore»
State_IP_VAULT2
(from Datastores)
State_IP_VAULT2 = Global variable inside a software unit that store the return of IP_VAULT2_GetState() software routine.

State Machine Diagrams Elements and Connectors

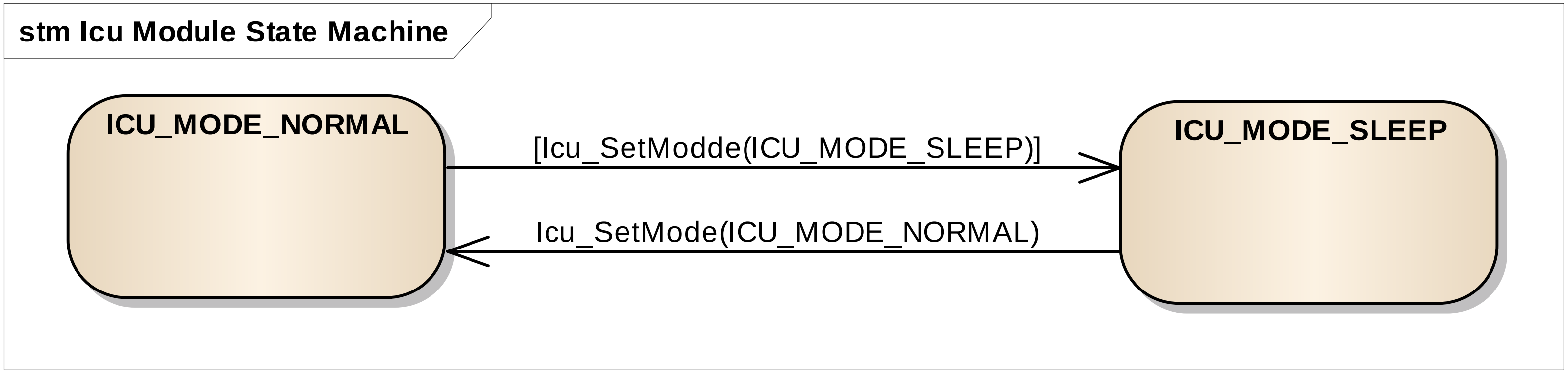
- **State Machine Diagrams** illustrate how an element (often a Class) can move between states, classifying its behavior according to transition triggers and constraining guards.

- **Elements and Connectors used in State Machine Diagrams**

Elements	Connectors
State	Transition

State Machine Diagrams SMCAL Examples

✓ASR Module State Machine
✓ASR Module Channel State Machine



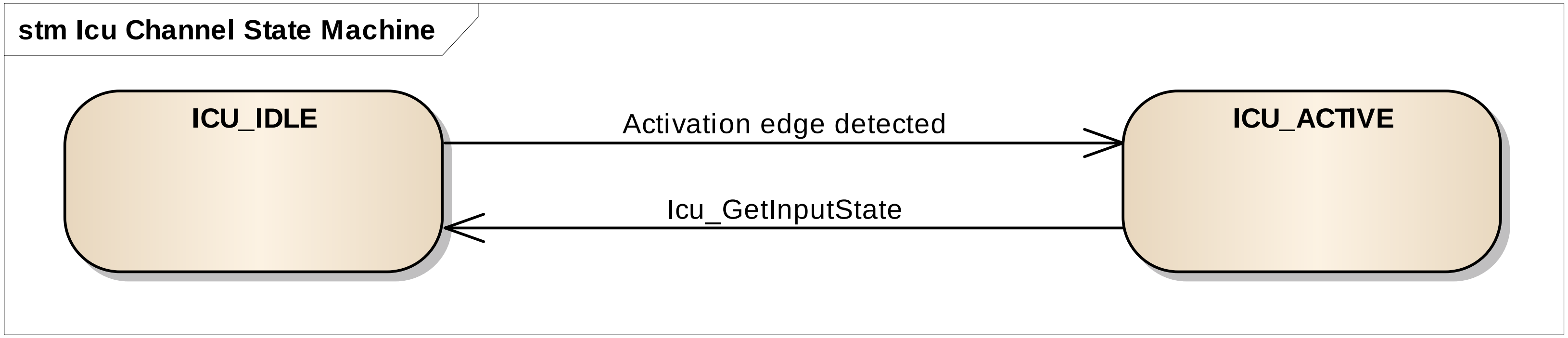
stm Icu Module State Machine

ICU_MODE_NORMAL

[Icu_SetModde(ICU_MODE_SLEEP)]

ICU_MODE_SLEEP

Icu_SetMode(ICU_MODE_NORMAL)



stm Icu Channel State Machine

ICU_IDLE

Activation edge detected

ICU_ACTIVE

Icu_GetInputState

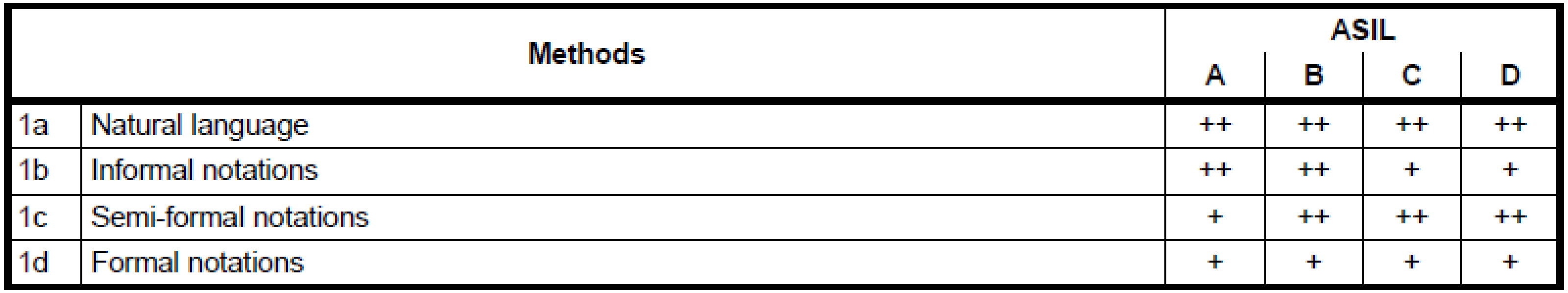
Part 3: ISO26262 – Part 6
SW Unit Design Requirements

ISO26262 Related Requirements

6- 8.4.2 To ensure that the software unit design captures the information necessary to allow the subsequent development activities to be performed correctly and effectively, the software unit design shall be described using the notations listed in Table 7.

NOTE In the case of model-based development with automatic code generation, the methods for representing the software unit design are applied to the model which serves as the basis for the code generation.

Table 7 — Notations for software unit design



Methods		ASIL			
		A	B	C	D
1a	Natural language	++	++	++	++
1b	Informal notations	++	++	+	+
1c	Semi-formal notations	+	++	++	++
1d	Formal notations	+	+	+	+

ISO26262 Related Requirements

6- 8.4.3 The specification of the software units shall describe the functional behavior and the internal design to the level of detail necessary for their implementation.
EXAMPLE Internal design can include constraints on the use of registers and storage of data.

ISO26262 Related Requirements

6 - 8.4.4 Design principles for software unit design and implementation at the source code level as listed in Table 8 shall be applied to achieve the following properties:

a) correct order of execution of subprograms and functions within the software units, based on the software architectural design;
b) consistency of the interfaces between the software units;
c) correctness of data flow and control flow between and within the software units;
d) simplicity;
e) readability and comprehensibility;
f) robustness; EXAMPLE Methods to prevent implausible values, execution errors, division by zero, and errors in the data flow and control flow.
g) suitability for software modification; and
h) testability.

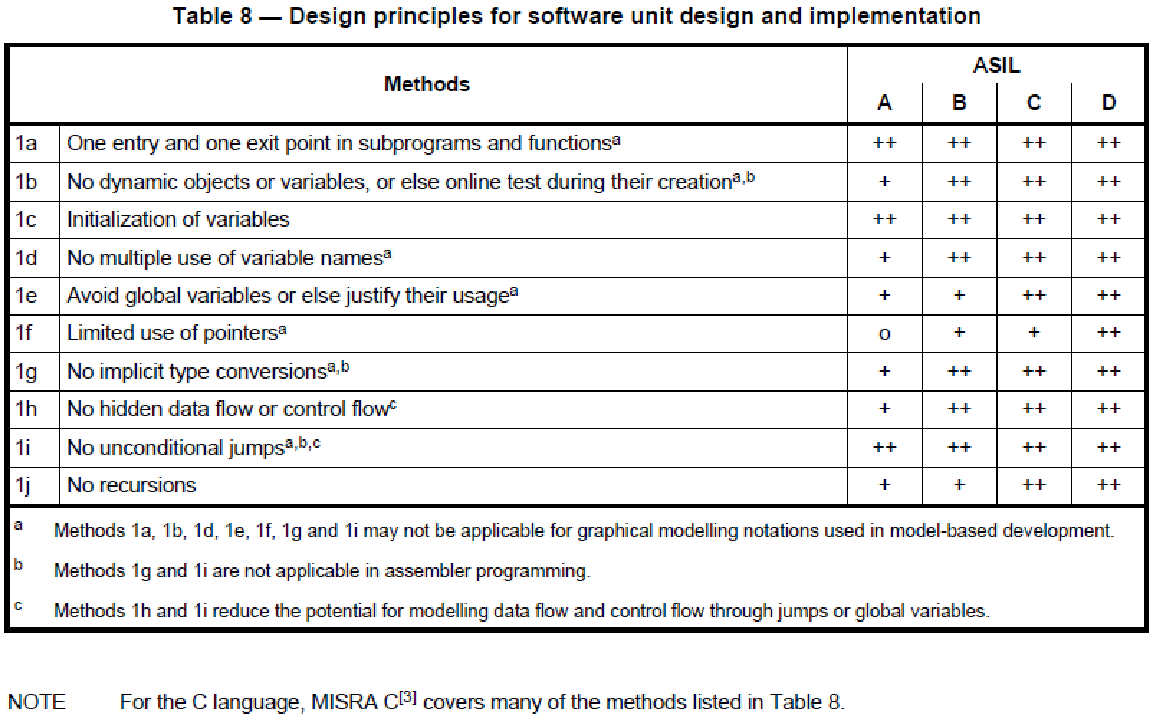


Table 8 — Design principles for software unit design and implementation

	Methods	ASIL			
		A	B	C	D
1a	One entry and one exit point in subprograms and functions[a]	++	++	++	++
1b	No dynamic objects or variables, or else online test during their creation[a,b]	+	++	++	++
1c	Initialization of variables	++	++	++	++
1d	No multiple use of variable names[a]	+	++	++	++
1e	Avoid global variables or else justify their usage[a]	+	+	++	++
1f	Limited use of pointers[a]	o	+	+	++
1g	No implicit type conversions[a,b]	+	++	++	++
1h	No hidden data flow or control flow[c]	+	++	++	++
1i	No unconditional jumps[a,b,c]	++	++	++	++
1j	No recursions	+	+	++	++

[a] Methods 1a, 1b, 1d, 1e, 1f, 1g and 1i may not be applicable for graphical modelling notations used in model-based development.

[b] Methods 1g and 1i are not applicable in assembler programming.

[c] Methods 1h and 1i reduce the potential for modelling data flow and control flow through jumps or global variables.

NOTE For the C language, MISRA C[3] covers many of the methods listed in Table 8.

ISO26262 Related Requirements

6 - 8.4.5 The software unit design and implementation shall be verified in accordance with ISO 26262-8:2011 Clause 9, and by applying the verification methods listed in Table 9, to demonstrate:

a) the compliance with the hardware-software interface specification (in accordance with ISO 26262-5:2011, 6.4.10);

b) the fulfilment of the software safety requirements as allocated to the software units (in accordance with 7.4.9) through traceability;

c) the compliance of the source code with its design specification; NOTE In the case of model-based development, requirement c) still applies.

d) the compliance of the source code with the coding guidelines (see 5.5.3); and

e) the compatibility of the software unit implementations with the target hardware.

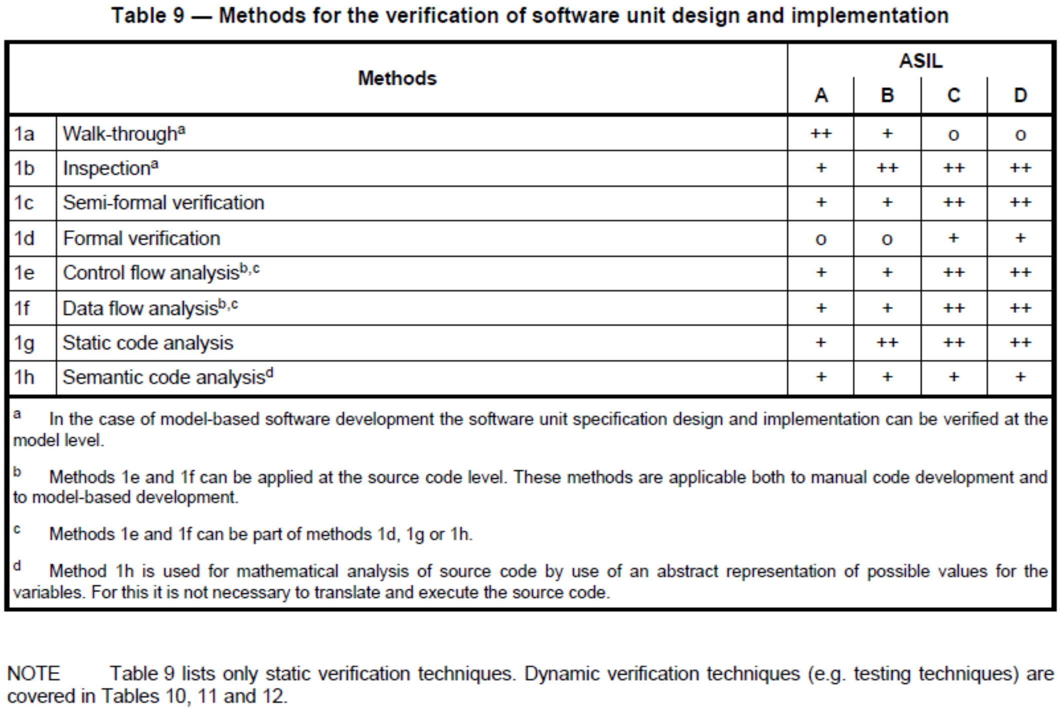


Table 9 — Methods for the verification of software unit design and implementation

Methods		ASIL			
		A	B	C	D
1a	Walk-through[a]	++	+	o	o
1b	Inspection[a]	+	++	++	++
1c	Semi-formal verification	+	+	++	++
1d	Formal verification	o	o	+	+
1e	Control flow analysis[b,c]	+	+	++	++
1f	Data flow analysis[b,c]	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Semantic code analysis[d]	+	+	+	+

[a] In the case of model-based software development the software unit specification design and implementation can be verified at the model level.

[b] Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

[c] Methods 1e and 1f can be part of methods 1d, 1g or 1h.

[d] Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

NOTE Table 9 lists only static verification techniques. Dynamic verification techniques (e.g. testing techniques) are covered in Tables 10, 11 and 12.

The Goal:

Being able to understand how the requirements will be deployed in the final software:

- *Understanding the overall architecture*
- *Being able to follow the logical flow of the software*
- *Being able to follow the data flow on the invocation path and across services*

ISO26262 Related Goals

2) Unit Testing

6-9.4.3 The software unit testing methods listed in Table 10 shall be applied to demonstrate that the software units achieve:

−compliance with the software unit design specification (in accordance with Clause 8);

−compliance with the specification of the hardware-software interface (in accordance with ISO 26262-5:2011, 6.4.10) ;

−the specified functionality ;

−confidence in the absence of unintended functionality ;

−robustness; and

EXAMPLE The absence of inaccessible software, the effectiveness of error detection and error handling mechanisms.

−sufficient resources to support their functionality.

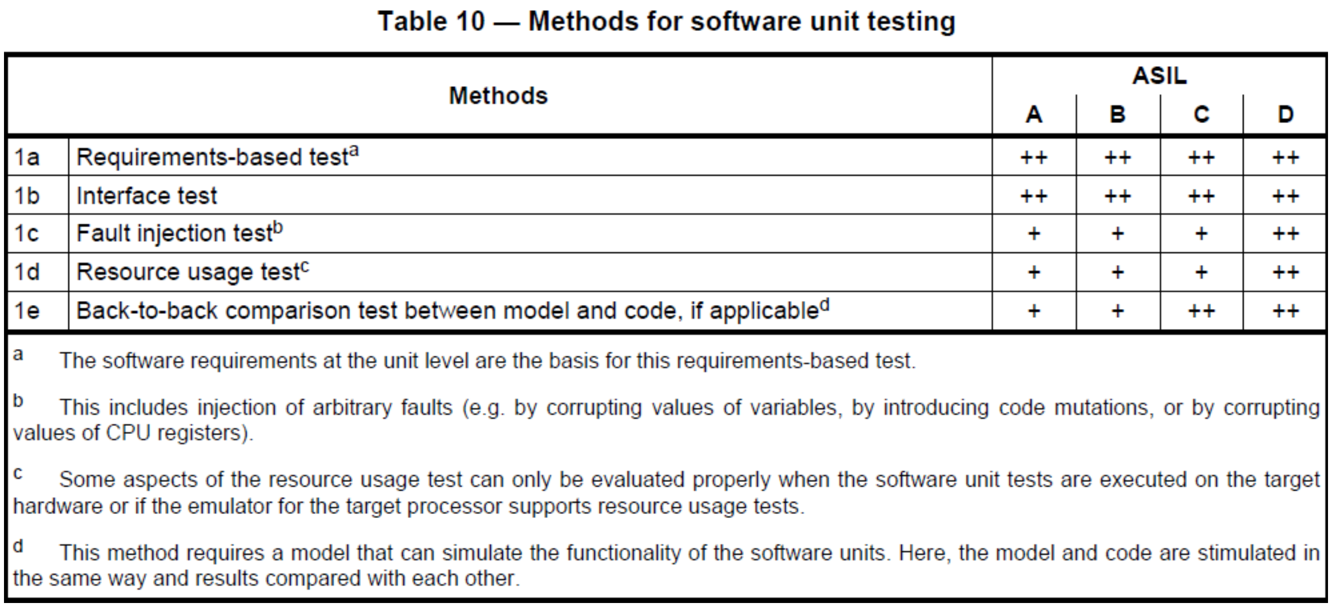


Table 10 — Methods for software unit testing

Methods		ASIL			
		A	B	C	D
1a	Requirements-based test[a]	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test[b]	+	+	+	++
1d	Resource usage test[c]	+	+	+	++
1e	Back-to-back comparison test between model and code, if applicable[d]	+	+	++	++

[a] The software requirements at the unit level are the basis for this requirements-based test.

[b] This includes injection of arbitrary faults (e.g. by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

[c] Some aspects of the resource usage test can only be evaluated properly when the software unit tests are executed on the target hardware or if the emulator for the target processor supports resource usage tests.

[d] This method requires a model that can simulate the functionality of the software units. Here, the model and code are stimulated in the same way and results compared with each other.

freescale™

www.Freescale.com