EULYNX Initiative

Europe's Rail Joint Undertaking

Interpretation rules for model-based requirements

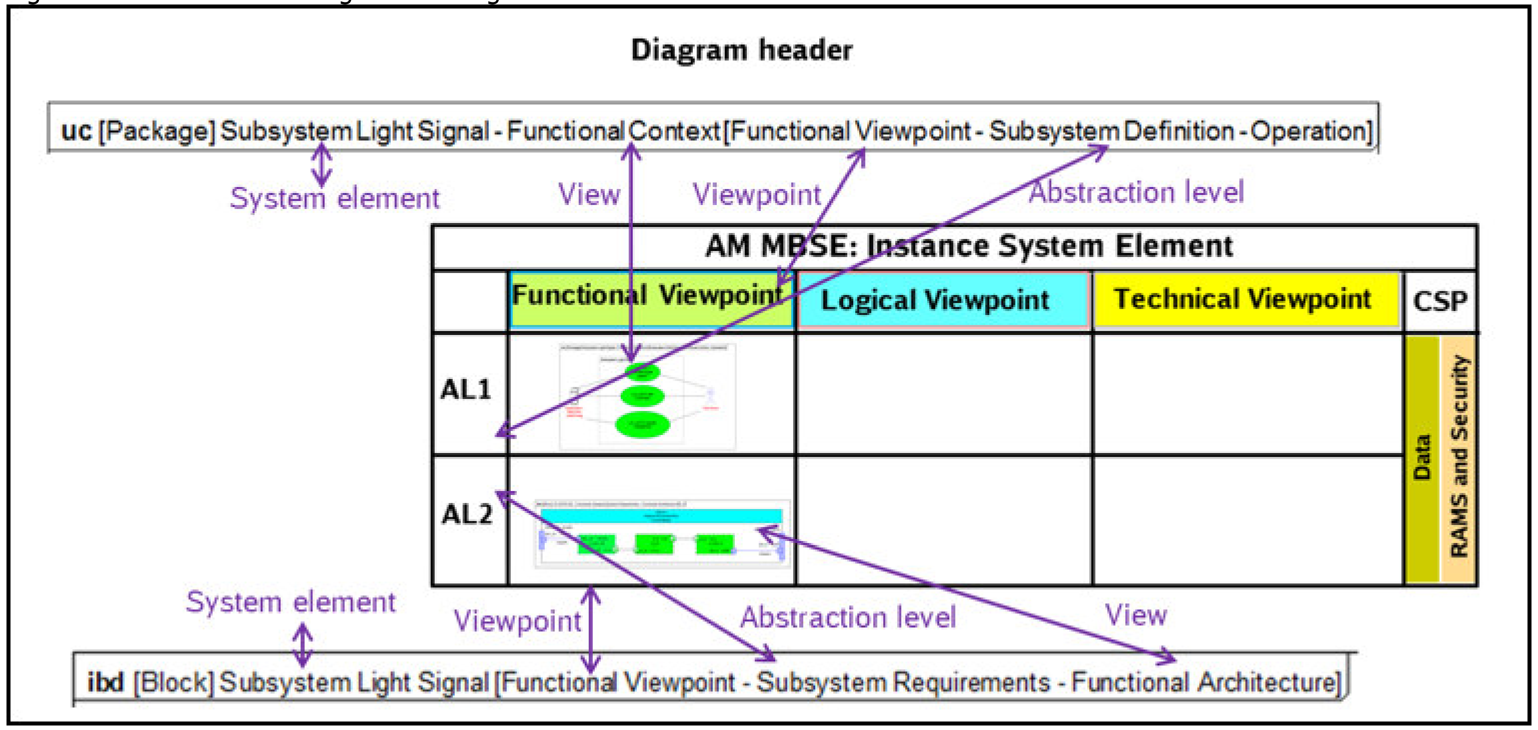
Document number: Eu.Doc.29
Version: 2.1 (0.A)

Contents

ID	Requirements
Eu.ModIn.6	**1 Introduction**
Eu.ModIn.7	**1.1 Release information**
Eu.ModIn.8	[Eu.Doc.29] Interpretation rules for model-based requirements CENELEC Phase: 4-5 Version: 2.1 (0.A) EULYNX Baseline Set: 4 Approval date: ---
Eu.ModIn.232	**Version history**
Eu.ModIn.229	version number: 1.0 date: 17.2.2017 author: Randolf Berglehner review: - changes: -
Eu.ModIn.234	version number: 1.1 (0.A) date: 21.3.2017 author: Randolf Berglehner review: CCB changes: EUMT-11
Eu.ModIn.235	version number: 1.1 (1.A) date: 08.12.2017 author: Randolf Berglehner review: CCB changes: EUB-120
Eu.ModIn.236	version number: 1.2 (0.A) date: 16.11.2018 author: Randolf Berglehner, Dennis Kunz review: - changes: update regarding new Modelling Standard version and new structure of the requirements specifications.
Eu.ModIn.337	version number: 1.3 (0.A) date: 10.12.2018 author: Randolf Berglehner review: CCB changes: EUMT-50, EUMT-51
Eu.ModIn.602	version number: 1.4 (0.A) date: 07.10.2019 author: Randolf Berglehner review: - changes: update according advancement of Eu.Doc.30 to be reviewed.
Eu.ModIn.615	version number: 1.4 (1.A) date: 25.10.2019 author: Randolf Berglehner review: Dennis Kunz (Signon), Martin Herz (Expleo) changes: update according to review results
Eu.ModIn.616	version number: 1.5 (0.A) date: 03.12.2019 author: Randolf Berglehner review: CCB changes: EUMT-59

ID	Requirements
Eu.ModIn.697	version number: 1.6 (0.A) date: 18.11.2021 author: Randolf Berglehner review: Dennis Kunz, Filip Giering, Felix Auris changes: complete revision due to further development of the methodology.
Eu.ModIn.740	version number: 1.7 (0.A) date: 24.02.2022 author: Randolf Berglehner review: CCB, UNIFE, Felix Auris changes: Incorporation of the CCB and UNIFE notes. Integration of trigger ports and views of the technical viewpoint at abstraction level AL2.
Eu.ModIn.834	version number: 1.7 (1.A) date: 12.04.2022 author: Randolf Berglehner review: --- changes: Synchronisation of the content of Eu.Doc.30 and Eu.Doc.29 - Baseline for CCB review BL4R1.
Eu.ModIn.835	version number: 2.0 (0.A) date: 02.05.2022 author: Randolf Berglehner review: CCB changes: CCB comments incorporated. Baseline approved by CCB.
Eu.ModIn.848	version number: 2.1 (0.A) date: 08.12.2023 author: Randolf Berglehner review: M&T changes:EUMT-61, EUMT-62, EUMT-63, EUMT-64, EUMT-65, EUMT-66, EUMT-70, EUMT-75, EUMT-76, EUMT-79, EUP-497
Eu.ModIn.9	**1.2 Impressum**
Eu.ModIn.10	Publisher: **EULYNX Initiative** A full list of the EULYNX Partners can be found on www.eulynx.eu/index.php/members.
Eu.ModIn.11	Responsible for this document: EULYNX Project Management Office www.eulynx.eu
Eu.ModIn.233	Copyright EULYNX Partners All information included or disclosed in this document is licensed under the European Union Public Licence EUPL, Version 1.1.
Eu.ModIn.12	**1.3 Purpose**
Eu.ModIn.13	This document explains the methodology introduced in the document Modelling Standard [Eu.Doc.30] and the language elements of the System Modeling Language (SysML). The document is written with the purpose to enable the reader of model-based requirements to interpret the requirements to be implemented, without having to acquire detailed knowledge of the SysML language.
Eu.ModIn.14	In order to avoid complexity, the language scope of the UML/SysML is restricted for the purpose of this document. More detailed explanations of the methodology used and the syntax and semantics of the SysML elements used can be found in the documents Modelling Standard [Eu.Doc.30], the SysML specification [https://sysml.org/.res/docs/specs/OMGSysML-v1.6-19-11-01.pdf] or the UML specification [https://www.omg.org/spec/UML/2.5.1/PDF].
Eu.ModIn.871	It should also be noted that the inserted diagrams are only to be understood as examples for methodological explanation and, although there are similarities to the content of current specifications, are not intended to convey any specification-specific content. The relevant specifications should be consulted for specification-specific content.
Eu.ModIn.741	Unlike the EULYNX specification documents, this document does not have an extra "Type" column to save space. A column "Type" is not necessary because all objects, apart from the headings, are of the type "Info". This means that the entire content is to be understood as information.
Eu.ModIn.15	**1.4 Objectives of the model-based requirements definition**
Eu.ModIn.16	The model-based requirements definition is used to: • enable a continuous CENELEC-compatible top-down specification of a (sub)system (refinement of the requirements across different abstraction levels) • describe the functional requirements of a (sub)system or an interface operationally and therefore suitable for simulation, i.e. testable in a uniform format

ID	Requirements
	• support achieving consistency, non-ambiguity and completeness of the requirements as far as possible • allow for the testing by simulation of the functional requirements of a (sub)system or an interface already during the specification phase (moving error detection to the specification phase) • support the generation of (sub)system or interface test cases from the requirements specification
Eu.ModIn.17	**1.5 Boundary conditions of modelling**
Eu.ModIn.18	The functional system requirements are described in a consistent, non-ambiguous and compact form using the standardised semiformal language SysML. The SysML model elements and their interaction are to be understood as a means of describing the system requirements and not as implementation specifications. They are to be implemented with regard to their semantics. The type of representation and the underlying methodology sometimes differs from common text-based specifications. However, the requirements can be further processed into functional specifications and products in accordance with the tested processes.
Eu.ModIn.19	**1.6 Applicable standards and regulations**
Eu.ModIn.198	A list of applicable standards and regulations used in EULYNX is listed in the EULYNX Reference Document List [Eu.Doc.12].
Eu.ModIn.23	**1.7 Terms and abbreviations**
Eu.ModIn.199	The terms and abbreviations are listed in the EULYNX Glossary [Eu.Doc.9].
Eu.ModIn.200	**1.8 Related documents**
Eu.ModIn.201	The current versions of documents used as input or related to this document are listed in the EULYNX Documentation Plan [Eu.Doc.11]. The relationships between the documents are displayed in the Appendix A1 Documentation plan and structure [Eu.Doc.11_A1].
Eu.ModIn.202	• Modelling Standard [Eu.Doc.30]
Eu.ModIn.34	**2 General structure of the requirement specifications**
Eu.ModIn.35	Following the definitions of the **EULYNX MBSE Specification Framework (MBSE SF)** [Eu.Doc.30], the functional system requirements are described in the form of a SysML model of the abstract solution of a • **System/Subsystem under Specification (SUS)** or • **System/Subsystem Interface under Specification (SIUS)**.
Eu.ModIn.330	The **Architecture Model MBSE (AM MBSE)** as vital part of the MBSE SF facilitates the seamless, modelbased description of a SUS or a SIUS from three core viewpoints namely • **Functional Viewpoint**, • **Logical Viewpoint** and • **Technical Viewpoint** and with varying degrees of granularity.
Eu.ModIn.341	A SUS or SIUS description from a specific viewpoint and with a specific degree of granularity is called a **view** (or model view). A view is represented by one or multiple SysML diagrams.
Eu.ModIn.344	The viewpoints describe a SUS or a SIUS with respect to different stakeholder concerns. However, these descriptions may vary in their degree of granularity. For complex systems in particular, it is reasonable to start with rather high-level descriptions. Once these high-level descriptions have been created, these views are typically refined and detailed step by step. Therefore, AM MBSE supports views with different degrees of granularity i.e. views at different abstraction levels (AL).
Eu.ModIn.340	Following **CENELEC (EN 50126)** and the **System engineering process [Eu.Doc.27]**, in the current models the following two abstraction levels of the AM MBSE are applied: • **AL1: Subsystem/Interface Definition**, • **AL2: Subsystem/Interface Requirements**
Eu.ModIn.654	Viewpoint, abstraction level and view name are made evident in the header of the diagram representing a certain view.
Eu.ModIn.656	Examples: • The view "Functional Context" depicted in *Figure 1* describing a certain aspect of system element Subsystem Light Signal by a SysML use case diagram (uc) belongs to the "Functional Viewpoint" and has the granularity of abstraction level AL1 (Subsystem Definition). • The view "Functional Architecture" depicted in *Figure 1* describing a certain aspect of system element Subsystem Light Signal by a SysML internal block diagram (ibd) belongs to the "Functional Viewpoint" and has the granularity of abstraction level AL2 (Subsystem Requirements).

ID	Requirements
Eu.ModIn.655	Figure 1 Structure of the diagram headings 
Eu.ModIn.789	**2.1 Binding nature of the requirements and their structuring**
Eu.ModIn.338	The SUS and SIUS SysML models are stored in the repository of the modelling tool. Relevant artefacts of them are depicted in a traceable manner as surrogates in the requirement specification documents in the form of atomic referenceable functional SUS or SUIS requirements.
Eu.ModIn.777	Each of these atomised requirements is assigned a liability in the form of an object type. A distinction is made between the object types "Req", "Def", "Info" and "Head".
Eu.ModIn.778	• "**Req**": This denotes a mandatory requirement.
Eu.ModIn.779	• "**Def**": This denotes referenceable model elements that are used in the model-based creation of requirements.
Eu.ModIn.781	• "**Info**": This denotes additional information to help understand the specification. These objects do not specify any additional requirements.
Eu.ModIn.782	• "**Head**": This denotes chapter headings.
Eu.ModIn.845	**Please note:** State machines or several state machines linked together in a Functional Architecture define the totality of all functional requirements of an SUS or an SIUS in a coherent and consistent manner. State diagrams of a corresponding state machine are marked with the object type "**Req**". For the later design and implementation, it is not the description language SysML that is binding, but the domain-specific meaning expressed by it. The specified behaviour can be converted into a vendor specific language but must retain the domain specific meaning describing the functional requirements. The specific model elements are additionally specified and defined by object type "**Def**" to allow for traceability to supplier designs or test cases. The compliance of products to the specifications must be demonstrated by testing against EULYNX test cases, which are derived from the functionality specified by the models.
Eu.ModIn.833	**Please note:** The bindings assigned to each model view in this document can be adjusted on a project-specific basis. Thus, the bindings assigned in the specifications always apply.
Eu.ModIn.339	A functional requirement consists of the respective SysML model element, for instance a SysML diagram, and if necessary, an additional extension of it.
Eu.ModIn.342	For this reason, functional requirements have two attributes "**Requirement Part 1**" and "**Requirement Part 2**", which are shown in adjacent columns (see Figure 2).
Eu.ModIn.36	In "Requirement Part 1" the respective SysML model element is listed and in "Requirement Part 2" the corresponding extension is shown. Column 'Type' defines the bindingness of the requirement and applies normally both to "Requirement Part 1" and "Requirement Part 2".
Eu.ModIn.37	In the case of requirements with a binding character "**Req**", in which the "Requirement Part 2" is provided with the heading "**Information**", the defined binding character "**Req**" only applies to "Requirement Part 1".
Eu.ModIn.293	Figure 2 "Requirement Part 1" and "Requirement Part 2" as shown in the requirement specifications.

<table>
<tr><th>ID</th><th>Requirements</th></tr>
<tr><td></td><td>
<table>
<tr><th>ID</th><th>Type</th><th>Requirement Part 1</th><th>Requirement Part 2</th></tr>
<tr><td>Eu.LS.4687</td><td>Req</td><td>Cd_Indicate_Signal_Aspect</td><td>Command (Cd) from the Subsystem - Electronic Interlocking to the Subsystem - Light Signal to indicate the transmitted Signal Aspect.</td></tr>
</table>
</td></tr>
<tr><td>Eu.ModIn.343</td><td>Just this partition of requirements is applied throughout the entire requirement specification document regardless of whether a requirement has its origins in the SUS or SIUS model or it is for example a text-based nonfunctional requirement manually added.</td></tr>
<tr><td>Eu.ModIn.333</td><td>In the following chapters of this document, the SUS/SIUS views represented by the diagrams used in the model are explained. For each model element a rule is provided that defines how the element is to be interpreted as a requirement. Chapter 4 concentrates on the model views used to specify the EULYNX subsystems and chapter 5 the ones to define standard communication interfaces. The model views for the description of functional entities (FE) and technical functional entities (TFE) used for both the specification of EULYNX subsystems and EULYNX interfaces are defined in chapter 6.</td></tr>
<tr><td>Eu.ModIn.334</td><td>As a prerequisite, chapter 3 defines needed underlying methodology based on [Eu.Doc.30], which is used in the abstraction levels.</td></tr>
<tr><td>Eu.ModIn.91</td><td><h1>3 Concept of model-based requirements</h1></td></tr>
<tr><td>Eu.ModIn.332</td><td>This chapter reflects necessary parts of the methodology defined in [Eu.Doc.30] and the rationale for the structure of the requirements in order to enable the correct interpretation of the current EULYNX specifications.</td></tr>
<tr><td>Eu.ModIn.359</td><td><h2>3.1 Basic characteristics of model-based requirements</h2></td></tr>
<tr><td>Eu.ModIn.346</td><td>User requirements are a model of the problem domain and define the results that the users want.</td></tr>
<tr><td>Eu.ModIn.347</td><td>System requirements (functional and nonfunctional) are a model of an abstract solution of the future SUS or SIUS and must be defined completely, correctly and consistently satisfying the user requirements. This has to be approved by means of verification and validation of the specification results.</td></tr>
<tr><td>Eu.ModIn.348</td><td>In order to support this verification and validation effort in the best possible way and keep the specification comprehensible for engineers, the EULYNX specification approach follows the objective of describing the functional system requirements of a SUS/SIUS in the form of an operational specification.</td></tr>
<tr><td>Eu.ModIn.349</td><td>An operational specification of a functional system requirement is a specification of a set of reproducible operations that can be executed by different stakeholders to find out whether or not the functional system requirement is present in the specification of a SUS or SIUS.</td></tr>
<tr><td>Eu.ModIn.350</td><td>For an operationally specified functional system requirement, there is a test that all stakeholders can perform and agree on the outcome - either the SUS or SIUS to be specified does or does not satisfy this requirement.</td></tr>
<tr><td>Eu.ModIn.352</td><td>The command control and signalling (CCS) systems currently specified in EULYNX are reactive systems and characterised by the constant interaction and synchronisation between the system and its environment.</td></tr>
<tr><td>Eu.ModIn.353</td><td>A reactive system, when switched on, engages in stimulus-response-behaviour in order to create desirable effects in its environment. For that reason, the EULYNX methodology proposes the specification of the functional system requirements in stimulus-response form. Stimulus-response specifications are an important class of operational specifications.</td></tr>
<tr><td>Eu.ModIn.351</td><td>A stimulus-response specification has the form

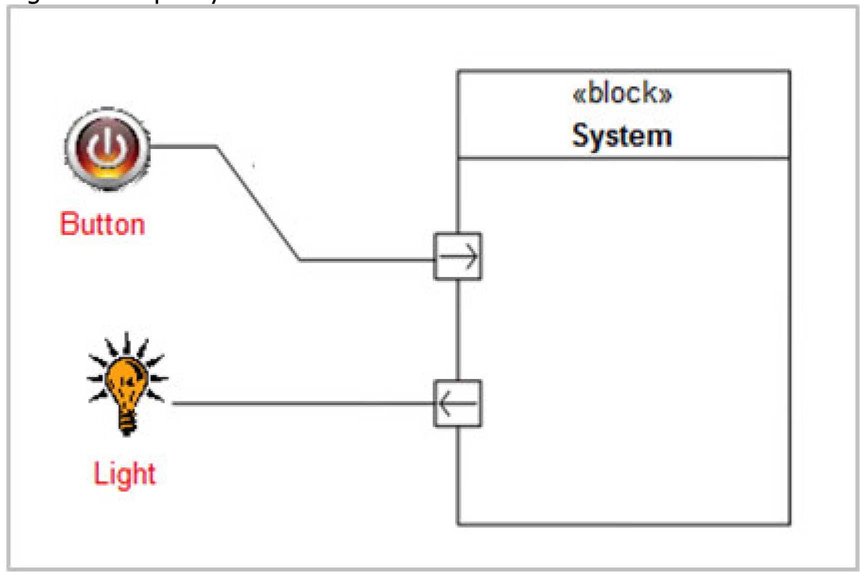
    s AND C = > r

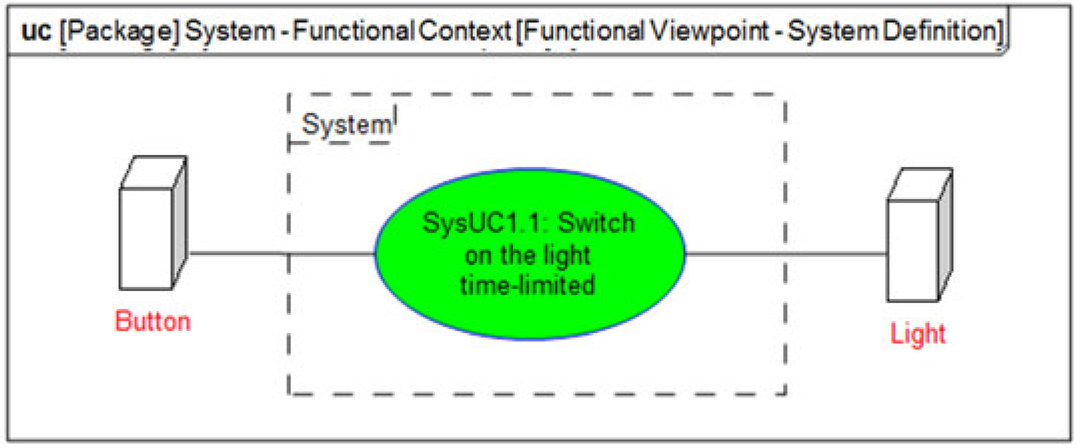
where s is a stimulus, C is a condition on the system state, and r is a response. The design process consists of decisions about r.</td></tr>
<tr><td>Eu.ModIn.370</td><td>In a nutshell, whenever a stimulus occurs there will be a corresponding response. The kind of response depends on the condition on the state of the system. Please note: this is also said to be a response if a stimulus occurs and the system "keeps quiet".</td></tr>
<tr><td>Eu.ModIn.354</td><td>A single stimulus-response pair is henceforth also referred to as an interaction.</td></tr>
<tr><td>Eu.ModIn.355</td><td>An interaction is generally formulated according to the following schema comprising four action steps:

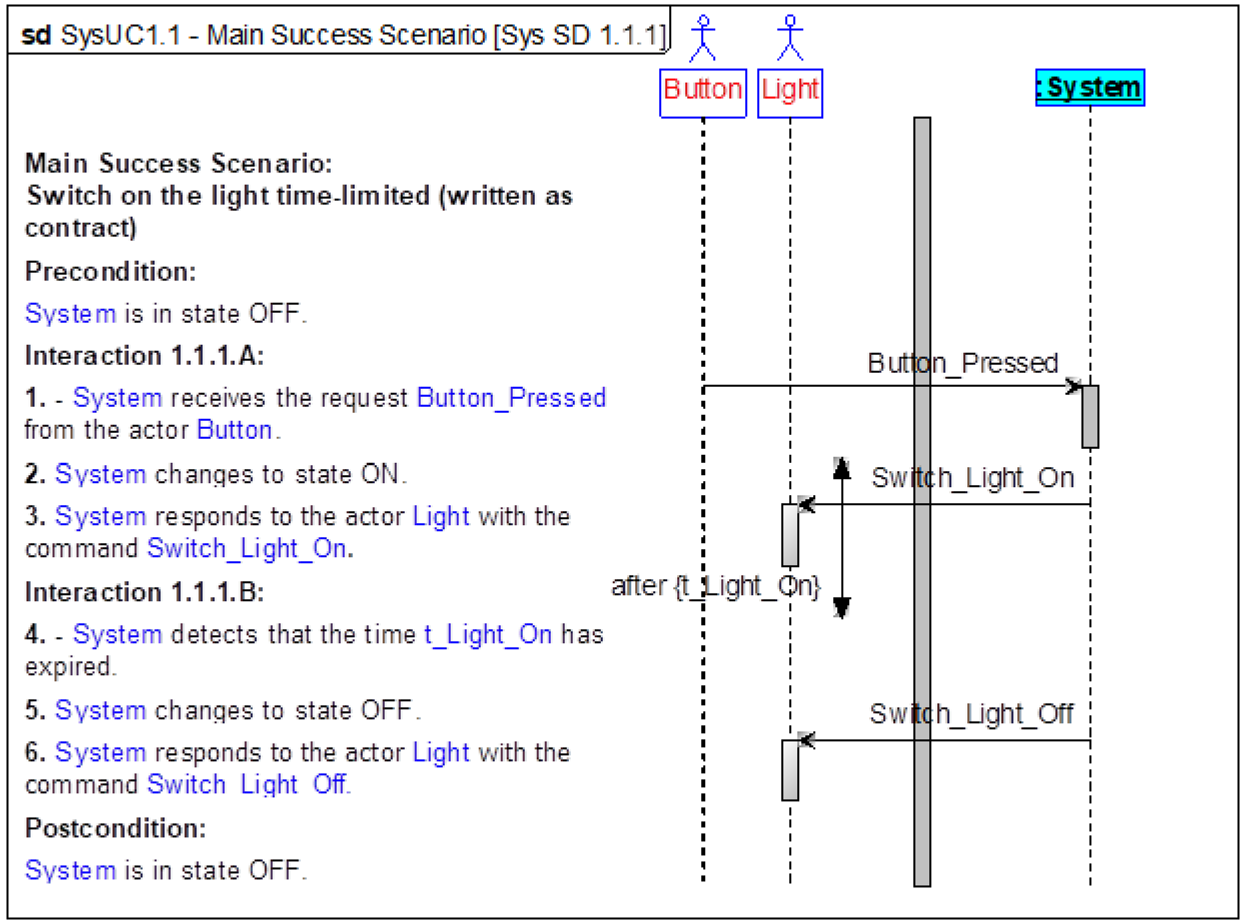
Interaction:
I. - The SUS or SIUS receives a stimulus.
II. The SUS or SIUS validates the stimulus.
III. The SUS or SIUS changes its internal state (or not).
IV. The SUS or SIUS responds with the result (Please note: a result may also be that the SUS or SIUS "keeps quiet").

However, there may be more than four action steps applied or fewer (see ID 358).</td></tr>
</table>

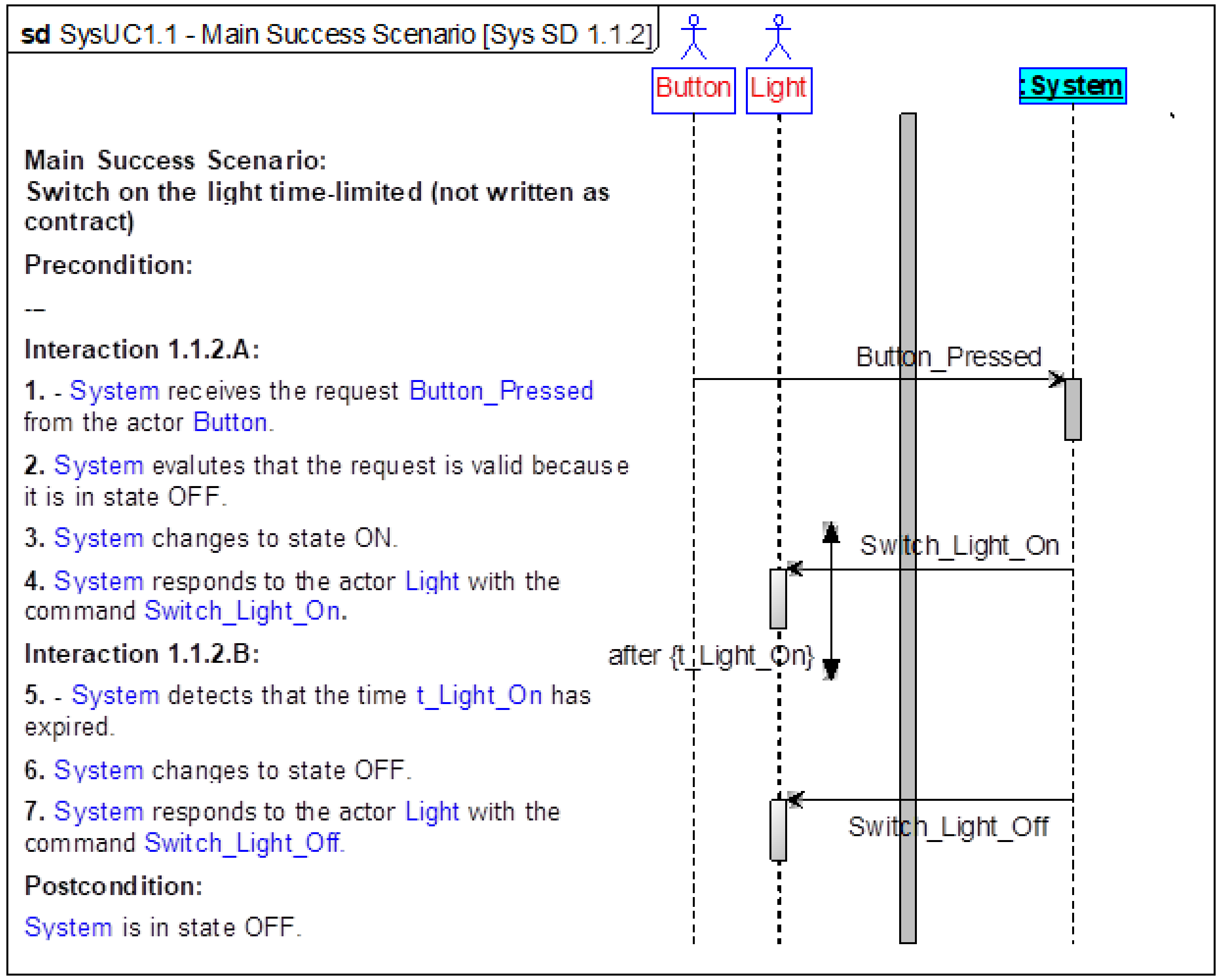
ID	Requirements
Eu.ModIn.356	An interaction always starts with the stimulus identified by a dash "-" (see step I in ID 355 above). A stimulus may have its origin • in the **request of a primary actor** (a primary actor is an actor in the environment of the SUS or SIUS who requires a service from it), • in a **timed trigger**, • in an **intrasystem event** (that is, an event that occurs in the system) or • in the **entering or leaving a system state**.
Eu.ModIn.103	**Interactions** may be extended to contracts.
Eu.ModIn.371	The central idea of **contracts** is a metaphor on how the **SUS** or **SIUS** and the actors collaborate on the basis of mutual obligations and benefits. Having written functional requirements in the style of interactions, those contracts can easily be obtained - interactions together with pre- and postconditions.
Eu.ModIn.357	If a **SUS** or **SIUS** provides a certain functionality, it may **a)** expect a certain condition to be guaranteed on entry by an actor that sends the request: the precondition of the interaction - an obligation for the actor, and a benefit for the SUS or SIUS, as it relieves it from having to handle the cases outside of the precondition. **b)** guarantee a certain property on exit: the postcondition of the interaction - an obligation for the system, and obviously a benefit (the main benefit of the request) for the actor.
Eu.ModIn.105	The following applies for **preconditions** and **postconditions** in this context: **a)** The interaction may only be triggered by the actor if the precondition is met; this presupposes that the actor knows the current system condition, **b)** The system must ensure in turn that the postcondition is met after the completion of the interaction. If no explicit postcondition has been defined (indicated by three dashes "---"), the requirement applies that the postcondition is identical to the precondition.
Eu.ModIn.358	A **contract** is formulated according to the following schema: **Precondition**: Definition of the precondition **Interaction:** **I.** - The SUS or SIUS receives a stimulus. **III.** The SUS or SIUS changes its internal state (or not). **IV.** The SUS or SIUS responds with the result (Please note: a result may also be that the SUS or SIUS "keeps quiet"). **Postcondition:** Definition of the postconditions
Eu.ModIn.106	Alternatively to this, functional system requirements may be written without using **contracts**. In these cases it can not be assumed that the actor knows the current SUS or SIUS condition and complies with the precondition. The preconditions of the interactions are empty and the SUS or SIUS must first check on itself whether the preconditions are met before responding to the stimulus. The above schema is modified as follows (see text in italics): **Precondition**: --- **Interaction**: **I.** - The SUS or SIUS receives a stimulus. **II.** *The SUS or SIUS validates the stimulus considering the current internal state*. **III.** The SUS or SIUS changes its internal state (or not). **IV.** The SUS or SIUS responds with the result (Please note: a result may also be that the SUS or SIUS "keeps quiet"). **Postcondition:** Definition of the postconditions
Eu.ModIn.107	In those cases, the check may fail in the second step. From this step on, a different internal condition might need to be entered and a different response might need to take place. **Variants of the interaction** would therefore have to be considered.
Eu.ModIn.360	**3.2 Basic description methods of model-based requirements**

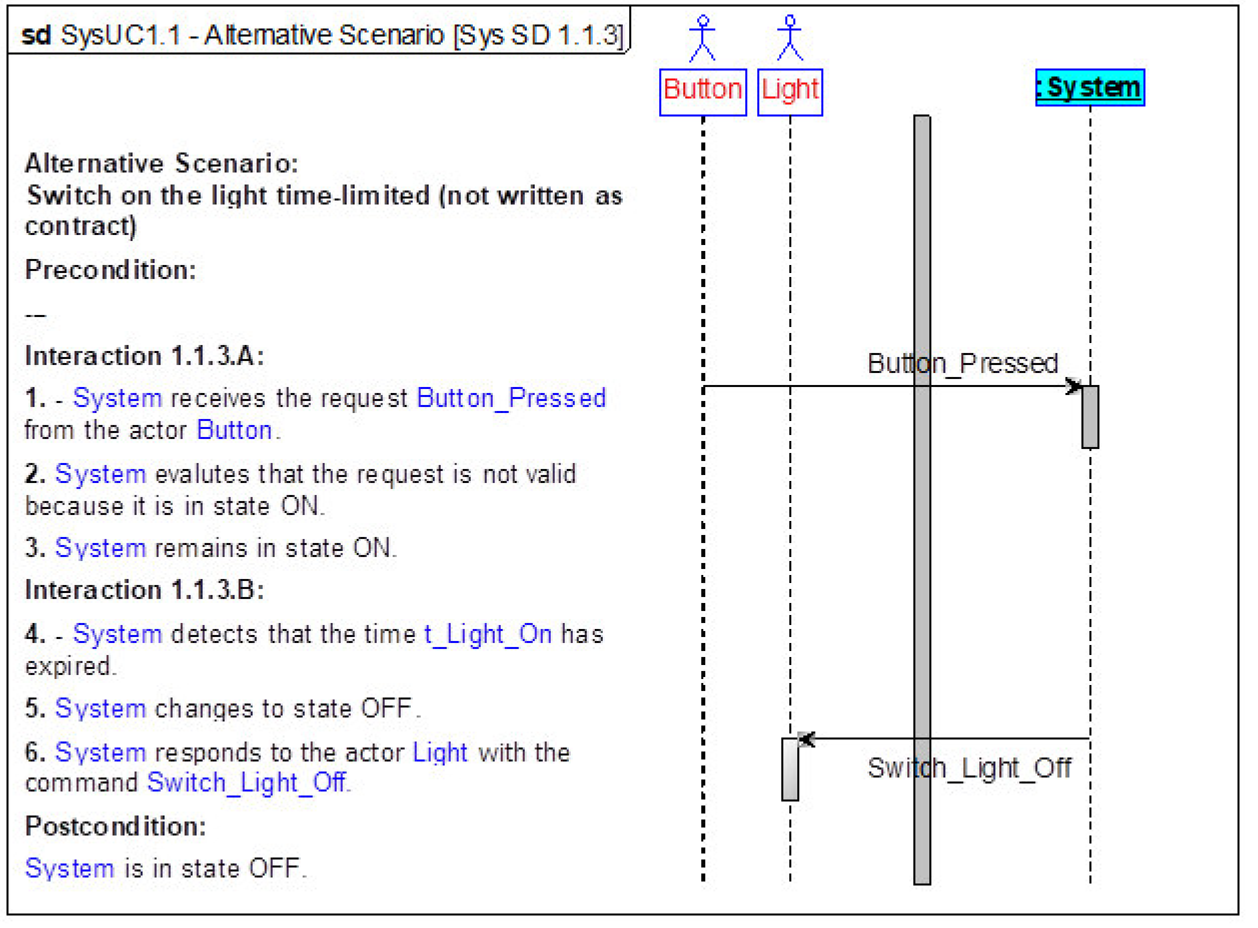
ID	Requirements
Eu.ModIn.94	**Interactions** and **contracts**, as defined above, provide the basic schemata for the model-based description of functional system requirements in **stimulus-response form**. Depending on the abstraction level two model-based description methods are used: • **Interaction scenarios** are used at abstraction level **AL1 System Definition** defining the interaction of the subsystem with its environment. • **State machines** are used at abstraction level **AL2 System Requirements** completely refining the externally visible stimulus-response behaviour described by means of the interaction scenarios at abstraction level AL1 System Definition.
Eu.ModIn.215	These two model-based description methods will be demonstrated defining the functional system requirements of a simple system based on the **functional user requirements (FUR)** listed below: **FUR1:** The user wants to switch on the light by pressing a button if the light is off, **FUR2:** The user wants the light to be switched off automatically after a defined time.
Eu.ModIn.373	As shown in *figure 3* the **SUS** named "**System**" is connected to the two actors "**Light**" and "**Button**" in the environment.
Eu.ModIn.213	Figure 3 Simple system 
Eu.ModIn.93	According to the **functional user requirements** described above the SUS is required to fulfil the **functional system requirements (FSR)**, described in classical textual form below: **FSR1:** The system shall switch on the light if the light is switched off and the button is pressed, **FSR2:** The system shall switch off the light automatically after the time t_Light_On has expired.
Eu.ModIn.100	**3.2.1 Description method using interaction scenarios**
Eu.ModIn.376	The functional user requirements **FUR1** and **FUR2** defined above (see ID 215) require the SUS "System" to provide a **service** for the users. As shown in *figure 4*, this service is defined as **system UseCase** "SysUC1.1: Switch on the light time-limited".
Eu.ModIn.377	**System UseCases** describe the functionality of a SUS or SIUS in terms of how it is used to achieve the goals of its various users. The users of a SUS or SIUS are described by **actors** (i.e. "Button" and "Light"), which may represent external systems or humans who interact with the system. A UseCase is denoted by an ellipse, and the actors participating in the UseCase are connected to the ellipse by solid lines.

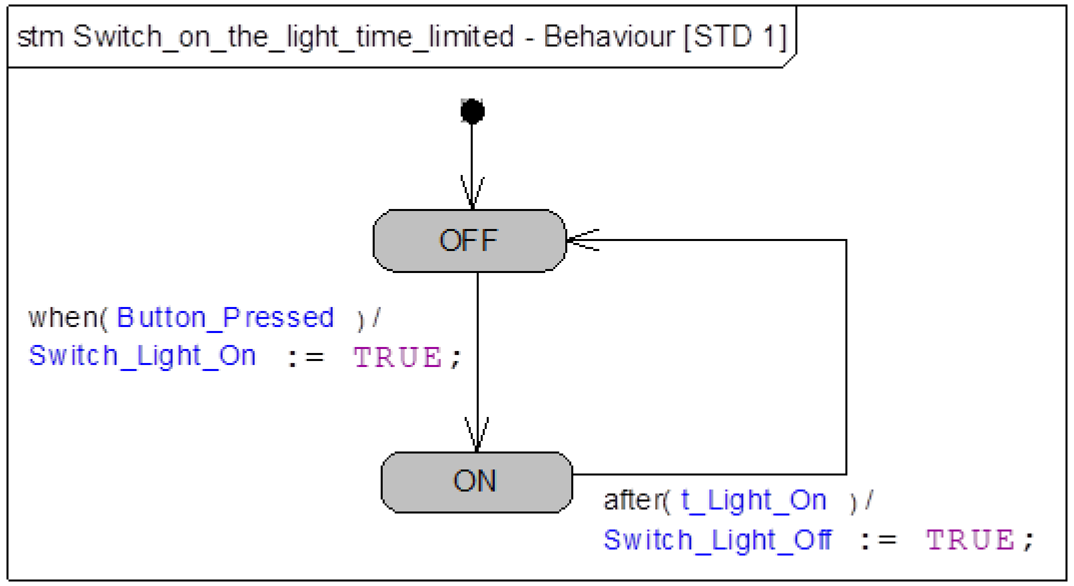
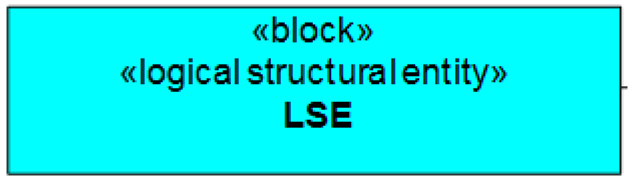
ID	Requirements
Eu.ModIn.361	Figure 4 UseCase shown in a UseCase diagram 
Eu.ModIn.362	A complete UseCase, i.e. a primary UseCase consists of **one or multiple interactions** which can alternatively be formulated as **contracts**. A UseCase having **only one interaction** is an interaction written as a UseCase.
Eu.ModIn.378	The interactions specifying a UseCase such as "**SysUC1.1: Switch on the light time-limited**" are described in a model-based way by **interaction scenarios**, also referred to as **use case scenarios**. Interaction scenarios are represented by **SysML sequence diagrams**.
Eu.ModIn.175	The specification of the interaction scenarios may cover a **standard sequence** and one or several **alternative sequences**, e.g. to represent a failed validation of the stimulus. Normally, the "good case" of an interaction scenario is specified in the "standard sequence" and deviating sequences in "alternative sequences". If no unique standard sequence can be determined, it is also possible that only "alternative sequences" exist.
Eu.ModIn.380	For this reason, a UseCase may be defined by interaction scenarios in the following compositions: - **one** Main Success Scenario and **any number** of Alternative scenarios, - **only one** Main Success Scenario, - **any number** of Alternative Scenarios **without** a Main Success Scenario.
Eu.ModIn.379	Several interactions may be combined directly after each other without explicitly depicting the pre- and postconditions between them in an interaction scenario if the **postconditions** of the previous interaction are identical to the **preconditions** of the subsequent interaction.
Eu.ModIn.101	If it can be assumed that the current state of the SUS is visible in its environment, the textually formulated functional requirements **FSR1** and **FSR2** (see ID 93) can be described as contracts: **FSR1:** **Precondition:** System is in state OFF **Interaction:** **I.** - System receives the request "Button_Pressed" from the actor "Button". **III.** System changes to state "ON". **IV.** System responds to the actor "Light" with the command "Switch_Light_On". **Postcondition:** System is in state ON **FSR2:** **Precondition:** System is in state ON **Interaction:** **I.** - System detects that the time "t_Light_ON" has expired. **III.** System changes to state "OFF". **IV.** System responds to the actor "Light" with the command "Switch_Light_OFF". **Postcondition:** System is in state OFF

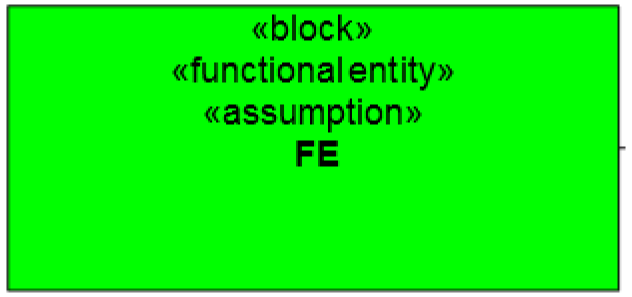
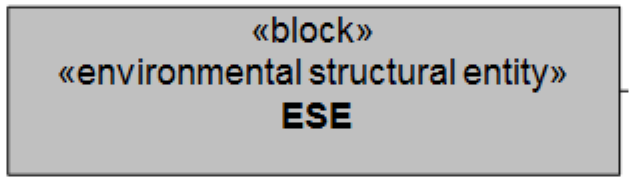
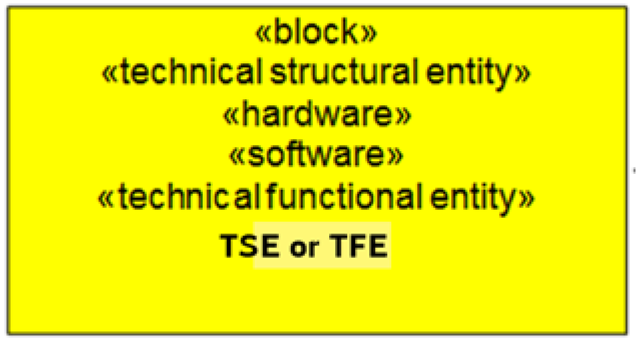
ID	Requirements
Eu.ModIn.104	The corresponding interaction scenario in the form of a Main Success Scenario is depicted in *Figure 5*. FSR1 and FSR2 are written as contracts and as a consequence no Alternative Scenarios are required. As the precondition of FSR2 is identical to the postcondition of FSR1 they are not explicitly depicted in the interaction scenario.
Eu.ModIn.363	Figure 5 Main Success Scenario with FSR1 and FSR2 written as contracts 

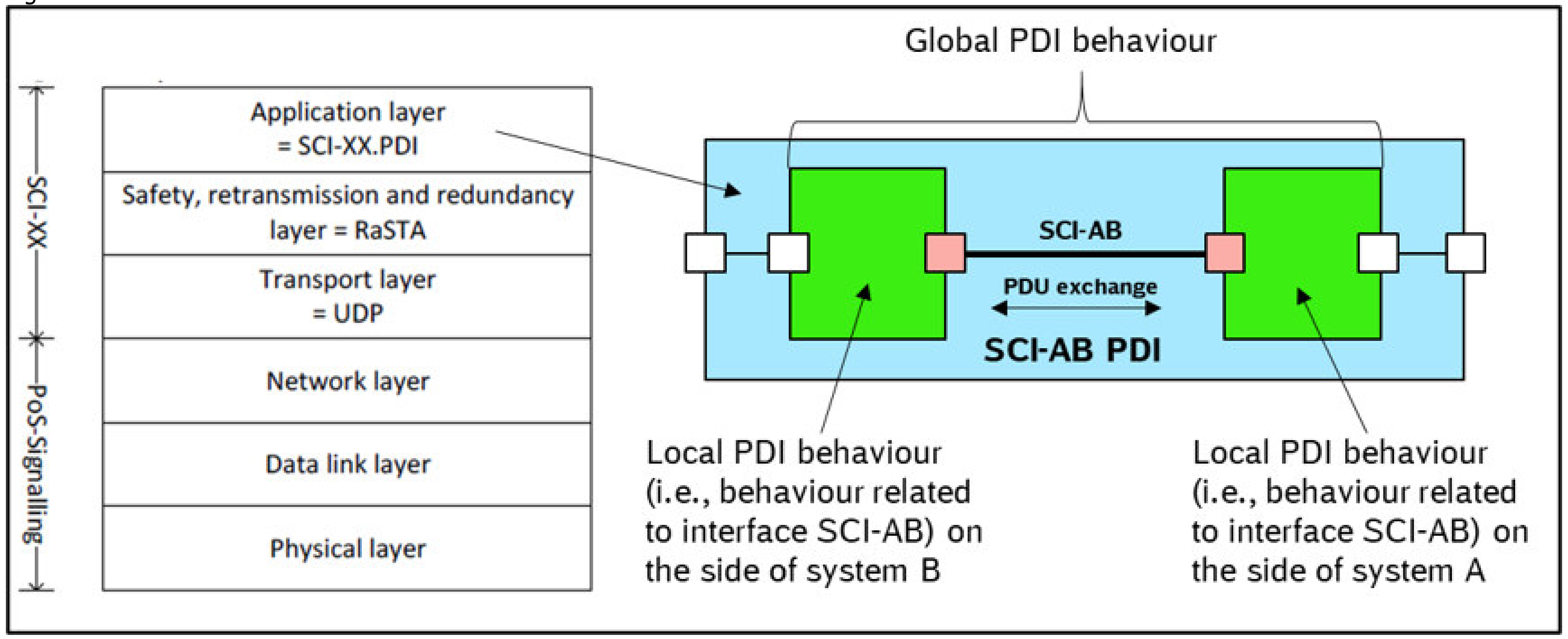
ID	Requirements
Eu.ModIn.102	If it can not be assumed that the current state of the SUS is visible in its environment, the textually formulated functional requirement **FSR1** is to be described as interaction without precondition. **FSR2** may be described as contract because the interaction is internally time-triggered and it is required that the current state may only be changed by this trigger: **FSR1:** **Precondition:** --- **Interaction:** **I.** - System receives the request "Button_Pressed" from the actor "Button". **II.** System evaluates that the request is valid because it is in state OFF. **III.** System changes to state "ON". **VI.** System responds to the actor "Light" with the command "Switch_Light_On". **Postcondition:** System is in state ON **FSR2:** **Precondition:** System is in state ON **Interaction:** **I.** - System detects that the time "t_Light_ON" has expired. **III.** System changes to state "OFF". **IV.** System responds to the actor "Light" with the command "Switch_Light_OFF". **Postcondition:** System is in state OFF
Eu.ModIn.364	The corresponding interaction scenario in the form of a Main Success Scenario is depicted in *Figure 6*.

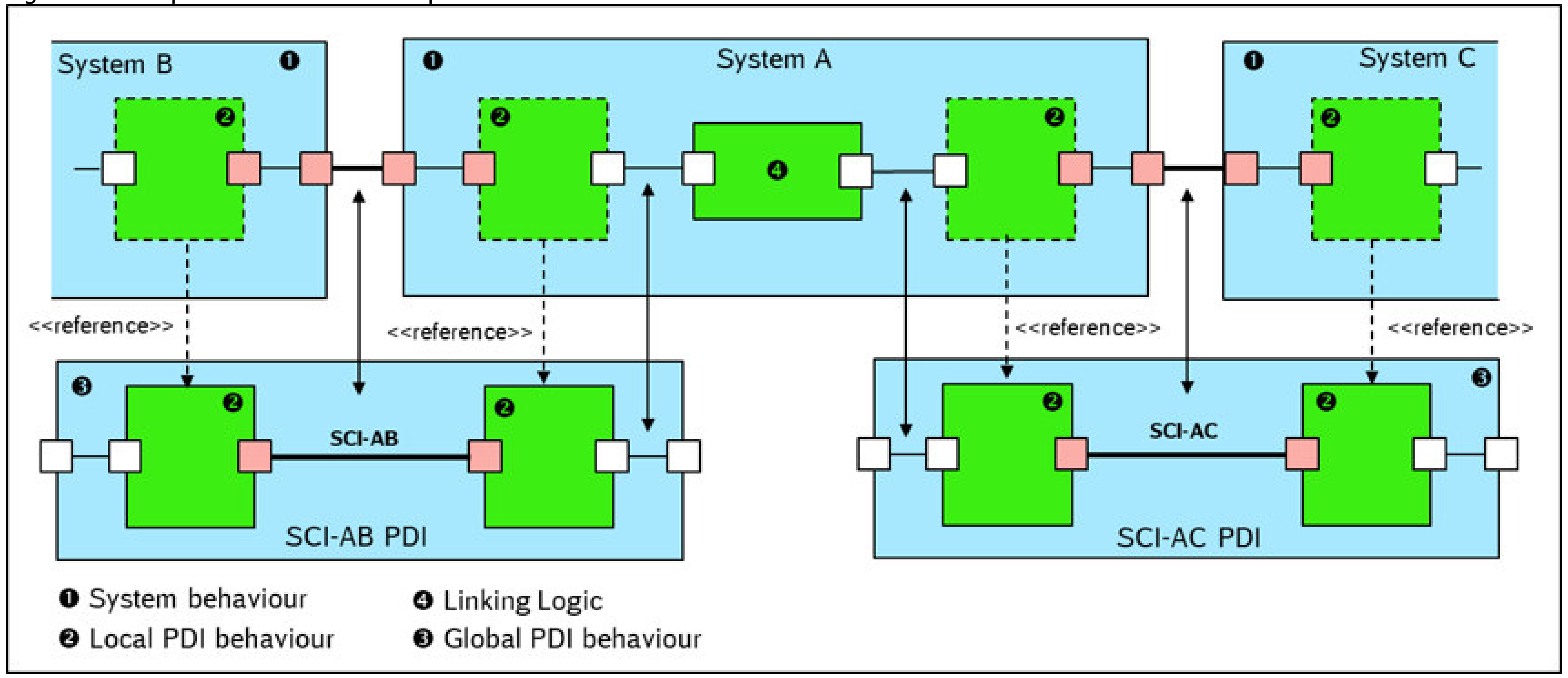
ID	Requirements
Eu.ModIn.365	Figure 6 Main Success Scenario with FSR1 not written as contract 
Eu.ModIn.375	As FSR1 is not written as a contract, action step 2 of the corresponding interaction may be evaluated as not valid. As a consequence, an alternative variant of the interaction has to be described: **FSR1:** **Precondition:** --- **Interaction:** **I.** - System receives the request "Button_Pressed" from the actor "Button". **III.** System evaluates that the request is not valid because it is in state ON. **IV.** System remains in state "ON". **Postcondition:** System is in state ON **FSR2:** **Precondition:** System is in state ON **Interaction:** **I.** - System detects that the time "t_Light_ON" has expired. **III.** System changes to state "OFF". **IV.** System responds to the actor "Light" with the command "Switch_Light_OFF". **Postcondition:** System is in state OFF

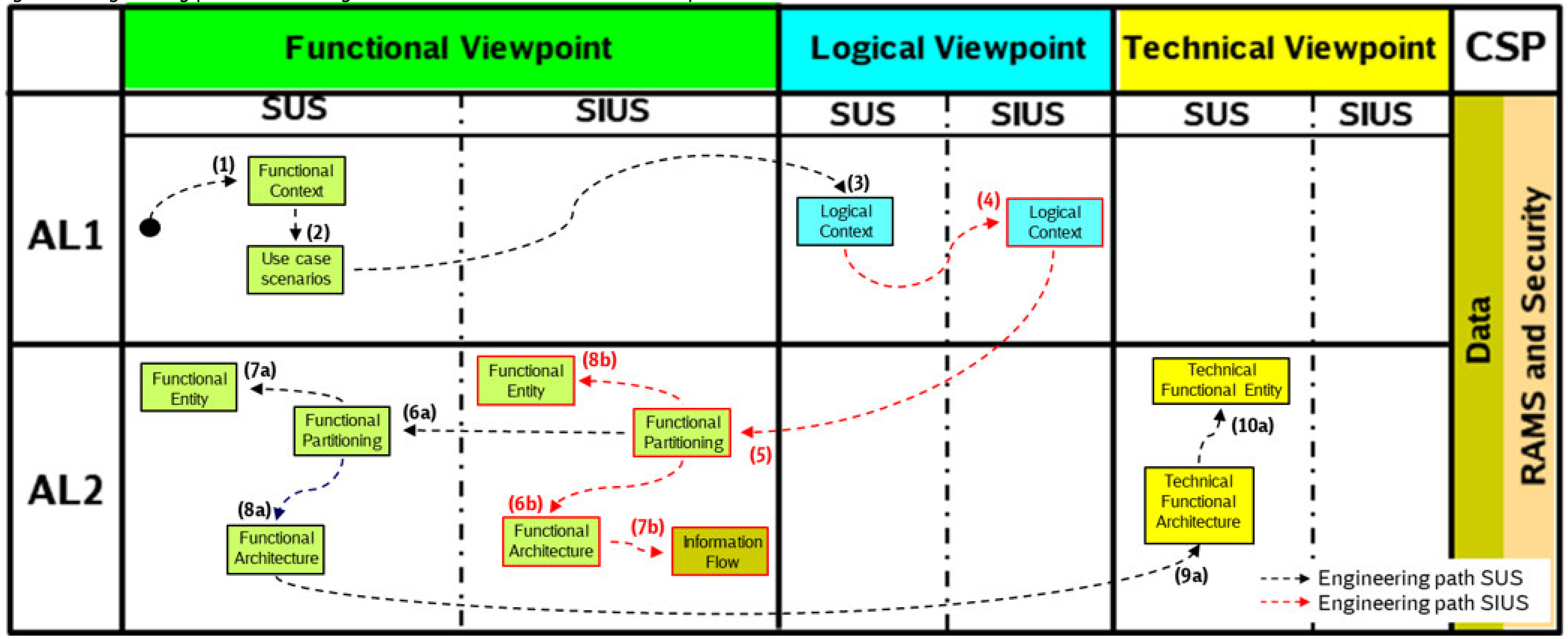
ID	Requirements
Eu.ModIn.366	The corresponding interaction scenario in the form of an Alternative Scenario is depicted in *Figure 7*.
Eu.ModIn.367	Figure 7 Alternative Scenario 
Eu.ModIn.95	**3.2.2 Description method using state machines**
Eu.ModIn.381	**State machines** are used at abstraction level **AL2 System Requirements** to completely refine the stimulus-response behaviour which has been described by means of the interaction scenarios at abstraction level **AL1 System Definition.**
Eu.ModIn.96	*Figure 8* shows a state machine specifying the stimulus-response behaviour of the UseCase "**SysUC1.1: Switch on the light time-limited**".

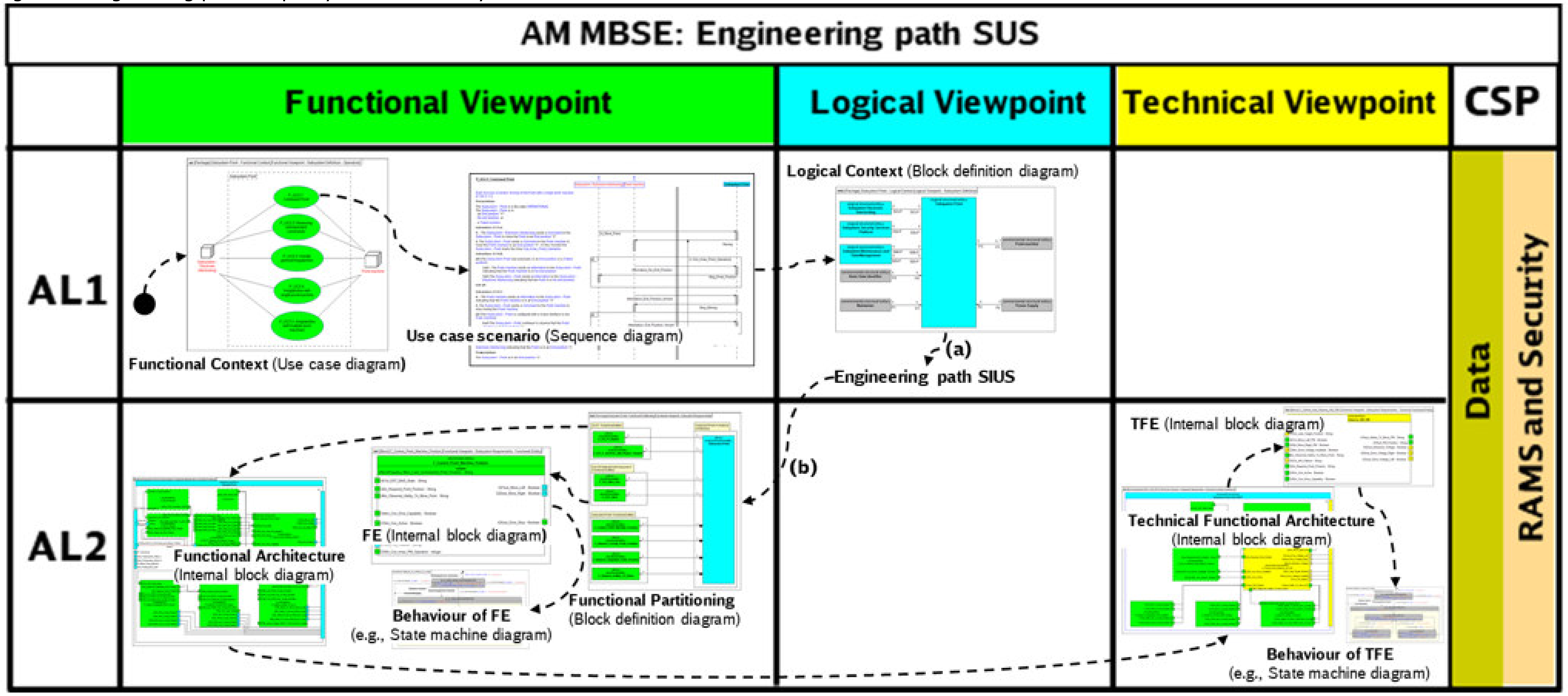
ID	Requirements
Eu.ModIn.369	Figure 8 FSR1 and FSR2 specified using a state machine 
Eu.ModIn.98	The declaration of this state machine is identical to the original textual requirements (see ID 93) **FSR1** (Transition from state "**OFF**" to state "**ON**") and **FSR2** (Transition from state "**ON**" to state "**OFF**"): **FSR1:** The system shall switch on the light ("**Switch_Light_On := TRUE**") if the light is switched off (state "**OFF**") and the button is pressed ("**when(Button_Pressed)**"). The **Transition from state "OFF" to state "ON"** represents a **functional system requirement** and may be textually formulated in the requirements specification document as shown below: **Info | OFF** **Req | when(Button_Pressed)/Switch_Light_On := TRUE {OFF - ON}** **Info | ON** **FSR2:** The system shall switch off the light ("**Switch_Light_OFF := TRUE**") automatically after the time t_Light_On has expired ("**after(t_Light_On)**"). The **Transition from state "ON" to state "OFF"** represents a **functional system requirement** and may be textually formulated in the requirements specification document as shown below: **Info | ON** **Req | after(t_Light_On)/Switch_Light_Off := TRUE {ON - OFF}** **Info | OFF**
Eu.ModIn.617	**3.3 Conventions**
Eu.ModIn.620	**3.3.1 General description of the model elements**
Eu.ModIn.621	**3.3.1.1 Logical Structural Entity (LSE)**
Eu.ModIn.630	A Logical Structural Entity (block in turquoise, stereotyped as <<logical structural entity>>) represents a system element from a logical point of view. It encapsulates either one or more LSEs interconnected in the form of a Logical Architecture or one or more FEs interconnected in the form of a Functional Architecture.
Eu.ModIn.641	Figure 9 Logical Structural Entity 
Eu.ModIn.622	**3.3.1.2 Functional Entity (FE)**

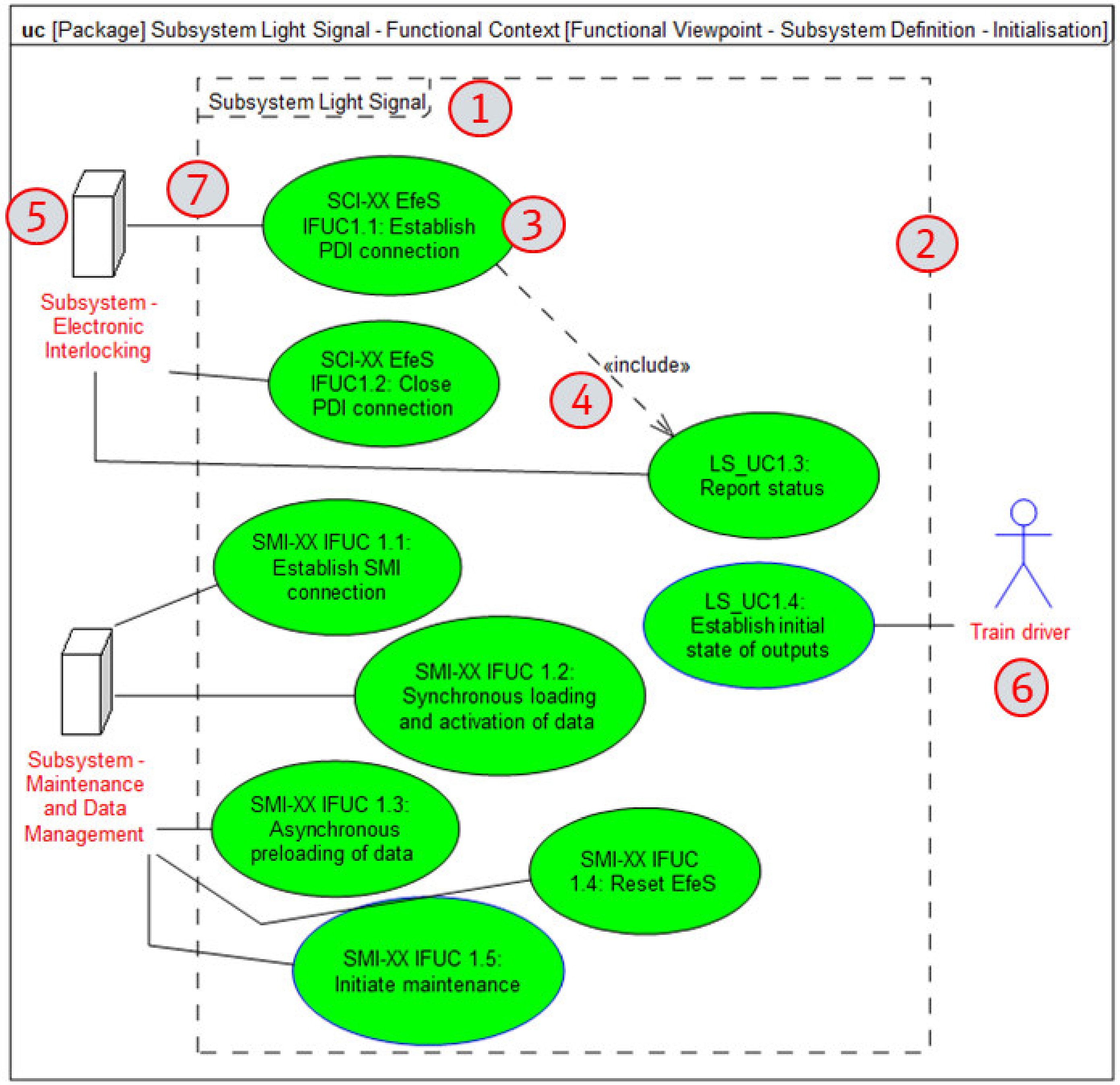
ID	Requirements
Eu.ModIn.631	A functional entity (green block, stereotyped with <<functional entity>>) encapsulates a certain portion of technology-independent system behaviour of a system element.
Eu.ModIn.640	A functional entity additionally stereotyped with <<assumption>>represents a set of assumptions which are not functional requirements. Assumptions are mainly used to restrict the environment of a FE.
Eu.ModIn.639	Figure 10 Functional Entity 
Eu.ModIn.623	**3.3.1.3 Environmental Structural Entity (ESE)**
Eu.ModIn.632	In the environment of a system element, there may be other system elements belonging to the same overall system (subsystems) with which the system element in question has a communication relationship. These system elements are described by logical structural entities. However, the system element can also have a relationship with system elements that are outside the associated overall system. These system elements are described by structural entities in the environment (gray block, stereotyped with <<environmental structural entity>>) represents.
Eu.ModIn.642	Figure 11 Environmental Structural Entity 
Eu.ModIn.728	**3.3.1.4 Technical Structural Entity (TSE) or Technical Functional Entity (TFE)**
Eu.ModIn.730	**Technical Structural Entity:** A Technical Structural Entity (yellow-coloured SysML block stereotyped with <<technical structural entity>>) encapsulates one or more TSEs in the form of a Technical Architecture or one or more TFEs interconnected in the form of a Technical Functional Architecture based on technical requirements (<<hardware>>: TSE representing a hardware artefact, <<software>>**:** TSE representing a software artefact).
Eu.ModIn.732	**Technical Functional Entity:** A Technical Functional Entity (yellow-coloured SysML block stereotyped with <<technical functional entity>>) represents a certain piece of technology-dependent behaviour based on technical requirements in a Technical Functional Architecture supplementing or substituting the technology-independent behaviour defined by FEs.
Eu.ModIn.731	Figure 12 Technical Structural Entity or Technical Functional Entity 
Eu.ModIn.624	**3.3.1.5 Information objects**
Eu.ModIn.633	Information objects are the objects that are exchanged between the respective communication partners via a communication relationship. They are formed from signals and values of the signals, the so-called attributes and are made available or received at ports.

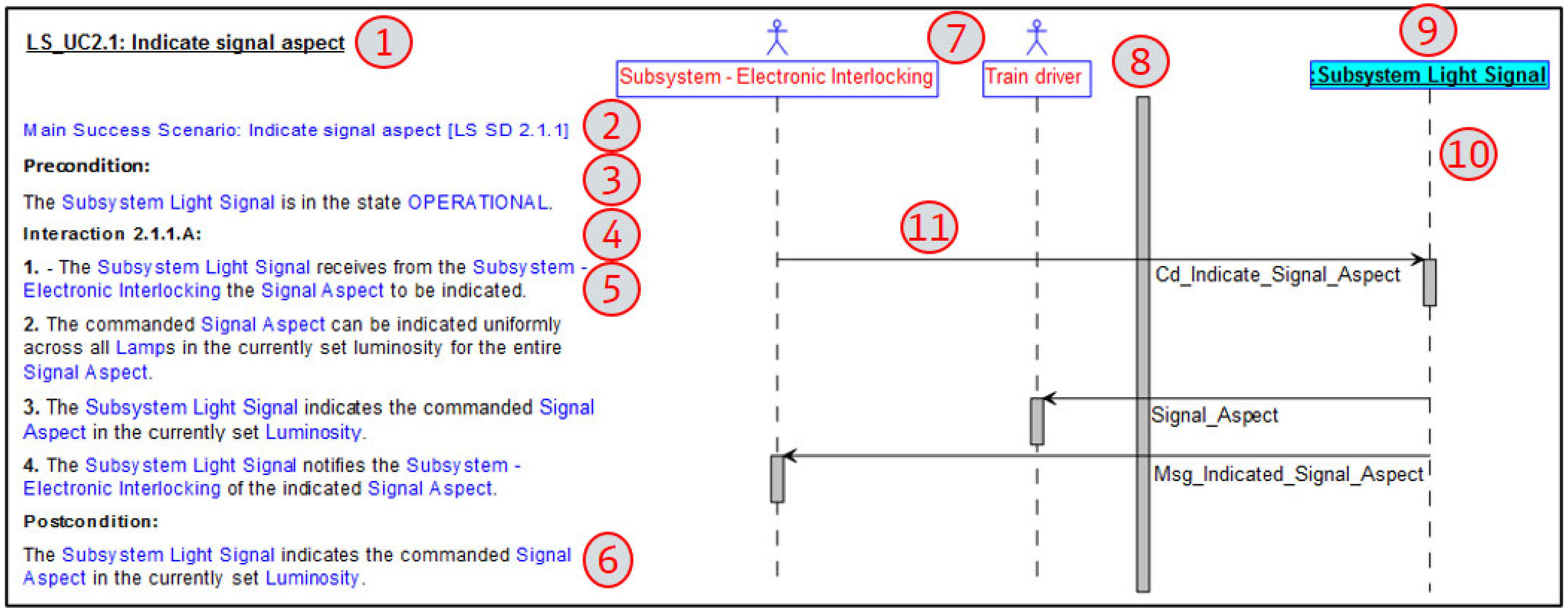
ID	Requirements
Eu.ModIn.634	Ports are represented by small squares at the edge of a Functional Entity and represent the connections to the interfaces to other internal or external Functional Entities to which a communication relationship exists, or to external interfaces. The port also indicates the arbitrary port name and interface type in the format "port name:interface type". Communication relationships between functional entities are assigned a reading direction. In the case of ports, this is represented by the interface type being shown in conjugated form, i.e. by the symbol "~", on one side of the communication relationship.
Eu.ModIn.643	**3.4 Interface centric specification**
Eu.ModIn.368	The EULYNX initiative is aiming at specifying EULYNX system elements and standardising the communication interfaces (SCI) between them.
Eu.ModIn.644	As the focus is on the specification of interfaces, the behaviours of EULYNX systems are specified using an interface centric approach.
Eu.ModIn.645	An interface centric approach is understood that the external visible stimulus-response behaviour (usage behaviour) of a system is largely described by the behaviours related to its interfaces. These behaviours are linked together and supplemented by behaviour relevant for more than one interface by means of linking behaviour.
Eu.ModIn.647	In the EULYNX specification approach, the models of the protocol stacks assigned to the communication interfaces are downscaled to the Process Data Interface protocols (PDI) defining the global PDI behaviours of the application layers (e.g., SCI-AB PDI).
Eu.ModIn.648	Global behaviour specifies the dependencies between the local PDI behaviours of the communication partners, that is the exchange of Process Data Units (PDU) between them in a chronological order.
Eu.ModIn.649	The local PDI behaviours represent the behaviours of the communicating systems related to a certain interface.
Eu.ModIn.650	The relation between local PDI behaviour and global PDI behaviour can be illustrated by a telephone call. The dialling is a local PDI behaviour at the initiator side, the ringing the associated local PDI behaviour at the partner side. Only the global PDI behaviour defines that the dialling must precede the ringing (i.e., the chronological order).
Eu.ModIn.646	Figure 13 Global PDI behaviour 
Eu.ModIn.382	As the local PDI behaviours represent the interface behaviours of the communicating systems they may be specified in the model of the PDI.
Eu.ModIn.651	In the model of a system element such as System A, these local PDI behaviours are referenced and linked together.

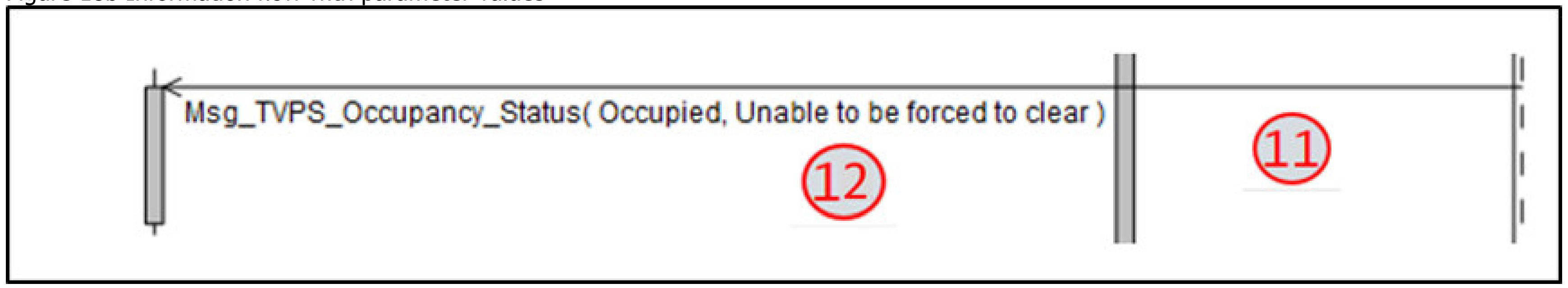
ID	Requirements
Eu.ModIn.216	Figure 14 Principle of interface centric specification 
Eu.ModIn.653	In the following chapters the model views used to specify EULYNX system elements (*chapter 4*) and the ones used to define EULYNX interfaces (chapter 5) are introduced. As the model view "Functional Entity" is used for the specification of EULYNX subsystems as well as for the specification of EULYNX interfaces it is described in the separate *chapter 6*.
Eu.ModIn.849	**3.5 Functional packages**
Eu.ModIn.850	The EULYNX specifications are divided into functional packages in the requirements management tool used. This is intended to enable Infrastructure Managers (IM) involved to select requirements in a targeted manner and thus apply the specifications to the desired capabilities of their products.
Eu.ModIn.851	There are two types of packages that relate to product capabilities: • 'Basic packages', i.e. one or more packes, at least one of them must be implemented. It is allowed to combine and implement more than one 'basic package' in a product. • 'Optional package', i.e. one or more packages that can be optionally implemented in addition to one or more basic packages.
Eu.ModIn.852	For the evaluation if a requirement is valid or not depending on the selected functional packages of an IM, the basic packages have an "or" relation and optional packages have an "and" relation to everything else. I.e. from mathematical point of view: ("Basic P1" or "Basic P2" or "Basic Pn") and "Option P1".
Eu.ModIn.853	The functional packages are allocated to the requirements in the requirements management tool used. The practical implementation of the allocation depends on the capabilities of the tool.
Eu.ModIn.854	The SysML specification model is structured in such a way that the required functional packages can be separated from the overall functionality in order to enable clear allocation as described above.
Eu.ModIn.855	For example, functional packages can be formed by encapsulating certain behaviours in functional entities, which are then used or not in the corresponding functional architecture as required.
Eu.ModIn.783	**3.6 Overview of the engineering paths to create EULYNY specification models**
Eu.ModIn.784	*Figure 48* shows the commonly used engineering paths for creating the model views of the SUS or SIUS specification models (see also *chapter 8.1.3* of Eu.Doc.30), which are explained in more detail in the following *chapters 4, 5 and 6*. Depending on the project-specific input conditions, the engineering paths can also be applied in a modified form.
Eu.ModIn.785	In general, the engineering path for creating the SUS model views (black dashed arrows) includes the engineering path for creating the SIUS model views (red dashed arrows).
Eu.ModIn.786	The model views used reflect the current state of the EULYNX MBSE methodology and may be complemented by further model views in the future (e.g. model views of the Technical Viewpoint or model views on AL3).
Eu.ModIn.787	The engineering path for creating the SUS model views starts at the Functional Viewpoint on abstraction level AL1.

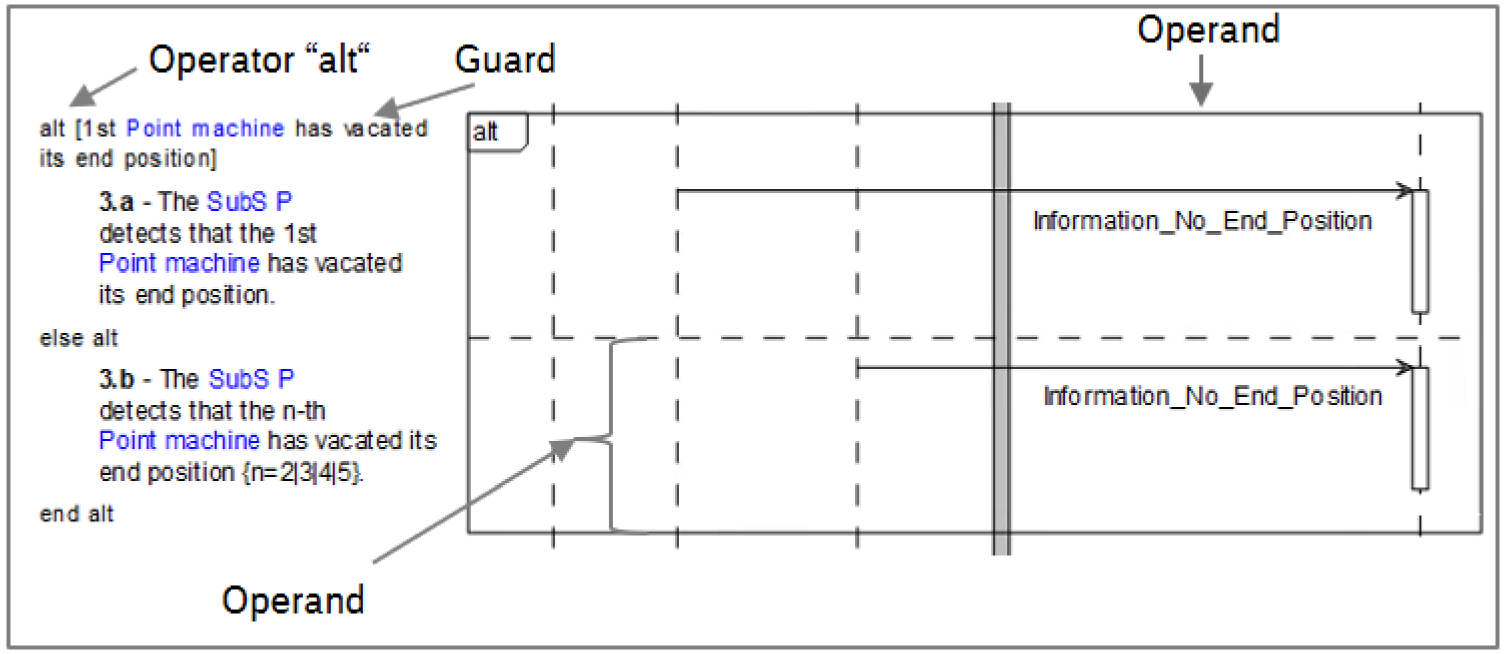
ID	Requirements
Eu.ModIn.788	Figure 48 Engineering paths for creating the model views of the SUS or SIUS specification models 
Eu.ModIn.248	**4 Model views used to specify EULYNX subsystems**
Eu.ModIn.286	**Model view "Functional Context": Use Cases (uc)** The model view "Functional Context" defines the services to be provided by the SUS in the form of use cases. Relationships are used to represent which actors interact with which SUS use case.
Eu.ModIn.657	**Model view "Use case scenario": Sequence Diagram (sd)** The model view "Use case scenario" describes the behaviour of the use cases defined in the model view "Functional Context" by means of one or more use case scenarios.
Eu.ModIn.45	**Model view "Logical Context": Block Definition Diagram (bdd)** The model view "Logical Context" describes in the form of a block definition diagram (bdd) at the top level • the system/subsystem under specification (SUS), • the actors in the environment interacting with the SUS and their quantity structure (multiplicities) as well as the logical interfaces between the SUS and the actors.
Eu.ModIn.743	**Model view "Functional Partitioning": Block Definition Diagram (bdd)** The model view "Functional Partitioning" describes the refinement of the SUS by means of the FEs defined in the SIUS model view "Functional Partitioning", which represent the local behaviours of the PDI, as well as the FEs specific to the SUS (linking behaviour according to *chapter 3.4*).
Eu.ModIn.658	**Model view "Functional Architecture": Internal Block Diagram (ibd)** The model view "Functional Architecture" refines or completes the behaviour of an SUS defined in the model view "Use case scenarios". The behaviour of the SUS is divided into Functional Entities" (FE), which communicate with each other via internal interfaces and with the environment via external interfaces. The FEs are defined in model view "Functional Partitioning".
Eu.ModIn.744	**Model view "Technical Functional Architecture": Internal Block Diagram (ibd)** The model view "Technical Functional Architecture" supplements the behaviour described in the model view "Functional Architecture", which is independent of technology, with behavioural components derived from technical requirements. Either the entire behaviour can be described in a technical context or a mixture of functional and technical aspects.

ID	Requirements
Eu.ModIn.659	**Model views "Functional Entity" and "Technical Functional Entity": Internal Block Diagram (ibd) and State Machine (stm)** The model view "Functional Entity" encapsulates a subset of technology-independent functional requirements and the model view "Technical Functional Entity" a subset of technology-dependent functional requirements of a SUS in the form of a function module. It delimits the function module from its environment and defines the inputs and outputs. In the discrete case, the behaviour of the FE is described by means of state machines. In this, the binding functional requirements are specified in the form of state transitions. Both model views are described in the separate *chapter 6*.
Eu.ModIn.660	*Figure 15* shows the engineering path of the model views used to specify a SUS considering the Functional Viewpoint, the Logical Viewpoint and the Technical Viewpoint. It describes the context of the model views, with the arrows indicating which model views are developed from which. During the development of the model, the model views "Functional Context" (the Use Cases), "Use case scenarios" and "Logical Context" are created. These model views form the basis for the description of the model views "Functional Partitioning", "Functional Architecture" and "Functional Entity". For the creation of the model view "Functional Partitioning", the FEs defined in the model view "Functional Partitioning" of the SIUS are required (b: see *Figure 26* in *chapter 5*). In case technical requirements are to be considered, the model views "Technical Functional Architecture" and "Technical Functional Entity" are created based on the model view "Functional Architecture".
Eu.ModIn.287	Figure 15 Engineering path to specify a EULYNX subsystem 
Eu.ModIn.249	**4.1 Abstraction Level AL1: System Definition**
Eu.ModIn.662	**4.1.1 Model view "Functional Context" of a SUS**
Eu.ModIn.168	The model view "Functional Context" as shown in *Figure 17* defines the services to be provided by the SUS in the form of use cases. On one or more SysML UseCase diagrams all subsystem UseCases and their relationships to the SUS environment and between the subsystem UseCases themselves are depicted.
Eu.ModIn.169	In the use case diagrams, the boundary **(2)** of the SUS **(1)** is shown as a frame with a dotted line.
Eu.ModIn.323	The use cases of the SUS are shown as ellipses within the frame and have the name of the respective use case **(3)**.
Eu.ModIn.166	A use case describes a service a SUS provides to its environment and is specified by one or more interaction scenarios (model view "Use case scenario").
Eu.ModIn.71	Use cases are connected by interaction connectors **(7)** to those actors in the SUS environment with whom they interact. An actor may represent another system **(5)** or a person **(6)**.
Eu.ModIn.170	Use cases may be connected to each other through include relationships **(4)**, which are represented by arrows with a dashed line stereotyped with <<include>>. Such a relationship indicates that the interaction scenarios of the use case at the arrowhead are included in the use case at the other end of the arrow. These included use cases encapsulate services that occur more than once, for example, and can also be included in other use cases.

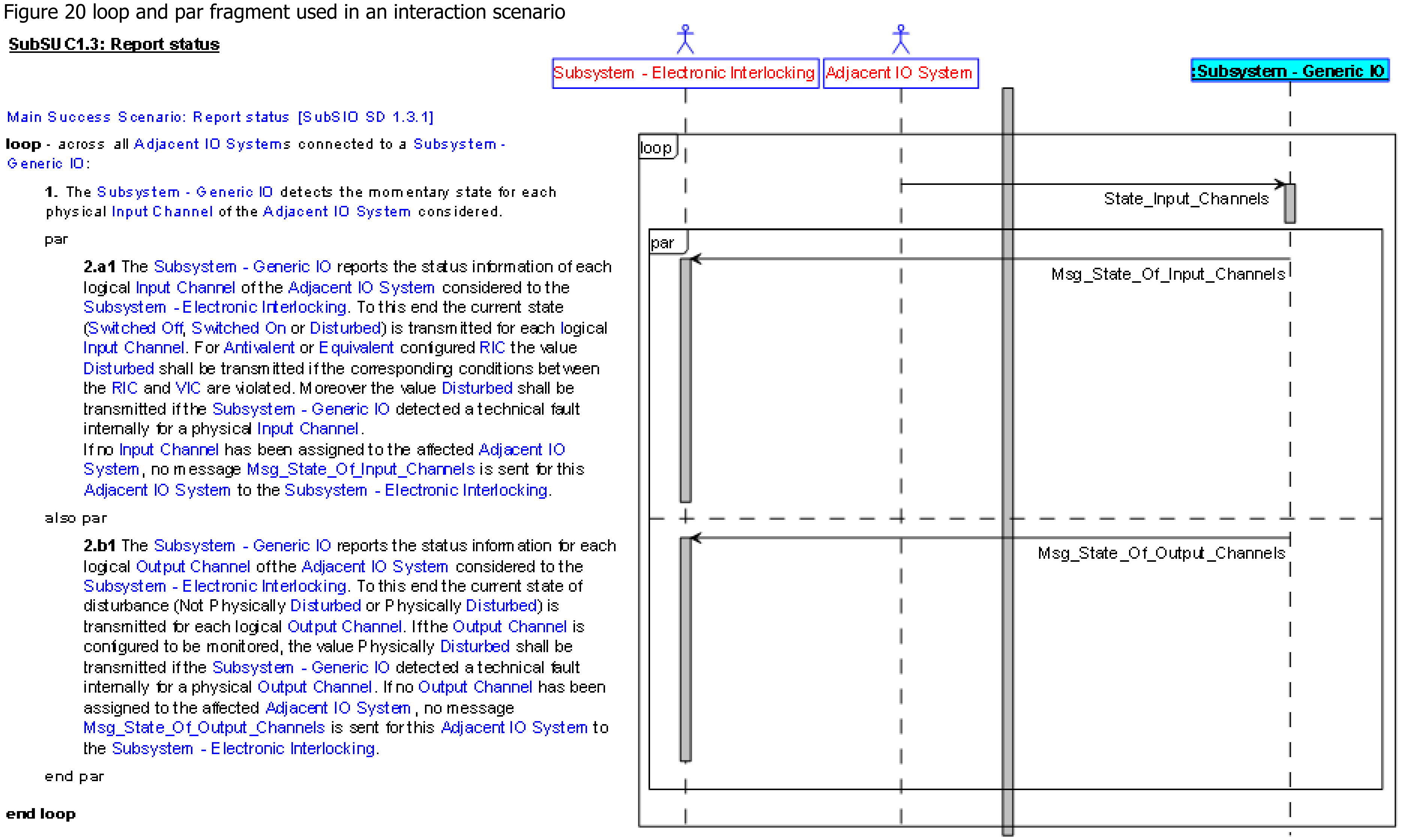
ID	Requirements
Eu.ModIn.326	Figure 17 Functional Context 
Eu.ModIn.790	**4.1.1.1 Binding** (see *chapter 2.1*)
Eu.ModIn.791	**Diagram of model view "Functional Context"** has an "**Info**" binding.
Eu.ModIn.793	**Use Case** has an "**Info**" binding if it is further specified in a refined model view.
Eu.ModIn.794	**Use Case** has a "**Req**" binding if it is not further specified in a refined model view.
Eu.ModIn.163	**4.1.2 Model view "Use case scenario" of a SUS**
Eu.ModIn.164	The model view "Use case scenario" as shown in *figure 18a* defines the behaviour of the use cases defined in the model view "Functional Context" by means of one or more interaction scenarios at the upper level of abstraction. These interaction scenarios describe the interaction between the SUS and the actors in the SUS environment using SysML sequence diagrams.

ID	Requirements
Eu.ModIn.325	Figure 18a Use case scenario 
Eu.ModIn.167	**4.1.2.1 Use case name**
Eu.ModIn.324	Name of the use case **(1)** to which the interaction scenario belongs (e.g., LS_UC2.1: Indicate signal aspect).
Eu.ModIn.173	**4.1.2.2 Use case scenario name**
Eu.ModIn.165	The use case scenario name **(2)** is the name of a possible information flow (shown as a sequence diagram) within a use case (Main Success Scenario or Alternative Scenario).
Eu.ModIn.322	**4.1.2.3 Preconditions**
Eu.ModIn.174	Preconditions **(3)** are conditions that must be met and known to the actor triggering the stimulus for the scenario to start (see chapter 3.2.1).
Eu.ModIn.177	**4.1.2.4 Interaction**
Eu.ModIn.388	An interaction **(4)** consists of a sequence of steps, starting with a stimulus (prefixed by a dash "-"), a validation, possibly a state change and a reaction. In addition, combined fragments may be included. A use case scenario can consist of one or more interactions. The structure of an interaction follows the principle of the Action Block Scheme as described in chapter 3.1.
Eu.ModIn.663	**4.1.2.5 Sequences and information flows**
Eu.ModIn.179	Sequences consist of a text part describing the sequence **(5)** and, in the case of an information flow, a graphical representation of the information flow in the form of arrows between the lifelines **(11)**. In the text part, elements of the model are shown in blue and explanatory text in black. In the graphical part, the corresponding exchange of information objects is shown accordingly. Here in the example (sequence 1), the information object "Cd_Indicate_Signal_Aspect" is sent from the "Subsystem Electronic Interlocking" to "Subsystem Light_Signal". As it is a stimulus it is prefixed by a dash "-" in the text part of the sequence. In sequence 2, the validation of the information object in the "Subsystem Light Signal" is described in the text part, without representation in the graphical part.
Eu.ModIn.847	In order to increase comprehensibility, the values of the parameters **(12)** transmitted with the respective information flows **(11)** are sometimes shown, as for example in *figure 18b*. However, this is not generally to be expected for all information flows.

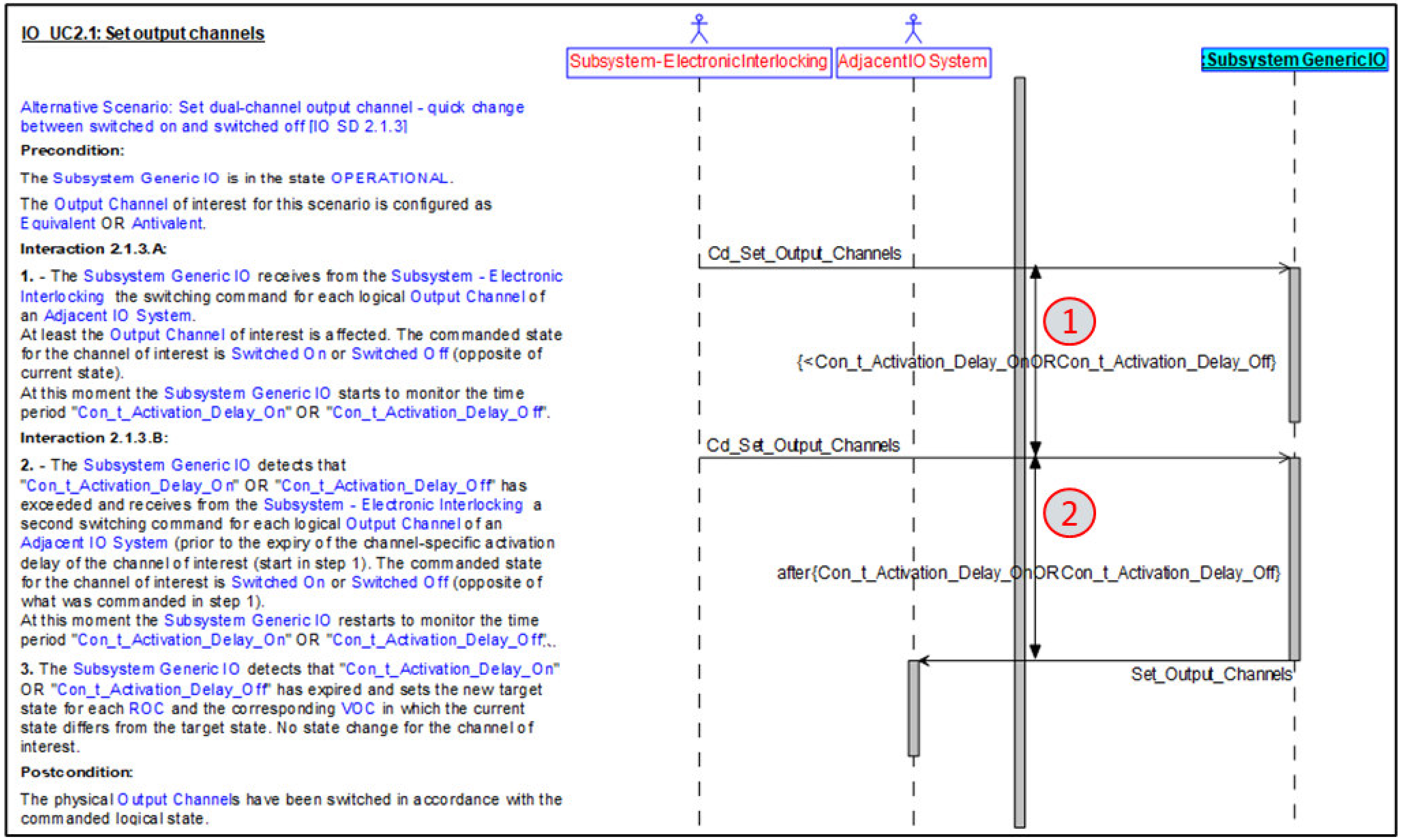
ID	Requirements
Eu.ModIn.846	Figure 18b Information flow with parameter values 
Eu.ModIn.664	**4.1.2.6 Postconditions**
Eu.ModIn.181	Postconditions **(6)** are conditions for which changes have resulted from the sequence diagram. Conditions that have already been mentioned in the preconditions are not listed here.
Eu.ModIn.665	**4.1.2.7 Actors**
Eu.ModIn.180	Actors **(7)** are systems (e.g., Subsystem Electronic Interlocking) or persons that interact with the SUS, i.e. trigger a stimulus and/or receive a response.
Eu.ModIn.666	**4.1.2.8 System under specification and System boundary**
Eu.ModIn.389	The boundary between the system under specification (SUS) and the actors is symbolised by a thick grey bar **(8)**. The SUS **(9)** is located to the right of the grey bar and the actors **(7)** to the left.
Eu.ModIn.667	**4.1.2.9 Lifelines**
Eu.ModIn.335	Lifelines **(10)** represent the time axis of the SUS and the actors, with the time running from top to bottom.
Eu.ModIn.186	**4.1.2.10 Combined fragments**
Eu.ModIn.187	Using so-called "combined fragments", it is possible to modify the normally strictly defined sequence of information flows in the interaction scenarios. Combined fragments are shown by a box around the information flows concerned (**Operand**) and by a corresponding indentation of the specification texts in the left-hand area. A parameter in the top left corner of the box (**interaction operator**) and as a key word in the text indicate the type of the combined fragment (see *figure 19*).
Eu.ModIn.393	An **operand** may have a **guard** containing a constraint expression that indicates the conditions under which it is valid for the operand to begin execution (see the example depicted in *figure 19*).
Eu.ModIn.195	**4.1.2.10.1 alt - alternative sequence**
Eu.ModIn.196	The **alt** fragment defines at least two (but possibly more) **operands** in the sequence diagram demarcated by dotted lines and the key words **"alt"**, **"else alt"** and **"end alt"** . Each area can include several information flows. The meaning of this fragment is that only one of the areas is run through in the sequence dependent on the defined conditions. This allows for different sequences to be mapped in an interaction scenario.

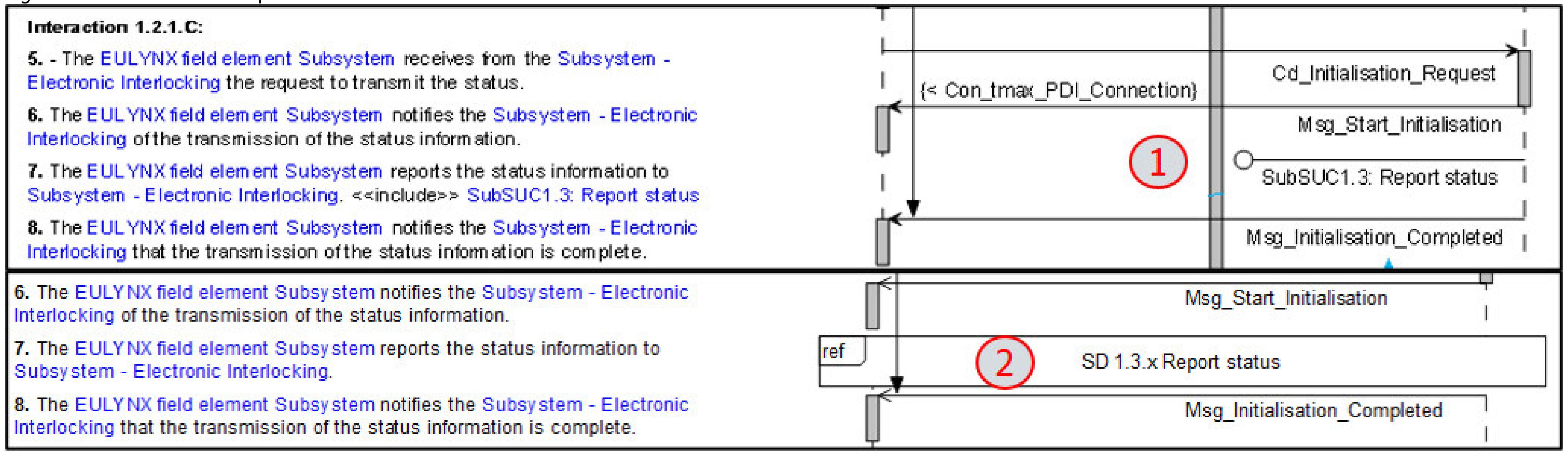
ID	Requirements
Eu.ModIn.208	Figure 19 alt fragment in a sequence diagram 
Eu.ModIn.391	**4.1.2.10.2 opt - optional sequence**
Eu.ModIn.392	The **opt** fragment is equivalent to an **alt** fragment with only one **operand**. This implies that the operand is either executed or skipped depending on the validity of the **guard** (condition).
Eu.ModIn.188	**4.1.2.10.3 par - Parallelism**
Eu.ModIn.189	Parallelising of information flows: The **par** fragment (see *figure 20*) consists of at least two (but possibly more) areas (Operands) demarcated by the key words **"par"**, "**also par**" and **"end par"**. Each area can cover several information flows.
Eu.ModIn.190	The meaning of this fragment [as a requirement] is that the information flows within an area must take place in the shown order but no order is specified between the areas. If there are two areas A and B with the information flows A1, A2 and B1, B2 respectively, then information flow A1 must always be followed by A2 and B1 always by B2. However, whether (A1 and A2) flow first or (B1 and B2) is not specified by the sequence diagram.
Eu.ModIn.191	**4.1.2.10.4 Loop**
Eu.ModIn.192	The **loop** fragment (see *figure 20*) defines that the information flows contained are transmitted several times consecutively in the order specified. The textual specification area must define how often the loop is run through. This may be a concrete number specification (loop - n times) or an implied specification via a volume or cancellation criterion (loop - For all messages present).

ID	Requirements
Eu.ModIn.231	Figure 20 loop and par fragment used in an interaction scenario

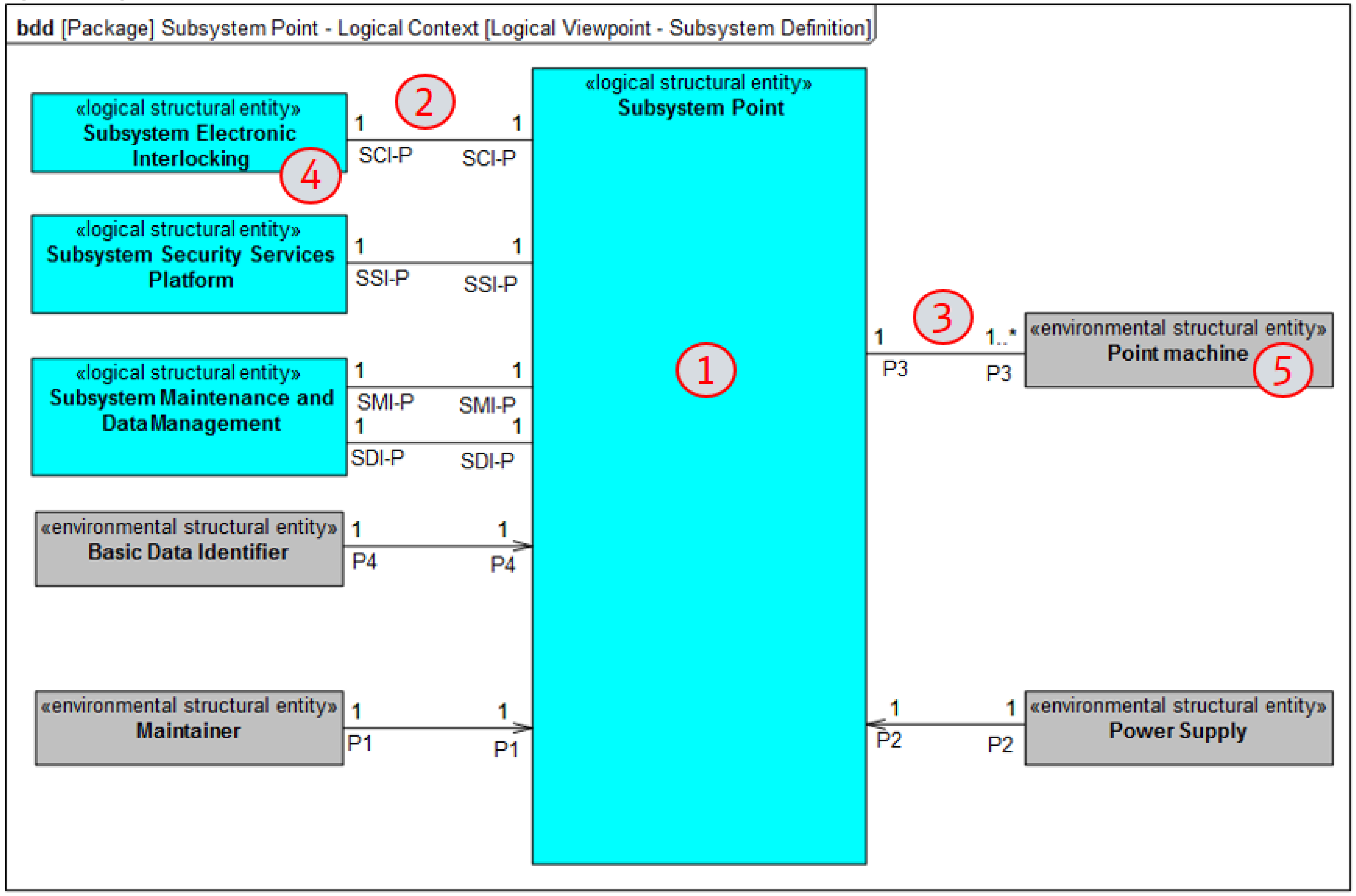


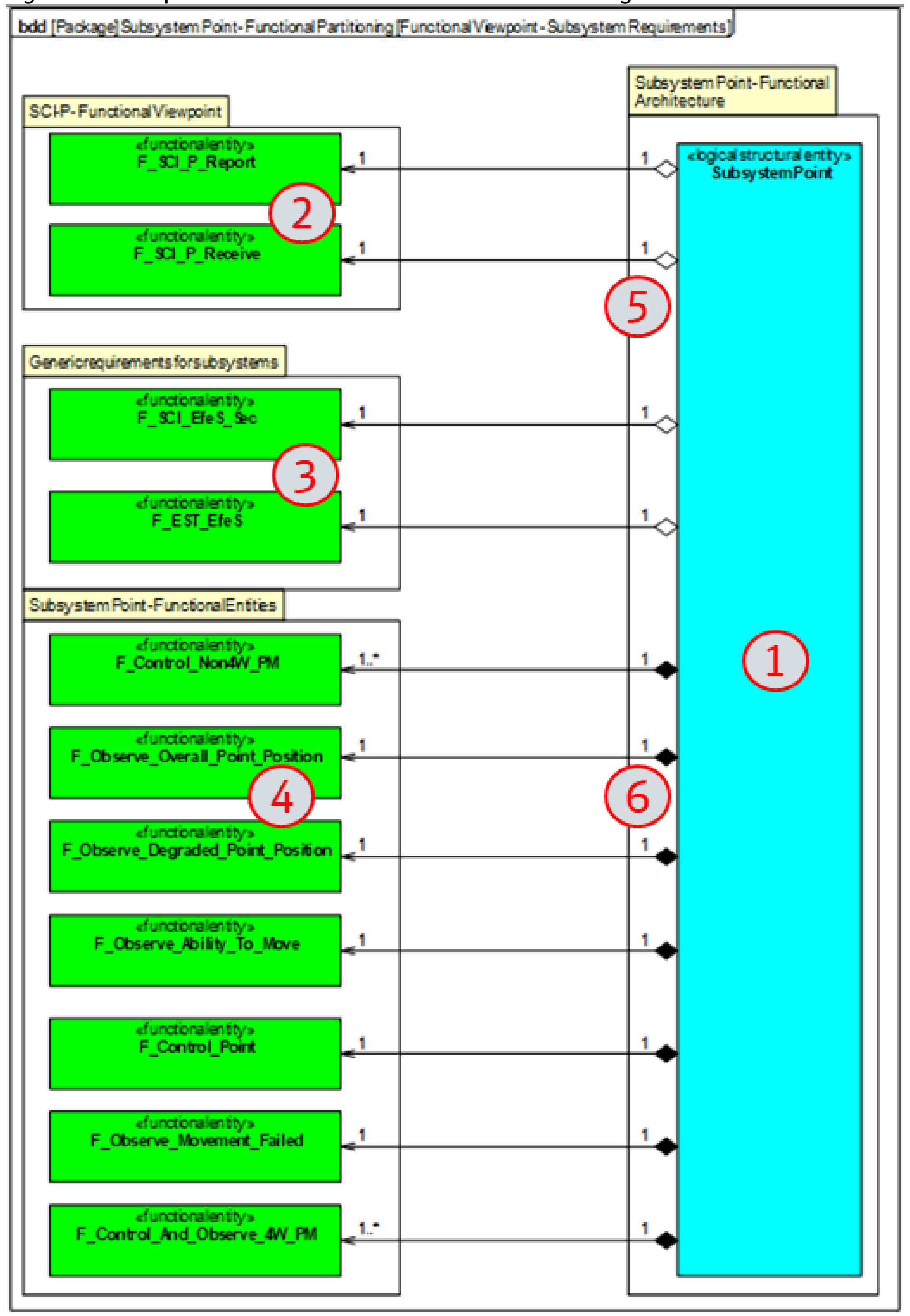
ID	Requirements
Eu.ModIn.386	**4.1.2.11 Representing time in an interaction scenario**
Eu.ModIn.183	As depicted in *figure 21* time may be represented in interaction scenarios as duration constraints **(1)** and timed triggers **(2)**.

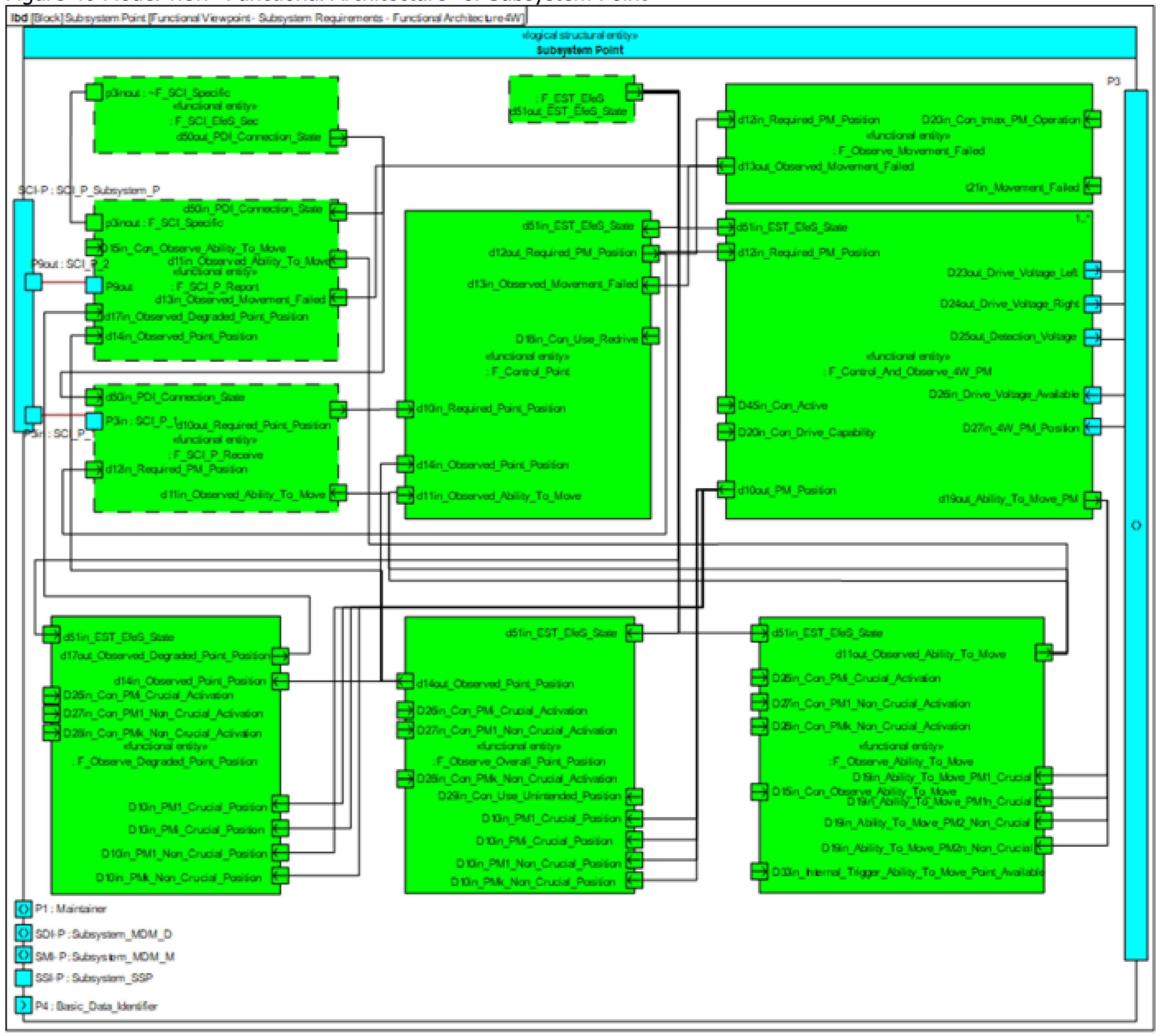
<table>
<tr><th>ID</th><th>Requirements</th></tr>
<tr><td>Eu.ModIn.304</td><td>Figure 21 Representing time

</td></tr>
<tr><td>Eu.ModIn.394</td><td>4.1.2.11.1 Duration constraints</td></tr>
<tr><td>Eu.ModIn.184</td><td>Time conditions (1) can also be mapped in the sequence diagram. To this end, a time limit is specified between two information flows consisting of a vertical double arrow. This is supplemented by a duration constraint, e.g., {>X s} or {<= X s}. Interpreted as a requirement, this means that two sequences (e.g., sequence 1 and sequence 2 in figure 21) may or must follow each other within the minimum or maximum time specified (e.g., <Con_t_Activation_Delay in Figure 21).</td></tr>
<tr><td>Eu.ModIn.395</td><td>4.1.2.11.2 Timed trigger</td></tr>
<tr><td>Eu.ModIn.396</td><td>A timed trigger (2) indicates that a given time interval has passed since the occurrence of some event, such as entering a state or receiving a request during the execution of the scenario.</td></tr>
<tr><td>Eu.ModIn.397</td><td>The term "after" followed by the time such as "after {10 sec}" indicates that the time is relative to the moment of an occurrence.</td></tr>
<tr><td>Eu.ModIn.398</td><td>An example of a timed trigger is shown in the scenario depicted in figure 21. "Subsystem Generic IO" responses to the stimuli "Cd_Set_Output_Channels" with "Set_Output_Channels" after the time "Con_t_Activation_Delay_On" or "Con_t_Activation_Delay_Off" has expired.</td></tr>
<tr><td>Eu.ModIn.856</td><td>The graphical representation of time behaviour as shown in figure 21 may be supplemented by a description in the description area of the sequences.
"Con_t_Activation_Delay_On" or "Con_t_Activation_Delay_Off" represent the defined time period (duration):
• Start of timer should be mentioned within the corresponding step (trigger).
  • "Subsystem X starts to monitor the time period "Con_t_Activation_Delay_On" or "Con_t_Activation_Delay_Off"."
• Reaction for timer that shall be waited for --> where possible combine within corresponding step otherwise keep it separate.</td></tr>
</table>

ID	Requirements
	• "Subsystem X detects that time period "Con_t_Activation_Delay_On" or "Con_t_Activation_Delay_Off" has expired." • Reaction for timer that has been exceeded (unintended case) --> where possible combine within corresponding step otherwise keep it separate. • "Subsystem X detects that time period "Con_t_Activation_Delay_On" or "Con_t_Activation_Delay_Off" has exceeded." • Restart of a timer within the corresponding step (trigger). • "Subsystem X stops to monitor time period "Con_t_Activation_Delay_On" or "Con_t_Activation_Delay_Off" caused by first command and starts to monitor the time period "Con_t_Activation_Delay_On" or "Con_t_Activation_Delay_Off" caused by second command." • Reset of a timer within the corresponding step (trigger). • "Subsystem X stops to monitor time period "Con_t_Activation_Delay_On" or "Con_t_Activation_Delay_Off"."
Eu.ModIn.857	**Time periods** are defined without further specification of the values. The values to be used are specified separately in the requirements management tool (chapter 5.3 Configuration and engineering data) as binding requirements and linked to the corresponding definitions.
Eu.ModIn.390	**4.1.2.12 Include relationship**
Eu.ModIn.185	As shown in *figure 22* an <<include>> relationship can be used to jump from an interaction scenario to the interaction scenario of an included use case (e.g., SubSUC1.3: Report status). The text part and the include symbol **(1)** indicate which use case is to be accessed. After processing the included interaction scenario, the original interaction scenario is continued.
Eu.ModIn.182	Alternatively to the include symbol **(1)** an "interaction use" **(2)** may be used to indicate which included interaction scenario is to be accessed. "Interaction uses" are shown as frames with the keyword "ref" in the frame label. The body of the frame contains the name of the referenced interaction scenario.
Eu.ModIn.858	For each SD that is referenced in another SD by an "Interaction use", a note is inserted in "Requirements Part 2" of the specification document that corresponds to the following defined schema: • This SD is part of [referred SD].
Eu.ModIn.228	Figure 22 Include relationship in interaction scenarios 
Eu.ModIn.798	**4.1.2.13 Binding** (see *chapter 2.1*)
Eu.ModIn.799	**Diagram of model view "Use case scenario"** has an "**Info**" binding if it is further specified in a refined model view (e.g. through a state machine).
Eu.ModIn.859	**Diagram of model view "Use case scenario"** has a "**Req**" binding if it is not further specified in a refined model view.
Eu.ModIn.860	The definitions of **time periodes** (e.g. Con_tmax_PDI_Connection) have "**Def**" bindings.
Eu.ModIn.861	The **values of the defined time periods**, which are specified, have "**Req**" bindings.
Eu.ModIn.48	**4.1.3 Model view "Logical Context" of a SUS**

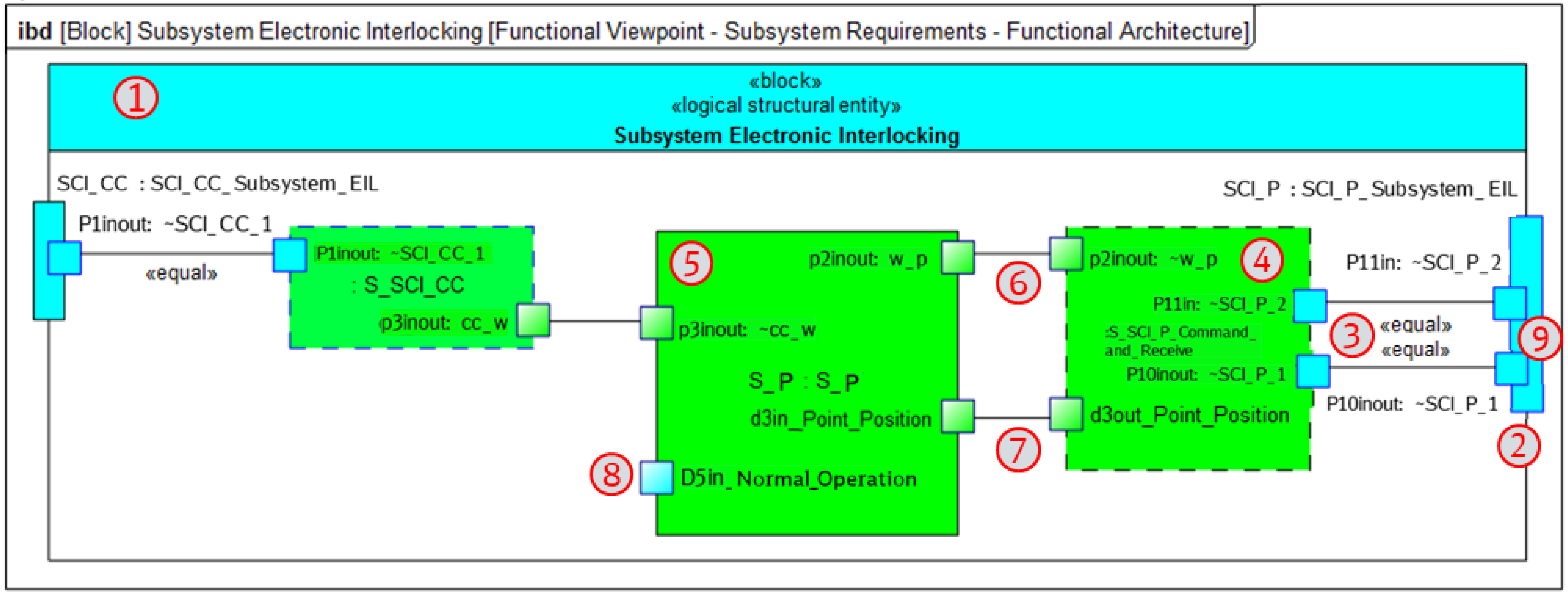
ID	Requirements
Eu.ModIn.50	The model view "Logical Context" as shown in *Figure 16* represents the environment of the SUS and provides initial information about the SUS boundaries and the relationships to the interaction partners. This diagram contains the following definitions relevant to implementation: • **Interaction partners:** the representation of the interaction partners as actors with whom the SUS concerned must be able to interact, • **Logical SUS interfaces:** - number of required logical interfaces represented by associations to interaction partners in the SUS environment defined by means of multiplicities at the association ends - possible directions of the interaction (uni- or bidirectional). - kinds of interfaces such as SCI-P, SMI-P and so on defined by means of roles at the association ends.
Eu.ModIn.53	**Interaction partners** Interaction partners **(4, 5)** of the SUS **(1)** are represented by actors. An actor describes a person (for example "Maintainer") or another system (for example the "Subsystem - Electronic Interlocking) in the role of a user of services offered by the SUS concerned (here "Subsystem Point"). At the logical viewpoint actors are represented by logical structural entities if they are in the context of a system element belonging to the same overall system. If an actor in the context of a system element is outside of the overall system of this system element (adjacent system) it is represented by an environmental structural entity.
Eu.ModIn.54	*Figure 16* therefore includes for example the following related definitions: • system element "Subsystem Electronic Interlocking" represented by a logical structural entity (LSE) assumes the role of an actor in the environment of "Subsystem Point" belonging to the same overall system **(4)**. • system element "Point machine" represented by an environmental structural entity (ESE) assumes the role of an actor in the environment of "Subsystem Light Signal" not belonging to the same overall system **(5)**.
Eu.ModIn.56	**Logical SUS interfaces** The connection between the SUS (represented by a logical structural entity) and an actor represents a logical interface **(2, 3)**. It is depicted as an association that is a continuous line between the actor and the SUS. It represents the requirement that the SUS must be able to interact with the connected actor through a corresponding logical interfaces.
Eu.ModIn.57	The association also represents the possible interaction directions of the interface. No arrow heads means that the interaction is bidirectional. An arrow head on the other hand indicates that an interaction is only possible in the direction of the arrow.
Eu.ModIn.58	On the side of the actor of the association, a multiplicity indication describes in more detail with how many of the respective actors the SUS concerned must be able to interact i.e., how many logical interfaces are required.
Eu.ModIn.336	The definition of the quantity of each actor by means of multiplicities represents an important requirement regarding system development. It is obvious that it makes a difference, for example, whether the system depicted in *figure 16* requires an interface to one "Subsystem Electronic Interlocking" or to several.
Eu.ModIn.59	The multiplicity "1" is defined at the SUS side of the association. The reason for this is that only requirements for the SUS concerned may be phrased in the respective requirements specification. However, according to the SysML syntax, a multiplicity indication at the SUS side would represent a statement for the actor.
Eu.ModIn.61	Some examples for the representation of multiplicities and their meaning: 1 or blank exactly one 0..1 none or one * none or several 1..* one or several 2..4 at least two and at most four
Eu.ModIn.62	*Figure 16* therefore includes for example the following related definitions: • the "Subsystem Point" must be able to interact with exactly one "Subsystem Electronic Interlocking" as an actor, with the interaction possible in two directions. • the "Subsystem Point" must be able to interact with one or more actors "Point machine", with the interaction possible in two directions. • the "Subsystem Point" must be able to interact with exactly one "Basic Data Identifier" as an actor, with an interaction only possible from "Basic Data Identifier" to the "Subsystem Point".
Eu.ModIn.661	Roles at the association ends represent the used "Interface kind" such as SCI-P, SMI-P and so on. In *figure 16* "Subsystem Point" sees for example "Subsystem Electronic Interlocking" in the role of "SCI-P" and vice versa.
Eu.ModIn.67	*Figure 16* therefore includes for example the following related definitions: • the interface between "Subsystem Point" and "Subsystem Electronic Interlocking" must be implemented according to the specification of "SCI-P". • the interface between "Subsystem Point" and "Subsystem Maintenance and Data Management" must be implemented according to the specification of "SMI-P". • the interface between "Subsystem Point" and "Subsystem Maintenance and Data Management" must be implemented according to the specification of "SDI-P". • the interface between "Subsystem Point" and "Subsystem Security Services Platform" must be implemented according to the specification of "SSI-P".

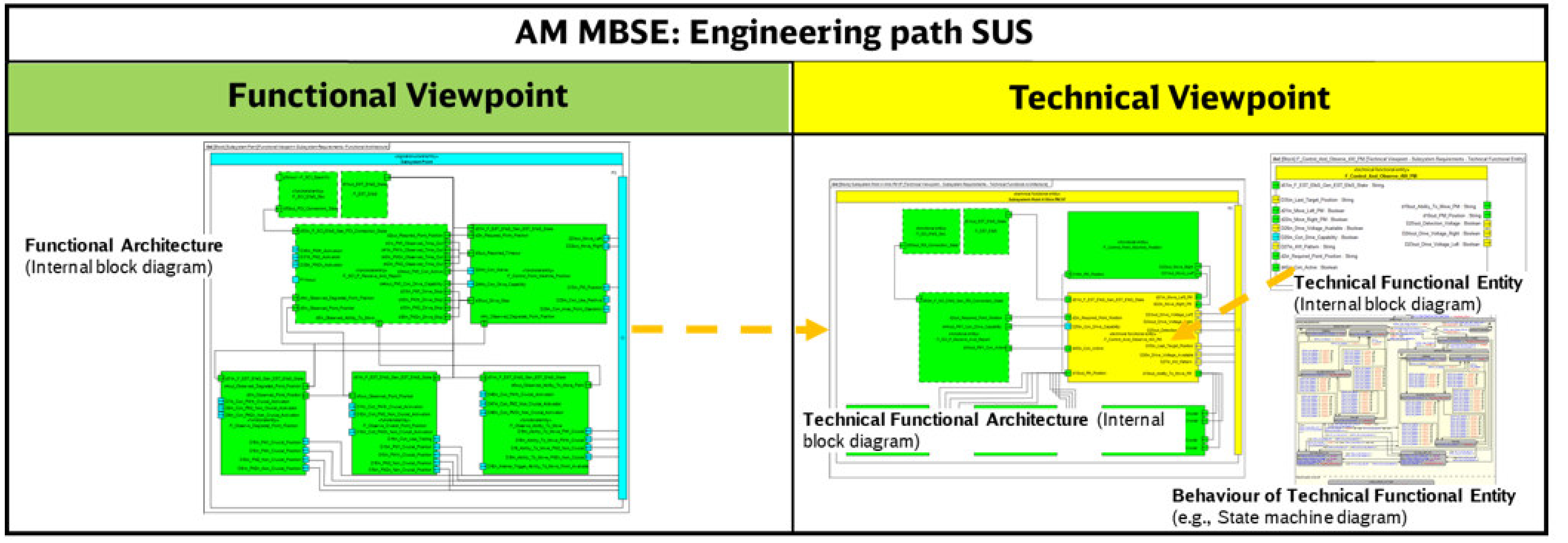
ID	Requirements
Eu.ModIn.203	Figure 16 Logical Context 
Eu.ModIn.800	**4.1.3.1 Binding** (see *chapter 2.1*)
Eu.ModIn.801	**Diagram of model view "Logical Context"** has a "**Def**" binding.
Eu.ModIn.238	**4.2 Abstraction Level AL2: System Requirements**
Eu.ModIn.745	**4.2.1 Model view "Functional Partitioning" of a SUS**
Eu.ModIn.746	The model view "Functional Partitioning" shown in *Figure 45* describes the refinement of the SUS **(1)** by FEs.
Eu.ModIn.747	The FEs **(2)** defined in the SIUS model view "Functional Partitioning" (see *chapter 5.2.1)*, which represent the local behaviours of the PDI (see *chapter 3.4*), and the generic FEs **(3)** are referenced by the SUS through reference associations **(5)**. FEs which are assigned to the subsystem via reference associations (marked with a white diamond) are not part of the subsystem, but are only used there. They represent the local behaviour of the PDI of the corresponding SIUS and are part of it.
Eu.ModIn.748	The SUS-specific FEs **(4)** are part of the SUS which is represented by composite associations **(6)**. FEs which are assigned to the subsystem via composite associations, i.e. so-called whole-part relationships (marked with a black diamond) are part of the subsystem. They represent the specific behaviour of the subsystem that influences more than one interface. This so-called "linking behaviour" is also used to link the behaviour assigned to the interfaces.
Eu.ModIn.749	The model view "Functional Partitioning" forms the basis for the model view "Functional Architecture" (see *chapter 4.2.2*). It defines the FEs in their maximum quantity structure in the form of multiplicities. Within the framework of this quantity structure, the FE configurations required for the definition of the functional requirements are then created in the model view "Functional Architecture".

ID	Requirements
Eu.ModIn.750	Figure 45 Example of SUS model view "Functional Partitioning" 
Eu.ModIn.804	**4.2.1.1 Binding** (see *chapter 2.1*)
Eu.ModIn.805	**Diagram of model view "Functional Partitioning"** has a "**Def**" binding.
Eu.ModIn.387	**4.2.2 Model view "Functional Architecture" of a SUS**
Eu.ModIn.751	*Figure 46* shows the model view "Functional Architecture" (FA) of Subsystem Point. It is created based on the in model view "Functional Partitioning" defined FEs.

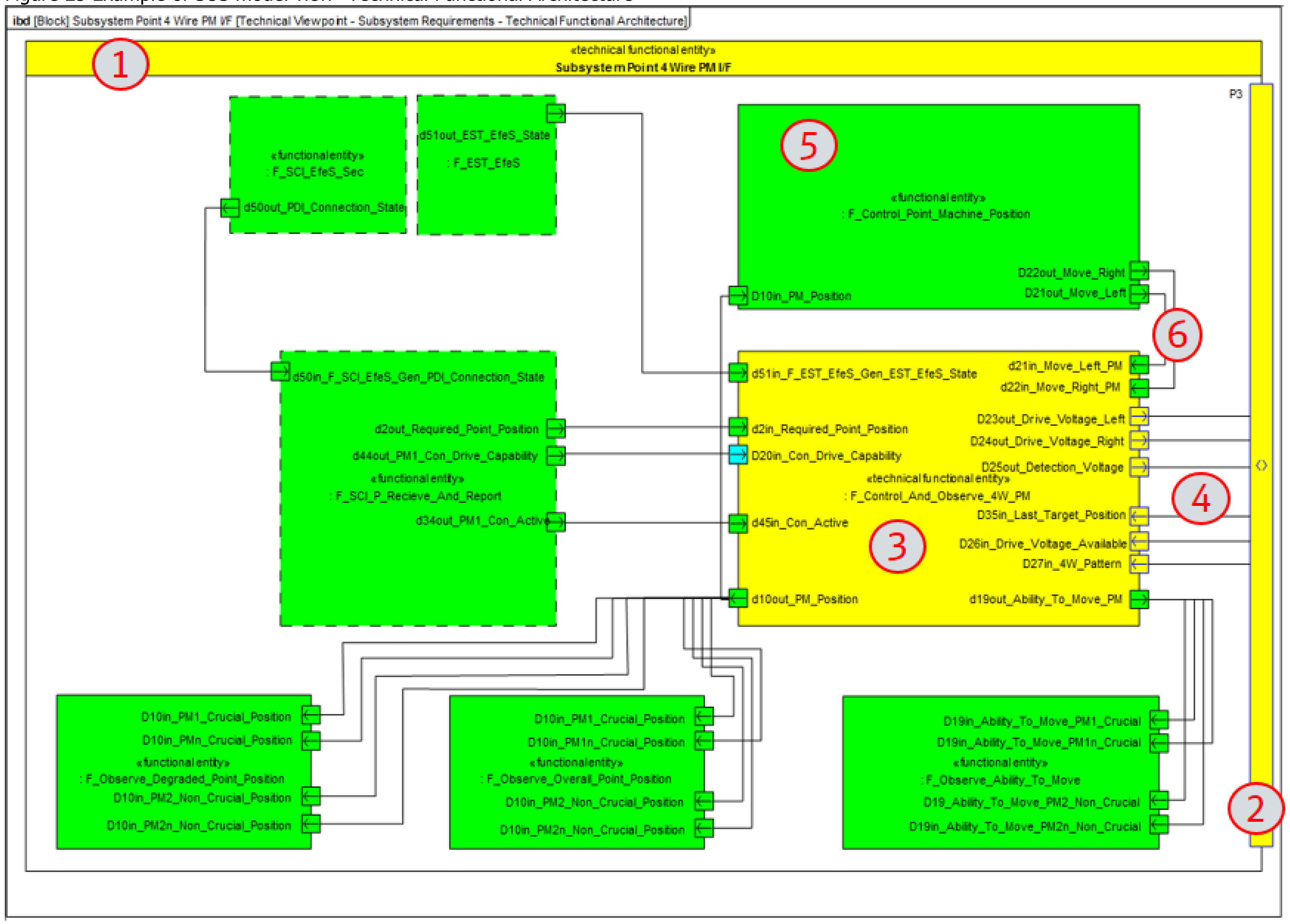
ID	Requirements
Eu.ModIn.752	Figure 46 Model view "Functional Architecture" of Subsystem Point 
Eu.ModIn.264	The model view "Functional Architecture" is explained in the following with a simple example as shown exemplarily in *Figure 23*. It describes the external visible stimulus-response behaviour of a SUS **(1)** represented by a Logical Structural Entity (LSE) that is structured in a way that enables an interface centric specification approach as described in *chapter 3.4*. The behaviour of the SUS is divided into Functional Entities" (FE), which communicate with each other via internal interfaces and with the environment via external interfaces.
Eu.ModIn.262	**Functional Entities** To describe the overall behaviour of an SUS observable externally in an FA structured, two different representations of the FEs **(4, 5)** are used: FEs with a solid border **(5)** and FEs with a dashed border **(4)**. Following the interface centric specification paradigm explained in chapter 3.4, a solid-bordered FE represents the directly specified behaviour of the SUS that is the "linking behaviour" (e.g. S_P : S_P). It is an inseparable part of the SUS behavioural model. FEs with dashed borders, on the other hand, are references (reference properties) to the interface protocols specified in the models of the application levels. These local behaviours are linked to the overall behaviour of the SUS by the directly specified SUS linking behaviour. The model view "Functional Entity" is described in *chapter 6*.
Eu.ModIn.669	In *figure 23*, for example, the functional entity "S_SCI_P_Command_and_Receive" is shown as a dashed block. This means that it is the local behaviour of the SCI-P protocol at application level, which is defined in the SCI-P specification (see *chapter 5*).
Eu.ModIn.679	**Internal FE-coupling** Internal FE-couplings are implemented in two variants. In variant 1 **(6)**, communication between two FEs takes place by means of signals and in variant 2 **(7)**, permanent information is transmitted.

ID	Requirements
Eu.ModIn.672	**Variant 1 (6):** an internal FE-coupling according to variant 1 defines an event-driven flow. It consists of two SysML proxy ports with the same name that are connected via a connector (SysML Connector). The connector represents the communication channel over which the information objects defined in the port type (SysML interface block) such as "w_p" can be exchanged. The information objects are represented by SysML signals (see *chapter 5.2.3* and *chapter 6.2.9.4*). The port type is used conjugated on one side (e.g., ~w_p). This means that an information object defined as outgoing in the interface block (port type) becomes an incoming information object through conjugation.
Eu.ModIn.673	Port name and port type are written in lower case. In addition, the ports are shown in the colour of the FEs.
Eu.ModIn.680	**Variant 2 (7):** an internal FE-coupling according to variant 2 defines a continuous flow. It consists of two SysML proxy ports or alternatively SysML flow ports with the same name that are connected via a connector (SysML Connector). The continuity of the information transmission is indicated by the abbreviation "d = data" at the beginning of the names of the ports involved.
Eu.ModIn.681	The information flows defined in the internal FE-couplings or the couplings themselves are to be interpreted as descriptive elements of the behaviour and are only binding in the context of the overall behaviour. That means that an information flow defined in an internal FE-coupling only becomes a mandatory requirement in the context of its active use, e.g. in a transition.
Eu.ModIn.760	**Please note:** In some cases, flow ports are still used to describe internal FE-couplings (see for example *Figure 7755*). However, these will gradually be replaced by proxy ports in the future.
Eu.ModIn.671	Ports used for internal FE-coupling are defined as **functional ports**. Their names are written in lower case. In addition, the ports are shown in the colour of the FEs.
Eu.ModIn.674	**External FE-coupling** The overall behaviour to be implemented by the manufacturers is connected to the **logical SUS interfaces (2)** via external FE-couplings **(3)**.
Eu.ModIn.675	An external FE-coupling consists of a proxy port representing a logical SUS interface, located at the SUS outer boundary and labelled with the designator of the interface concerned (e.g. SCI_P : SCI_P_Subsystem_EIL). The proxy ports delegated from the FEs relevant to the interface using binding connectors **(3)** and representing the information flows (e.g. P11in : ~SCI_P_2 or P10inout : SCI_P_1) are embedded in it **(9)**.
Eu.ModIn.676	In other words, the port (e.g. P10inout : ~SCI_P_1) at the FE is duplicated on the SUS outer boundary. Both ports are connected with a binding connector. The information flows and their direction remain unchanged in the interface block of the duplicated port.
Eu.ModIn.677	The names of the proxy ports used in an external coupling (e.g. P11in or P10inout) designate the information flows assigned to the logical SUS interface. The port types (e.g. SCI_P_2 or SCI_P_1) define the information objects of the information flows that must be able to be exchanged via the respective interface.
Eu.ModIn.678	The information objects defined in the information flows or the couplings themselves are to be interpreted as descriptive elements of the behaviour and are only binding in the context of the overall behaviour. That means that an information object defined in an external FE-coupling only becomes a mandatory requirement in the context of its active use, e.g. in a transition.
Eu.ModIn.759	**Please note:** In some cases, flow ports are still used to describe internal FE-couplings (see for example interface P3 in *Figure 7755*). However, these will gradually be replaced by proxy ports in the future.
Eu.ModIn.754	Ports used for external FE-coupling are defined as **logical ports**. Port name and port type are written in capital letters. In addition, the ports are shown in the colour blue.
Eu.ModIn.682	**Open ports** Open ports **(8)** that is ports not associated to connectors define interfaces to specification parts not contained in the model, i.e. expected behaviour in the environment of the FEs. This behaviour can be implemented proprietarily by each manufacturer, as long as the information expected at the ports is provided or the information delivered via the ports is processed accordingly.
Eu.ModIn.755	Ports used as open ports are defined as **logical ports**. Port name and port type are written in capital letters. In addition, the ports are shown in the colour blue.
Eu.ModIn.683	Open ports are also used to configure the specified behaviour.
Eu.ModIn.260	**Please note:** The Functional Architecture (FA) is not to be understood as a specification for an internal architecture of the SUS, but as a descriptive structuring. The FEs in communication relationship represent the expected overall behaviour of a SUS, which must be fulfilled by the respective manufacturer in its entirety.

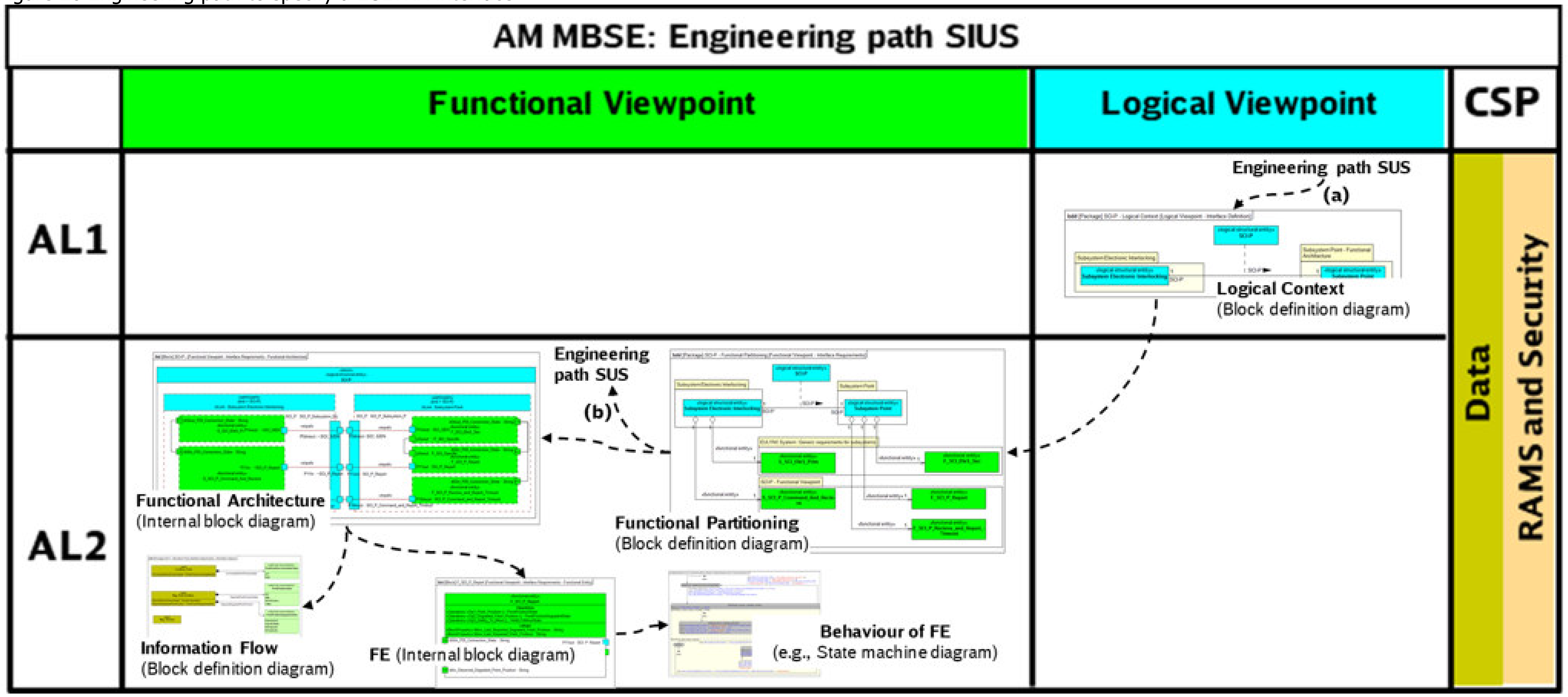
ID	Requirements
Eu.ModIn.299	Figure 23 Functional Architecture of a SUS 
Eu.ModIn.868	**Please note:** Regarding the use of **flow ports**, **flow specifications** and **flow properties** see **Eu.Doc.29 v1.5 (0:A)**.
Eu.ModIn.806	**4.2.2.1 Binding** (see *chapter 2.1*)
Eu.ModIn.807	**Diagram of model view "Functional Architecture"** has a "**Def**" binding.
Eu.ModIn.808	**Ports** have a "**Def**" binding.
Eu.ModIn.809	**Flow specifications** have an "**Info**" binding.
Eu.ModIn.862	**FLow properties** of the flow specifications have a "**Def**" binding if they are further refined elsewhere (e.g. by linked telegram definitions in separate interface specifications or further requirements in chapter 5.X. of the subsystem requirements specifications).
Eu.ModIn.863	**FLowProperties** of the FlowSpecifications have a "**Req**" binding if they are not further refined elsewhere.
Eu.ModIn.756	**4.2.3 Model view "Technical Functional Architecture" of a SUS**
Eu.ModIn.727	*Figure 24* shows the engineering path of the model views used to specify a SUS at the Technical Viewpoint on abstraction level AL2.

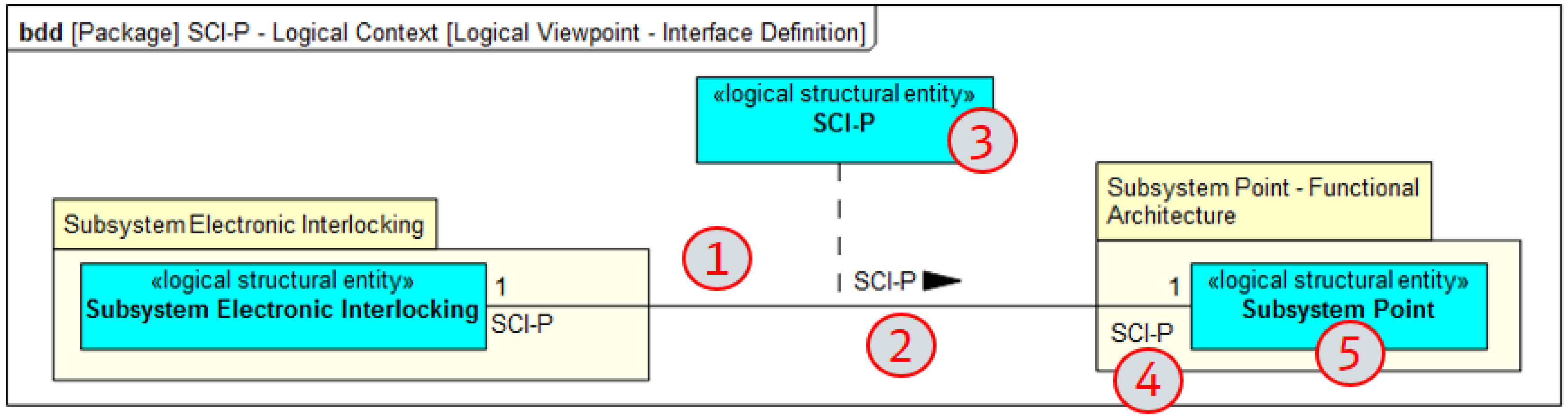
ID	Requirements
Eu.ModIn.724	Figure 24 Engineering path to specify a SUS at the Technical Viewpoint on abstraction level AL2 
Eu.ModIn.757	The model view "Technical Functional Architecture" (TFA) supplements or substitutes the behaviour described in the model view "Functional Architecture", which is independent of technology, with behavioural components derived from technical requirements. In other words, the FEs interconnected in the model view "Functional Architecture" are either transferred to the model view "Technical Functional Architecture" or completely or partially replaced by Technical Functional Entities (TFE).
Eu.ModIn.723	The SUS can either be described completely from a technical point of view (all FEs are replaced by TFEs) or only certain parts of it (interconnection of TFEs and transferred FEs).
Eu.ModIn.734	*Figure 25* shows an example of the transfer of the FES defined in the model view "Functional Architecture" to the model view "Technical Functional Architecture" of the SUS Subsystem Point. The SUS **(1)** is represented by a Technical Structural Entity (TSE). The transferred FEs **(5)** are supplemented with the TFE "F_Control_And_Observe_4W_PM" **(3)** that describes the functionality of the four-wire interface to a point machine based on technical requirements.
Eu.ModIn.742	In model view "Technical Functional Architecture" TFEs are coupled with each other, with the already defined FEs **(6)** and with the environment **(4)** via **external technical functional interfaces (2)**.
Eu.ModIn.761	The overall behaviour of a SUS structured by a TFA can be divided into several TFAs in the graphical representation.
Eu.ModIn.762	**Technical Functional Entities** To describe the overall behaviour of an SUS observable externally in an TFA structured, two different representations of the TFEs are used: TFEs with a solid border **(3)** and TFEs with a dashed border. Following the interface centric specification paradigm explained in *chapter 3.4*, a solid-bordered FE represents the directly specified behaviour of the SUS that is the "linking behaviour". It is an inseparable part of the SUS behavioural model. TFEs with dashed borders, on the other hand, are references (reference properties) to the interface protocols specified in the models of the application levels. These local behaviours are linked to the overall behaviour of the SUS by the directly specified SUS linking behaviour. The model view "Technical Functional Entity" is described in *chapter 6.*
Eu.ModIn.763	**Internal TFE-coupling and external TFE-coupling** The definitions for internal FE-coupling and external FE-coupling in *chapter 4.2.2* apply accordingly.
Eu.ModIn.764	Ports used for external TFE-coupling and internal TFE-coupling are defined as **technical functional ports**. They are shown in the colour yellow **(4)**.
Eu.ModIn.765	Ports used for internal coupling of FEs with TFEs are **functional ports**. They are shown in the colour green **(6)**.
Eu.ModIn.766	Ports representing **technical functional SUS interfaces (2)** can only be connected to technical functional ports **(4)**.
Eu.ModIn.767	**Open ports** Open ports that is ports not associated to connectors define interfaces to specification parts not contained in the model, i.e. expected behaviour in the environment of the TFEs. This behaviour can be implemented proprietarily by each manufacturer, as long as the information expected at the ports is provided or the information delivered via the ports is processed accordingly.
Eu.ModIn.768	Ports used as open ports are defined as **logical ports**. Port name and port type are written in capital letters. In addition, the ports are shown in the colour blue.

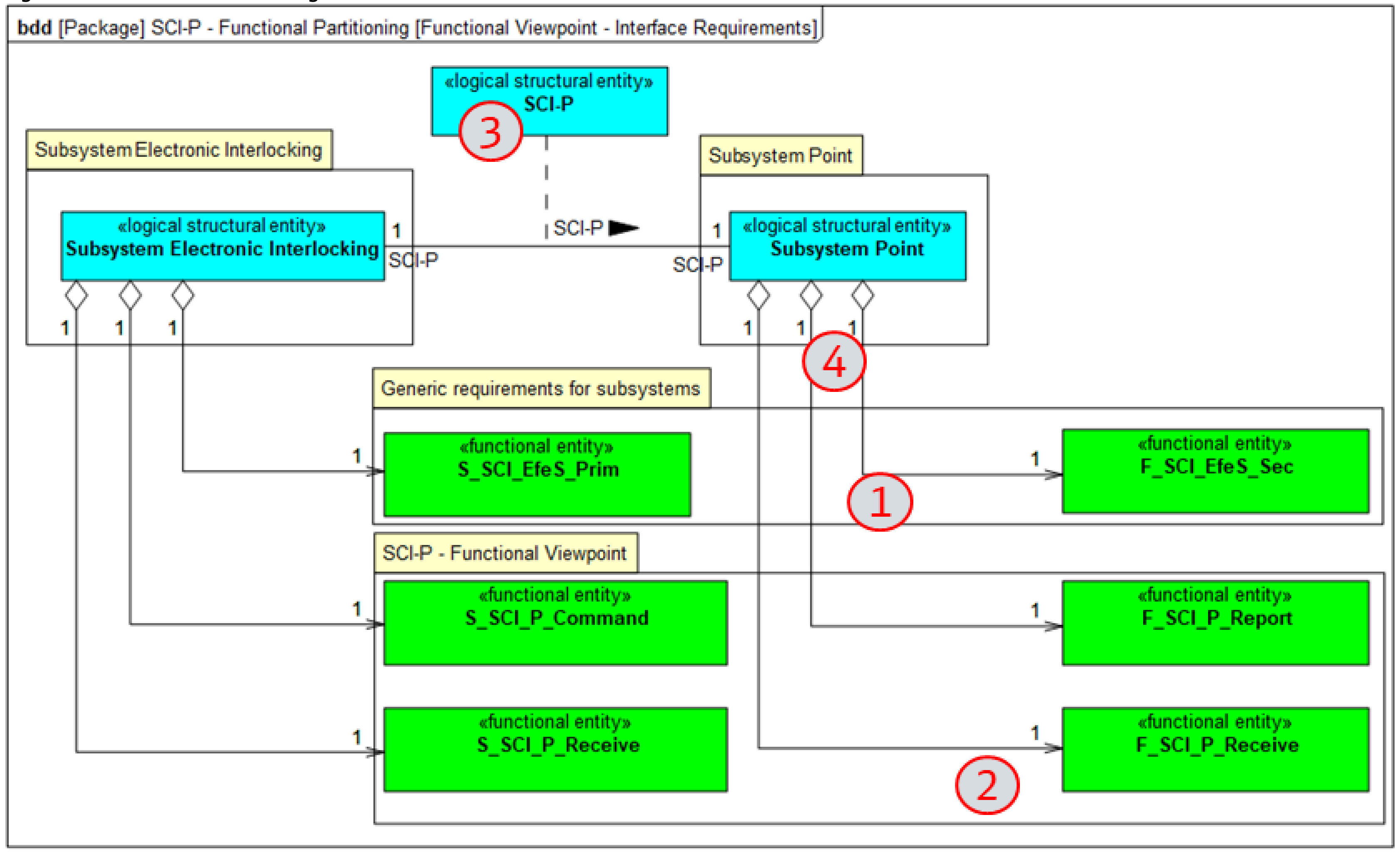
ID	Requirements
Eu.ModIn.758	**Please note:** The TFA is not to be understood as a specification for an internal architecture of the SUS, but as a descriptive structuring. The TFEs or FEs in communication relationship represent the expected overall behaviour of a SUS, which must be fulfilled by the respective manufacturer in its entirety.
Eu.ModIn.735	Figure 25 Example of SUS model view "Technical Functional Architecture"

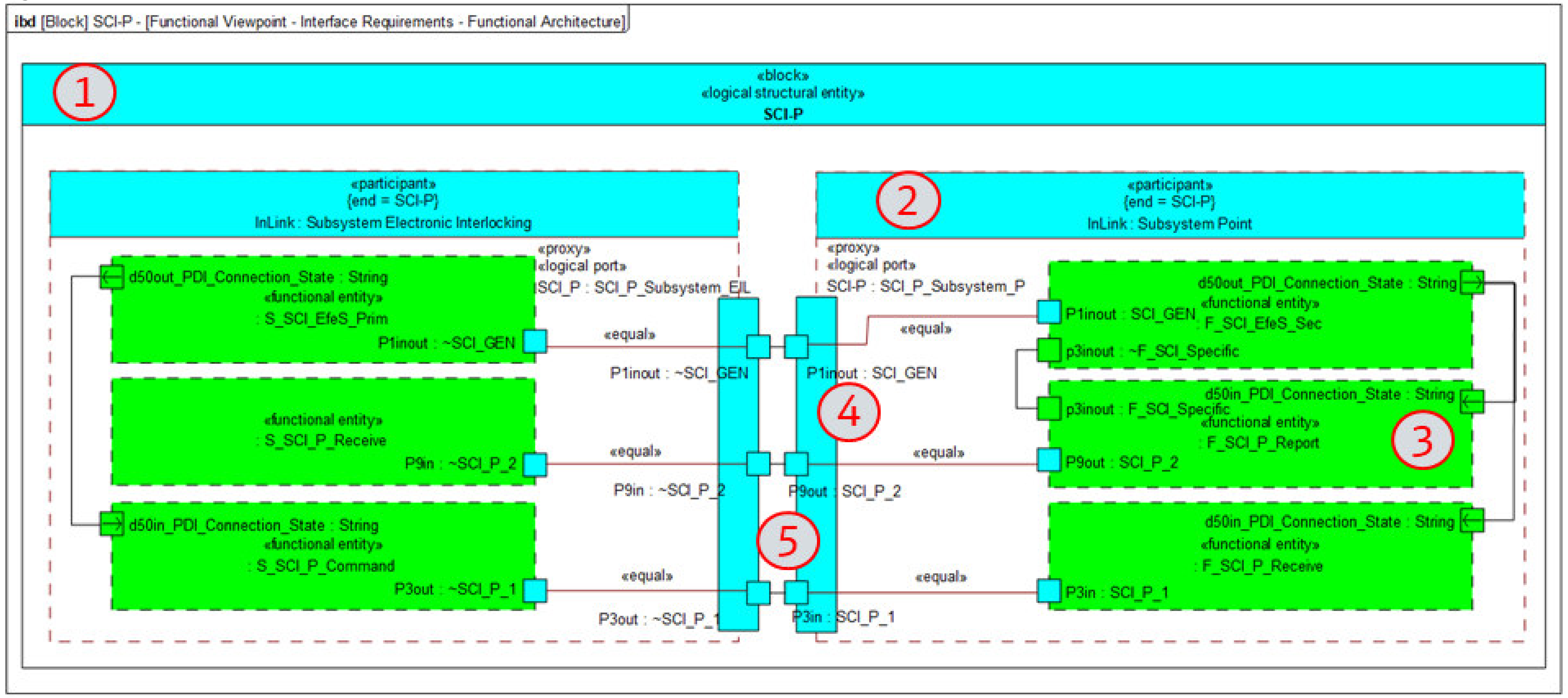


ID	Requirements
Eu.ModIn.810	**4.2.3.1 Binding** (see *chapter 2.1*)
Eu.ModIn.811	**Diagram of model view "Technical Functional Architecture"** has a "**Def**" binding.
Eu.ModIn.812	**Ports** have a "**Def**" binding.
Eu.ModIn.237	**5 Model views used to specify EULYNX interfaces**
Eu.ModIn.311	**Model view "Logical Context": Block Definition Diagram (bdd)** The model view "Logical Context" describes the logical view of an interface at the upper level of abstraction.
Eu.ModIn.276	**Model view "Functional Partitioning": Block Definition Diagram (bdd)** The model view "Functional Partitioning" describes the refinement of the interface defined in model view "Logical Context" using Functional Entities.

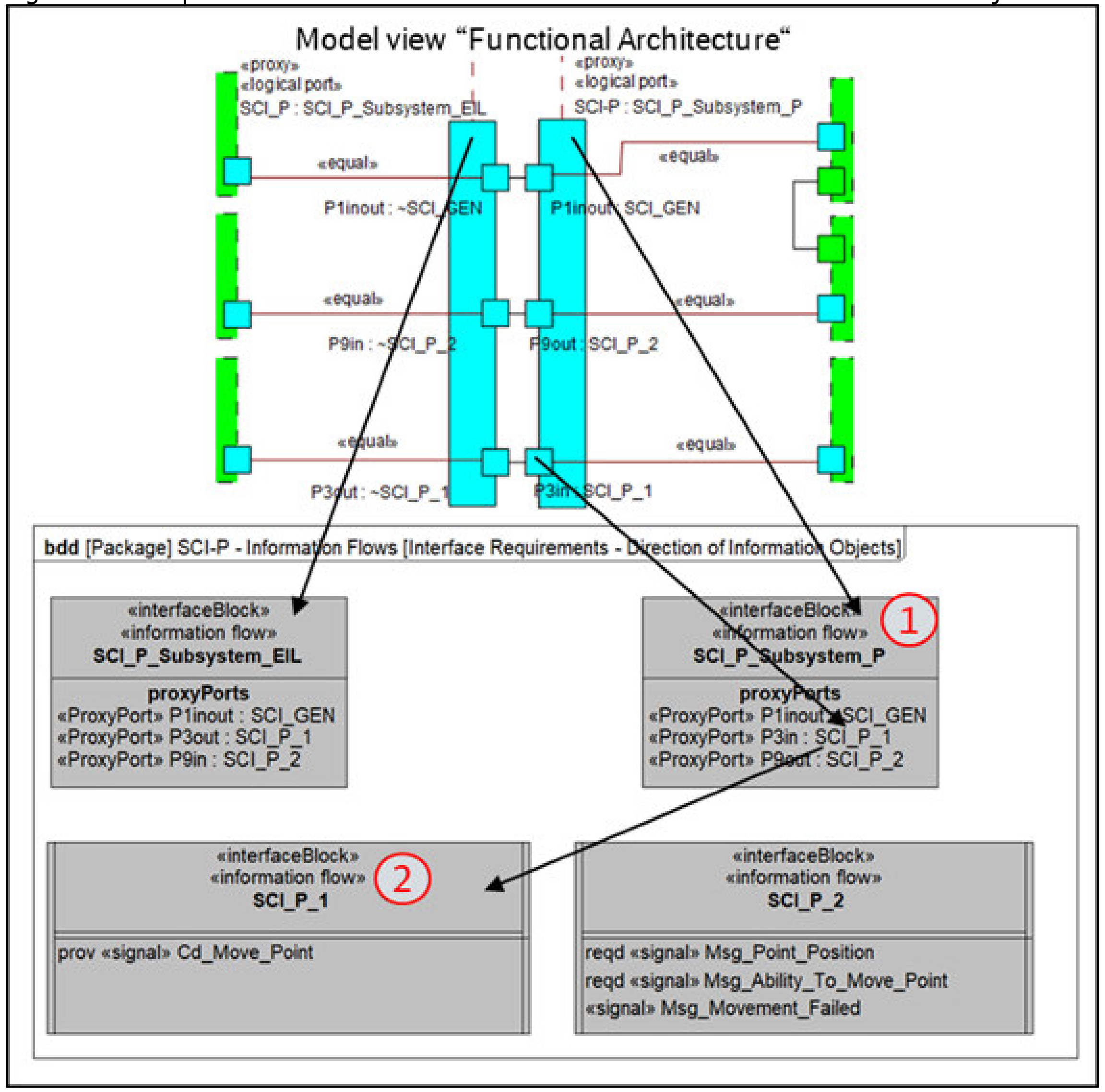
ID	Requirements
Eu.ModIn.684	**Model view "Functional Architecture": Internal Block Diagram (ibd)** The model view "Functional Architecture" defines the global behaviour of the application protocol (see *chapter 3.4*).
Eu.ModIn.686	**Model view "Functional Entity": Internal Block Diagram (ibd) and State Machine (stm)** The model view "Functional Entity" encapsulates a subset of the functional requirements of an SUS in the form of a function module. It delimits the function module from its environment and defines the inputs and outputs. In the discrete case, the behaviour of the function block is described by means of state machines. In this, the binding functional requirements are specified in the form of states and corresponding state transitions. As the model view "Functional Entity" is used for the specification of EULYNX system elements as well as for the specification of EULYNX interfaces it is described in the separate *chapter 6*.
Eu.ModIn.685	**Model view "Information Flow": Block Definition Diagram (bdd)** The model view „Information Flow" describes the information objects to be exchanged via an interface which are further refined to telegrams at abstraction level AL3. At present, the telegrams are not yet described in a model-based way. They are defined in the interface specifications (e.g. Interface Specification SCI-LS, Eu.Doc.38).
Eu.ModIn.698	*Figure 26* shows the engineering path of the model views used to specify a SIUS considering the Functional Viewpoint and the Logical Viewpoint. It describes the context of the model views, with the arrows indicating which model views are developed from which. Based on the definition of the logical SUS interfaces in model view "Logical Context" of the SUS (**a**: see *Figure 15* in *chapter 4*) the model views "Logical Context" and "Functional Partitioning" of the corresponding SIUS are created. The model view "Functional Partitioning" in turn forms the basis for the creation of the model view "Functional Architecture" of the SIUS and the model view "Functional Partitioning" of the SUS (**b**: see *Figure 15* in *chapter 4*). Subsequently, the model views "Information Flow" and "Functional Entity" are created.
Eu.ModIn.288	Figure 26 Engineering path to specify a EULYNX interface 
Eu.ModIn.250	**5.1 Abstraction Level AL1: Interface Definition**
Eu.ModIn.251	**5.1.1 Model view "Logical Context"**
Eu.ModIn.277	The model view "Logical Context" as shown in *figure 27* describes the logical view of an interface at the upper level of abstraction. In contrast to the logical context of a SUS in which the logical interfaces are also defined in terms of their number, an interface in its logical context is regarded as a one-to-one relationship.
Eu.ModIn.283	An interface **(1)** is generally defined as a unique connection between two communication participants **(5)**. From the logical viewpoint at the upper level of abstraction an interface is represented by a SysML association **(1).** An association is depicted as a continuous line between the communication participants. It also represents the possible interaction directions of the interface. No arrow heads means that the interaction is bidirectional. An arrow head on the other hand indicates that an interaction is only possible in the direction of the arrow. It represents the requirement that the two communication participants must be able to interact with each other.

ID	Requirements
Eu.ModIn.314	The logical interface represented by an association **(1)** is linked to a SysML association block **(3)**, which serves to refine the relationship. The global behaviour of the application protocol (Railway Control Protocol: RCP) is then specified in this later in the model view "Functional Architecture".
Eu.ModIn.261	A defined set of information objects (information flow) is transmitted via the interface in a precisely defined temporal sequence (protocol) in many cases. An information flow and the corresponding definition of the temporal sequence can apply to different interfaces. These two properties of an interface are called interface kind **(4)**. The interface kind is mapped at the association ends in the form of roles **(4)**. This separation of interface and interface kind makes it possible to communicate in the same way via several different "unique relationships = interfaces". The interface kind represents the requirement that it is to be applied to a specific interface.
Eu.ModIn.690	An interface is identified by a unique name **(2)** placed above or below the association **(1)** representing the interface.
Eu.ModIn.715	The black arrow shown in connection with the association indicates the reading direction. The directional arrow specifies the top-level navigation through the interface model to improve readability. It is taken into account when refining the model, for example when defining the conjugation of information flows. Beyond that, it has no meaning for the model.
Eu.ModIn.714	The interface name can be identical to the interface kind if it is certain that the interface kind is only applied to a specific interface and not to several different ones. If the interface name is the same as the interface kind, it may not be displayed.
Eu.ModIn.291	Figure 27 Logical context of an interface 
Eu.ModIn.814	**5.1.1.1 Binding** (see *chapter 2.1*)
Eu.ModIn.815	**Diagram of model view "Logical Context"** has a "**Def**" binding.
Eu.ModIn.687	**5.2 Abstraction Level AL2: Interface Requirements**
Eu.ModIn.252	**5.2.1 Model view "Functional Partitioning"**
Eu.ModIn.279	The model view "Functional Partitioning" as shown in *figure 28* describes the refinement of the interface defined in model view "Logical Context" using Functional Entities. These Functional Entities specify the local behaviours of the communication protocol stack scaled-down to the application layer (PDI: Process Data Interface Protocol) at each side of the communicating system elements.
Eu.ModIn.270	The specific **(2)** and generic **(1)** local behavioural parts of the application protocol defined by FEs are referenced by the communication partners via SysML reference associations **(4)**. Reference associations are marked with a white diamond and express that the FEs are not part of the subsystems, but are only used there. They are part of the PDI.
Eu.ModIn.836	The FEs are used in the model view "Functional Architecture" to specify the global behaviour of the application protocol represented by the internal structure of the association block **(3)** associated with the association representing the interface.

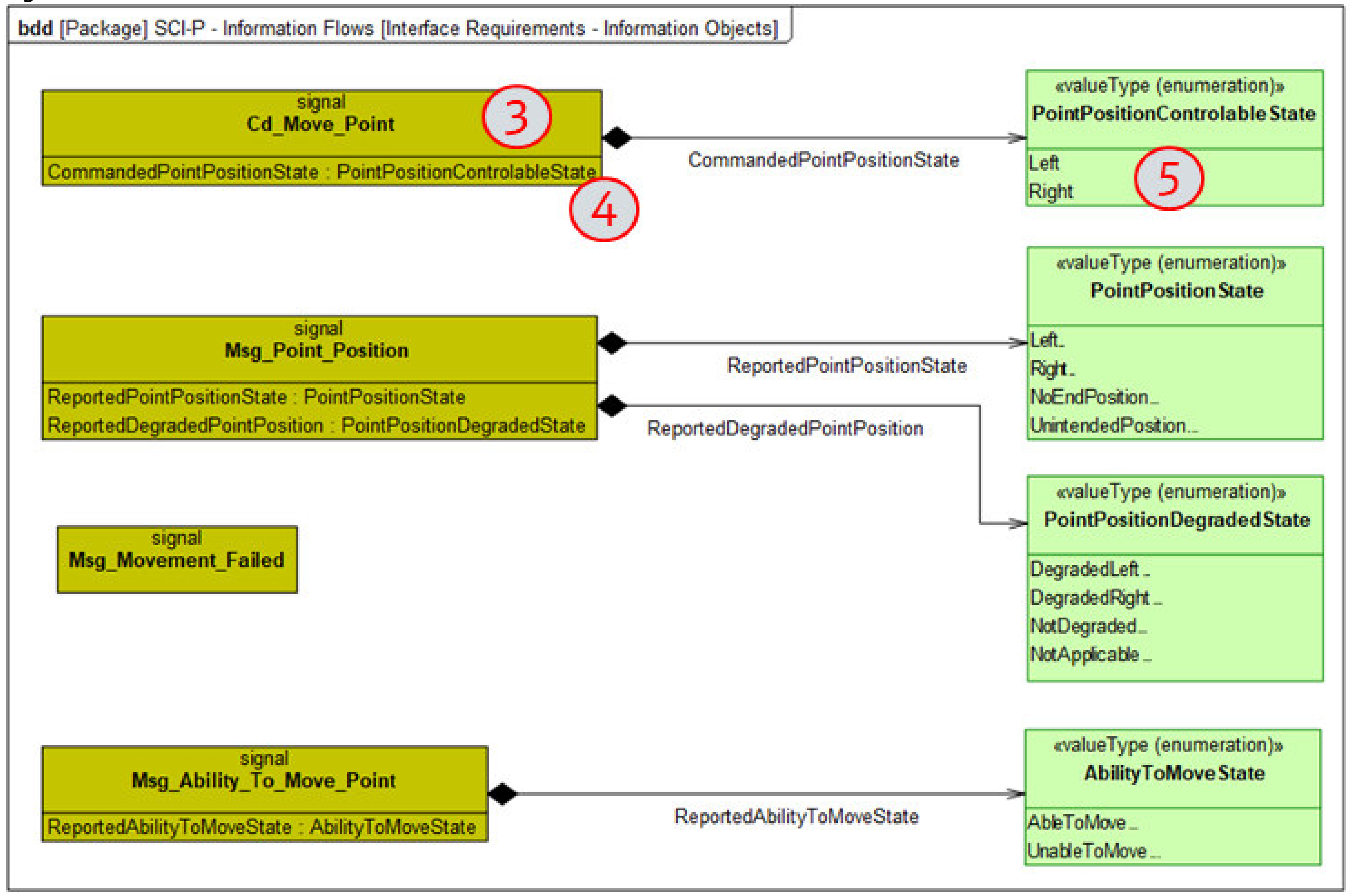
ID	Requirements
Eu.ModIn.292	Figure 28 Functional Partitioning of an interface 
Eu.ModIn.816	**5.2.1.1 Binding** (see *chapter 2.1*)
Eu.ModIn.817	**Diagram of model view "Functional Partitioning"** has a "**Def**" binding.
Eu.ModIn.253	**5.2.2 Model view "Functional Architecture"**
Eu.ModIn.281	The model view "Functional Architecture" as shown in *Figure 29* defines the global behaviour of the application protocol. The global behaviour is described by connecting the local behavioural components referenced by a communication partner with the corresponding ones of the neighbour via communication channels.
Eu.ModIn.691	The description of the global behaviour of the application protocol is done by the internal structuring of the association block **(1)** defined in the model view "Functional Partitioning". In this process, the communication partners **(2)**, which in turn reference the local behavioural parts of the protocol represented by FEs **(3)**, are referenced in the form of SysML participant properties and connected via their logical SUS interfaces **(4)** with connectors **(5)**.

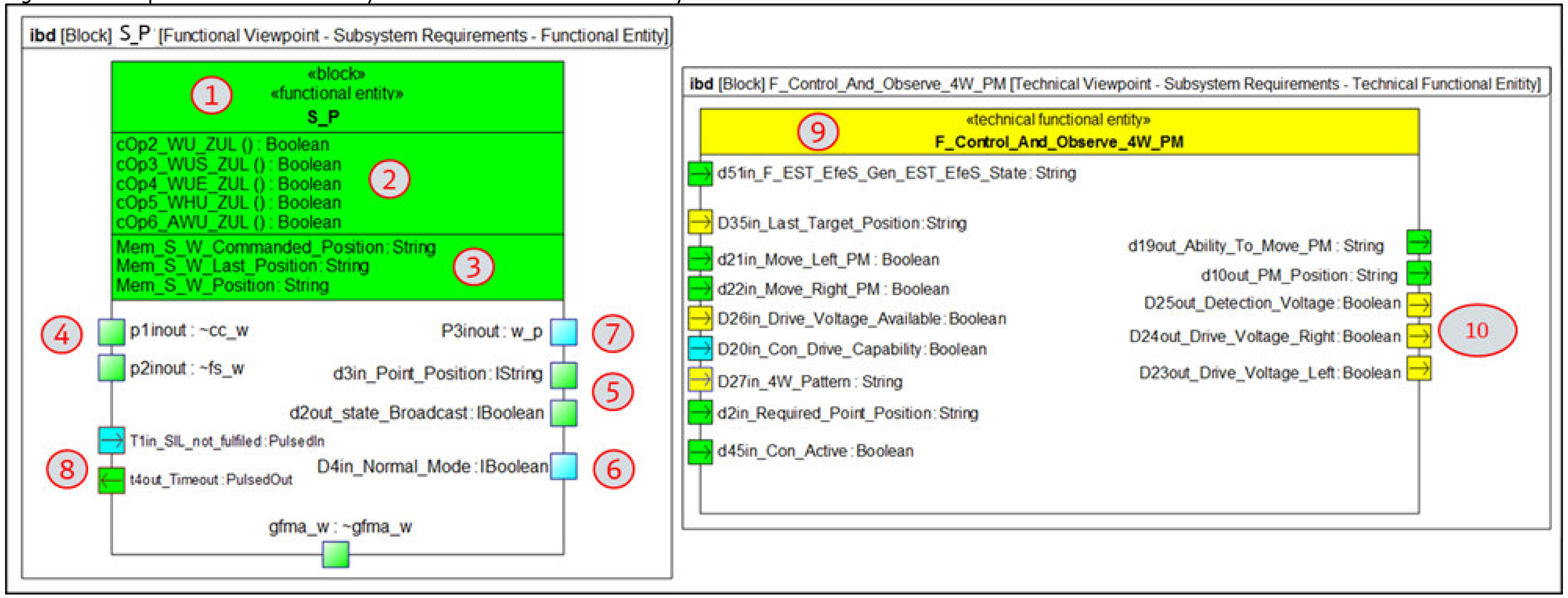
ID	Requirements
Eu.ModIn.290	Figure 29 Functional Architecture of an interface 
Eu.ModIn.818	**5.2.2.1 Binding** (see *chapter 2.1*)
Eu.ModIn.819	**Diagram of model view "Functional Architecture"** has a "**Def**" binding.
Eu.ModIn.837	**Ports** have a "**Def**" binding.
Eu.ModIn.254	**5.2.3 Model view "Information Flow"**
Eu.ModIn.303	The model view "Information Flow" describes the information objects to be exchanged via an interface. It consists of the two sub-model views "Direction of Information Objects" and "Information Objects", which are shown in *Figure 47* and *Figure 30* respectively.
Eu.ModIn.770	As shown in *Figure 47*, the SUS interfaces such as SCI_P are depicted by proxy ports. These are typified with interface blocks such as SCI_P_Subsystem_P **(1)**, which represent information flows in the form of embedded proxy ports such as P10inout. The embedded proxy ports are typed with interface blocks **(2)**, which in turn contain the information objects (e.g. Cd_Move_Point).

<table>
<tr><th>ID</th><th>Requirements</th></tr>
<tr><td>Eu.ModIn.769</td><td>
Figure 47 Example of SIUS model view "Information flow" - Direction of Information Objects



</td></tr>
<tr><td>Eu.ModIn.771</td><td>As shown in Figure 30, the information objects are represented by SysML signals such as "Cd_Move_Point" (3). These signals can in turn have attributes such as "CommandedPointPositionState" (4) that represent parameters of the information objects. The attributes are typed with basic data types or for example enumerations such as "PointPositionControlableState" (5).</td></tr>
</table>

ID	Requirements
Eu.ModIn.310	Figure 30 Information flow 
Eu.ModIn.694	**Please note:** These model views can also be used in an adapted form to define the information flows for internal couplings between FEs or TFEs in a Functional Architecture or Technical Functional Architecture.
Eu.ModIn.820	**5.2.3.1 Binding** (see *chapter 2.1*)
Eu.ModIn.821	**Diagram of model view "Information Flows - Direction of Information Objects"** has a "**Def**" binding.
Eu.ModIn.822	**Diagram of model view "Information Flows - Information Objects"** has a "**Def**" binding.
Eu.ModIn.823	**Information Objects (Signals)** have a "**Def**" binding if they are further specified in a refined model view or in the form of a separate requirement.
Eu.ModIn.839	**Information Objects (Signals)** have a "**Req**" binding if they are not further specified in a refined model view or in the form of a separate requirement.
Eu.ModIn.668	**6 Model view "Functional Entity" and "Technical Functional Entity"**
Eu.ModIn.316	**6.1 Concept and interpretation of Functional Entities and Technical Functional Entities**
Eu.ModIn.109	Within the EULYNX approach to specify model-based requirements the concept of Functional Entity (FE) and Technical Functional Entity (TFE) is used.
Eu.ModIn.399	FE and TFE represent behavioural entities and encapsulate a subset of the functional requirements of a SUS or SIUS in the form of stimulus-response behaviour independent of any architectural constraints. While FEs define technology-independent functional requirements, TFEs describe technology-dependent ones.
Eu.ModIn.407	**Please note:** FEs and TFEs are not to be interpreted as elements of the hardware- or software architecture.
Eu.ModIn.110	The stimulus-response behaviour of FEs and TFEs is defined by SysML state machines (see chapter 6.2).
Eu.ModIn.600	The principle structure of a Functional Entity and a Technical Functional Entity is shown in *Figure 31.*

<table>
<tr><th>ID</th><th>Requirements</th></tr>
<tr><td>Eu.ModIn.219</td><td>Figure 31 Example of a Functional Entity and a Technical Functional Entity

</td></tr>
<tr><td>Eu.ModIn.562</td><td>Apart from state machines, FEs and TFEs may own
• SysML block properties (3),
• SysML block operations (2),
• SysML proxy ports used as atomic "in ports" and "out ports" (5, 6) or typed with an interface block in which the information objects to be exchanged via the port are defined (4, 7),
• SysML flow ports used as atomic "in ports" and "out ports" (8, 10).</td></tr>
<tr><td>Eu.ModIn.127</td><td>The description of a FE (1) contains the stereotype <<functional entity>> as well as the FE name (e.g. S_P).</td></tr>
<tr><td>Eu.ModIn.737</td><td>The description of a TFE (9) contains the stereotype <<technical functional entity>> as well as the TFE name (e.g. F_Control_And_Observe_4W_PM).</td></tr>
<tr><td>Eu.ModIn.563</td><td><h3>6.1.1 Block properties</h3></td></tr>
<tr><td>Eu.ModIn.566</td><td>Block properties (3) are to be interpreted in the sense of variables or constants that store values. They are prefixed with "Mem".
Examples: Mem_last_Target_Requested, Mem_Current_Point_Position.</td></tr>
<tr><td>Eu.ModIn.572</td><td>All block properties are initialised. The initialisation can be carried out in the body of the init-block operation systematically named cOp1_init(). Alternatively it can be carried out directly in the transition effect of the transition outgoing from initial state of the state machine.
Example:
Mem_S_W_Position := "";
Mem_SW_Last_Position := "";
The assignments of values to the corresponding block properties are to be interpreted as definitions. They become mandatory requirements (binding character "Req") when they are used in a mandatory requirement, such as a transition of a state.</td></tr>
<tr><td>Eu.ModIn.564</td><td><h3>6.1.2 Block operations</h3></td></tr>
<tr><td>Eu.ModIn.567</td><td>Block operations (2) are used in order to specify
• internal broadcast events or
• algorithms of data transformations defined in the operation body (call behaviour or time advance behaviour).</td></tr>
<tr><td>Eu.ModIn.864</td><td>The content of an operation defined in the operation body is displayed in the specification document in "Requirements Part 1" and the name of the operation is noted above it as a comment. The name of the operation is also displayed in "Requirements Part 2".</td></tr>
</table>

ID	Requirements
Eu.ModIn.568	**Internal broadcast events** Internal broadcast events are prefixed with bc<Id> where "Id" is a natural number starting with 1 (example: bc1_indicate_signal_aspect). Example: bc1_Bc_info(), bc2_Bc_info()
Eu.ModIn.570	**Call behaviour** Block operations used to define call behaviour are prefixed with cOp<Id> where "Id" is a natural number starting with 1.

Eu.ModIn.569	Call behaviour is invoked on demand, executed and terminated after execution. It is supposed to define event-driven data transformations. The algorithm of the data transformations is described in the body of the corresponding block operation using the Atego Structured Action Language (see *chapter 6.1.5*).
	
	Example: cOp2_All_Left
	if cOp8_Supports_Multiple_PMs() then
	return (
	(D21in_PM1_Position = "LEFT") and
	(D22in_PM2_Position = "LEFT" or D13in_PM2_Activation= "INACTIVE")
	);
	else
	return D21in_PM1_Position = "LEFT";
	end if

Eu.ModIn.574	**Call operations** are used as • boolean expressions or parts of it in change events: e.g. when(cOp3_No_End_Position)/ • transition guards: e.g. when(cOp5_Trailed)[cOp7_Is_Trailable]/ • transition effects: e.g after(D5in_Con_tmax_Point_Operation/cOp12_Timeout();
Eu.ModIn.571	The call operation to initialise the block properties and Out Ports of a FE is named cOp1_init() systematically.
Eu.ModIn.575	Call operations are to be interpreted as definitions. They become mandatory requirements (binding character "Req") when they are used in a mandatory requirement, such as a transition of a state.
Eu.ModIn.840	**Time advance behaviour** Time advance behaviour is invoked once during system activation and executes continuously. It is supposed to define continuous data transformation. The algorithm of the data transformations is to be described in the ***body*** of the corresponding block operation using the Atego Structured Action Language (see *chapter 6.1.5)*.
Eu.ModIn.841	**Example:** tOp1_indicate_availability_ratio
Eu.ModIn.565	**6.1.3 SysML in ports and out ports**
Eu.ModIn.111	A FE features interfaces that define the stimuli consumed by the assigned state machine, represented by in ports, and responses generated by the assigned state machine, represented by out ports.
Eu.ModIn.315	In ports and out ports are specified as SysML proxy ports or SysML flow ports of the SysML block representing the FE depicted in an internal block diagram (ibd).
Eu.ModIn.414	In ports and out ports are described according to the port definition schema below: *<Port information type><PNo><Port direction>_<Port information>:<Data type>*.
Eu.ModIn.124	**Port information type** Used port information type: • **D** or **d**: data ports (D-Ports), • **T** or **t**: trigger ports (T-Ports).
Eu.ModIn.708	**Data ports** and **trigger ports** start with a small letter (such as d3in_Point_Position or t4out_Timeout) if they are part of an internal connection between two FEs or between a FE and a TFE. In this case they are referred to as **functional ports** and have the colour green like the corresponding F E **(5)**.
Eu.ModIn.709	**Data ports** and **trigger ports** start with a capital letter if they are part of an external connection between a FE and the system environment (system interface) or if it is an open port (such as D4in_ Normal_Mode or T1in_SIL_not_fulfiled). In this case they are referred to as **logical ports** and have the colour blue **(6)**.

ID	Requirements
Eu.ModIn.733	Data ports and trigger ports which are part of a connection between TFEs or a TFE and the system environment (technical system interface) are referred to as **technical functional ports** and have the colour Yellow **(10)**. They start with a small letter if they are part of an internal connection between two TFEs and with a capital letter if they are part of an external connection between a TFE and the system environment (technical system interface).
Eu.ModIn.125	**Data ports (5), (6)** Data ports are especially suited to indicate permanently available information. The value of a D-port only changes if it is explicitly changed.
Eu.ModIn.577	Data in ports are used as arguments of Boolean expressions in change events or transition guards. They may represent arguments in data transformations or other data, that need to be permanently reachable by the behaviour of a FE (e.g configuration data: d21in_Con_Downgrade_Most_Restrict). Their values can be permanently regarded as valid.
Eu.ModIn.613	Data out ports are used to provide continuous data created within a FE for its environment (e.g. to be available for adjacent FEs, reachable via their data in ports).
Eu.ModIn.716	**Trigger ports (8)** Trigger ports are especially suited to indicate singular events. They have a Boolean value that always enters false and only briefly changes to true when the event occurs (data types PulsedIn or PulsedOut). Afterwards the value is automatically returned to false.
Eu.ModIn.717	Trigger in ports are mainly used as arguments of Boolean expressions in change events.
Eu.ModIn.410	**Port number (PNo)** For each port of a FE with the port information type "D or d" or "T or t", a unique PNo is to be assigned in the format of a natural number. The ports need not be numbered consecutively. For example port numbers like 1, 2, 3, 4, 5 are possible, but also 1, 3, 6.
Eu.ModIn.126	**Port direction** The direction of the in Ports and out Ports are additionally defined, i.e. whether it is a stimulus or a response for the FE. • An "in" after the port number represents a stimulus or a permanently present value, • An "out" after the port number represents a response.
Eu.ModIn.412	**Port information** The port information defines the information type and the semantic meaning of the information to be transmitted, e.g. "Cd_Indicate_signal_aspect". *<Port information> := <Information type> _<Information>*
Eu.ModIn.411	**Information type:** Msg (message), Cd (command), Con (configuration data), Site (site data) or project-specifically determined information types.
Eu.ModIn.413	**Information:** semantic meaning of the information to be transmitted, e.g. Indicate_signal_aspect.
Eu.ModIn.409	**Data type** The data type which is assigned to any in port and out port is only shown on the diagram if it is necessary for a correct interpretation.
Eu.ModIn.573	**Initialisation of out ports** All data out ports are initialised. The initialisation can be carried out in the body of the init-block operation systematically named cOp1_init(). Alternatively it can be carried out directly in the transition effect of the transition outgoing from initial state of the state machine. Trigger out ports are set to "FALSE" by default and are not explicitly initialised. Example: D25out_Redrive := FALSE; The assignments of values to the corresponding out ports are to be interpreted as definitions. They become mandatory requirements (binding character "Req") when they are used in a mandatory requirement, such as a transition of a state.
Eu.ModIn.696	**6.1.4 SysML proxy ports describing an event-based flow of information**
Eu.ModIn.406	A FE features interfaces that define event-driven in-flow of information consumed by the assigned state machine and event-driven out-flow of information generated by the assigned state machine.
Eu.ModIn.576	The information flows are represented by SysML proxy ports typed with SysML interface blocks **(4, 7)**.
Eu.ModIn.408	The information objects to be exchanged are represented by signals. The interface blocks define the receptions for these signals.
Eu.ModIn.612	When a signal is received, a signal event is triggered by the corresponding reception, which is then used as a trigger for a state transition, for example.

ID	Requirements
Eu.ModIn.711	Proxy ports to describe a signal-based information flow are described according to the port definition schema below: *<Port information type><PNo><Port direction>_<Port information>:<Signature of Interface block aggregating information objects>.*
Eu.ModIn.772	**Port information type** Used port information type: **P** or **p**
Eu.ModIn.773	Ports and their interface blocks are written in small letter (such as p1inout : ~cc_w) if they are part of an internal connection between two FEs. In this case they they are referred to as **functional ports** and have the colour green like the corresponding FE **(4)**.
Eu.ModIn.710	Ports and their interface blocks are written in capital letters if they are part of an external connection (system interface) between a FE and the system environment (such as P3inout : W_P) or if they are open ports. In this case they are referred to as **logical ports** and have the colour blue **(7)**.
Eu.ModIn.738	Ports which are part of a connection between TFEs or a TFE and the system environment (technical system interface) are referred to as **technical ports** and have the colour yellow **(10)**. They start with a small letter if they are part of an internal connection between two TFEs and with a capital letter if they are part of an external connection between a TFE and the system environment (technical system interface) or if they are open ports.
Eu.ModIn.712	An information object defined as outgoing in the interface block (port type) becomes an incoming information object through conjugation. This conjugation is indicated by the character "~" preceding the corresponding interface block (example: p1inout : ~cc_w).
Eu.ModIn.774	**Port number (PNo)** For each port of a FE/TFE with the port information type "P or p", a unique PNo is to be assigned in the format of a natural number. The ports need not be numbered consecutively. For example port numbers like 1, 2, 3, 4, 5 are possible, but also 1, 3, 6.
Eu.ModIn.775	**Port direction** The direction of the ports are additionally defined ("in", "out", "inout").
Eu.ModIn.776	**Port information** Freely selectable and optional.
Eu.ModIn.601	**Signature of Interface block aggregating information objects** The information flow through a proxy port is represented by an interface block in which the receptions for the incoming and outgoing information objects are defined. The information objects are represented by signals. The use of interface blocks and signals is described in the chapters 5.2.3 (Model view "Information Flow"), 6.2.9.4 (Signal event) and 6.2.10.1 (Event-driven responses using signals).
Eu.ModIn.578	**6.1.5 Action language**
Eu.ModIn.579	The EULYNX methodology follows the objective of creating executable specification models. In order to specify the necessary executable behaviours in a target language independent way, the Atego Structured Action Language (ASAL) is used.
Eu.ModIn.580	ASAL is used to specify block operations or Event Action Blocks that define the transition effects on state machine diagrams.
Eu.ModIn.581	A description of the basic statements of ASAL is provided below:
Eu.ModIn.582	**6.1.5.1 Logical operators**
Eu.ModIn.583	• Greater than: **>** • Less than: **<** • Greater than or equal: **>=** • Less than or equal: **<=** • Equal: **=** • Not equal: **<>** • Conjunction: **AND** • Disjunction: **OR** • Negation: **NOT** • Exclusive disjunction: **XOR**
Eu.ModIn.842	The logical operators "AND", "OR", "NOT" and "XOR" are written in capital letters.
Eu.ModIn.603	**6.1.5.2 Data types**
Eu.ModIn.604	As the EULYNX specification approach follows the objective of creating executable specification models, the range of data types is limited to data types the simulation tool SySim supports.

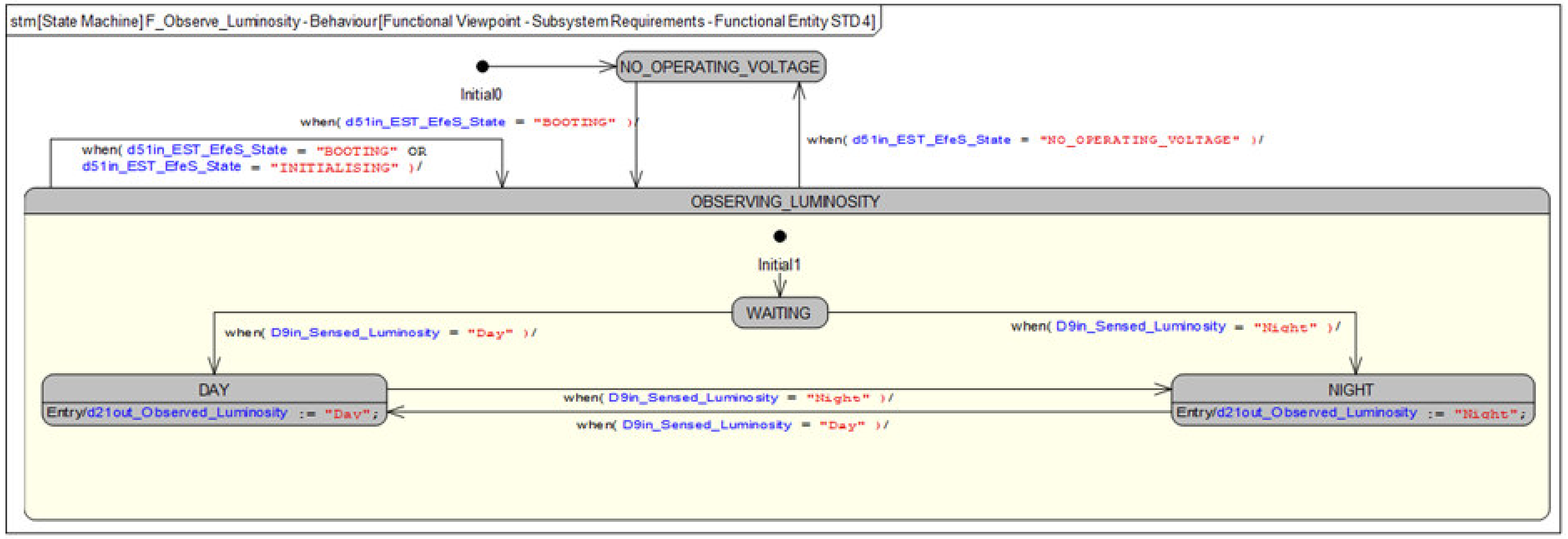
ID	Requirements
Eu.ModIn.605	Only the SySim value types, including the redefined data types "PulsedIn" and "PulsedOut" are used for the specification of systems requirements: • Boolean • DateTime • Single • String • Decimal • Double • Long • Integer • Timespan • PulsedIn • PulsedOut
Eu.ModIn.718	The data types “PulsedIn” and “PulsedOut” represent redefinitions of the data type Boolean and are exclusively reserved to be assigned to Trigger Ports (T-Ports). That is, a Trigger In Port is typed with the data type "PulsedIn" and a Trigger Out Port with the data type "PulsedOut".
Eu.ModIn.719	**Data type "PulsedOut"** Outgoing data typed with “PulsedOut” (as default false) that are set to true (for example, T1out_Cd_indicate_signal_aspect := true) automatically change back to false after a defined time. The defined time frame is sufficient to trigger a transition in a receiving state machine.
Eu.ModIn.720	**Data type "PulsedIn"** Incoming data at receiver side typed with “PulsedIn” apply the behaviour of the corresponding outgoing data at sender side typed with “PulsedOut”.
Eu.ModIn.843	For the typing of proxy ports, the specially adapted interface blocks are used: • IBoolean • IDateTime • IDecimal • IDouble • IInteger • ILong • ISingle • IString
Eu.ModIn.844	The data types "PulsedIn" and "PulsedOut" can only be used with flow ports but not in connection with proxy ports.
Eu.ModIn.584	**6.1.5.3 Reading the value of a port**
Eu.ModIn.585	The value of a port may be read using the name of the port on its own: The syntax is as follows: <A> := <port>; Where: <port> specifies the port whose value is being read. <A> specifies for example the value property the value of the port is to be assigned to. **Example:** Mem_D1_Signal_aspect := D1_Signal_aspect;
Eu.ModIn.586	**6.1.5.4 Setting the value of a port**

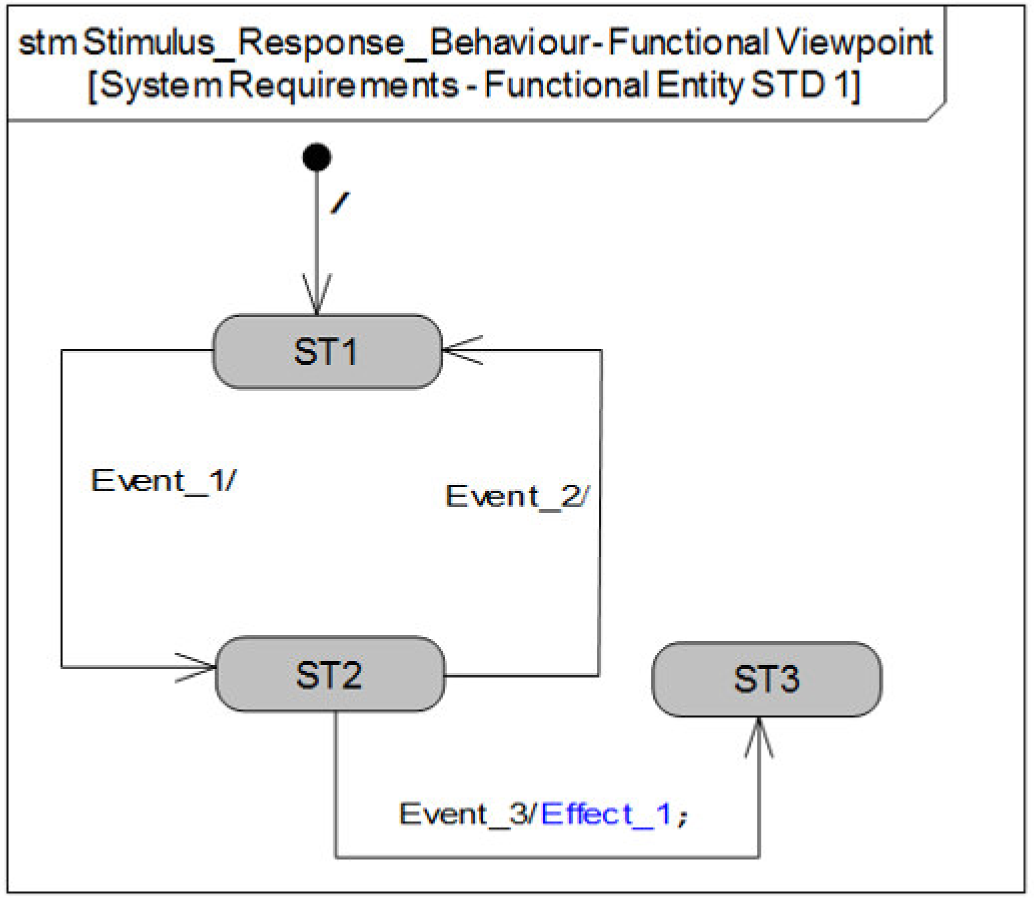
ID	Requirements
Eu.ModIn.587	The value of a port may be set using the name of the port: The syntax is as follows: <port> := <value>; Where: · <port> - specifies the port whose value is being set. · <value> - specifies the value that is being set for the port. **Example:** T1_Cd_Indicate_signal_aspect := true;
Eu.ModIn.588	**6.1.5.5 Calling an operation**
Eu.ModIn.589	To call an Operation item in ASAL, reference the Operation with its default (the default is 'This'). You must use parentheses for the operation, even if there are no parameters to pass. The syntax is as follows: <operation> ([<parameters>]); Where: · <operation> - specifies the operation that is being called. By default, the Operation is called against 'This'. · <parameters> - specifies any parameter values that are passed to the operation that is being called. **Examples:** MyOperation(True); OperationWithNoParameters();
Eu.ModIn.590	**6.1.5.6 Assigning values to variables**
Eu.ModIn.591	Values can be assigned to variables. The syntax is as follows: <variable> := <expression> ; Where: · <variable> - specifies the variable that is being assigned. · <expression> - specifies the value that is being assigned, which can be defined through an expression. **Example:** Mem_ped_wait := False;
Eu.ModIn.592	**6.1.5.7 Conditional execution of code**

ID	Requirements

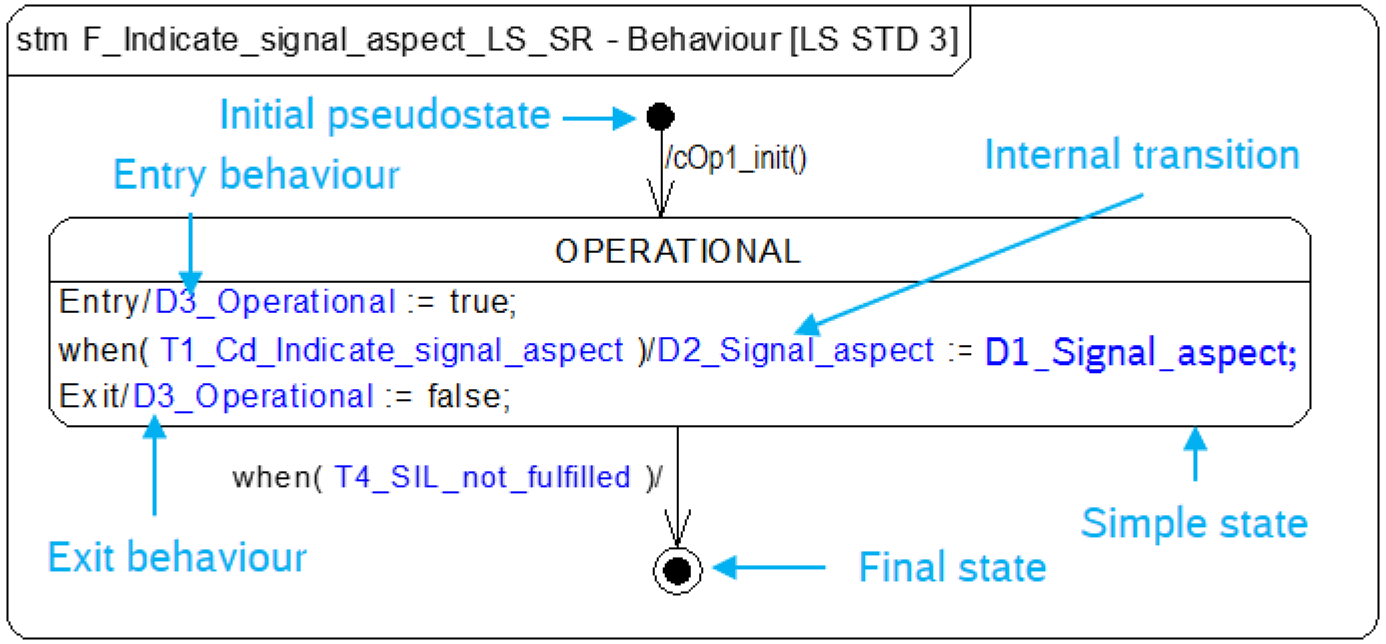
Eu.ModIn.593	The if, then, else statements provide a mechanism for conditional execution of code.
	The syntax is as follows:
	if <condition> then
	... //code to execute
	elseif <condition> then
	... //code to execute
	else
	... //code to execute
	end if
	Where:
	· <condition> - specifies the condition that is being tested.
	
	Example:
	if A < 100 then
	A := A + 1;
	elseif B < 100 then
	B := B + 1;
	else
	NowStop := True;
	end if

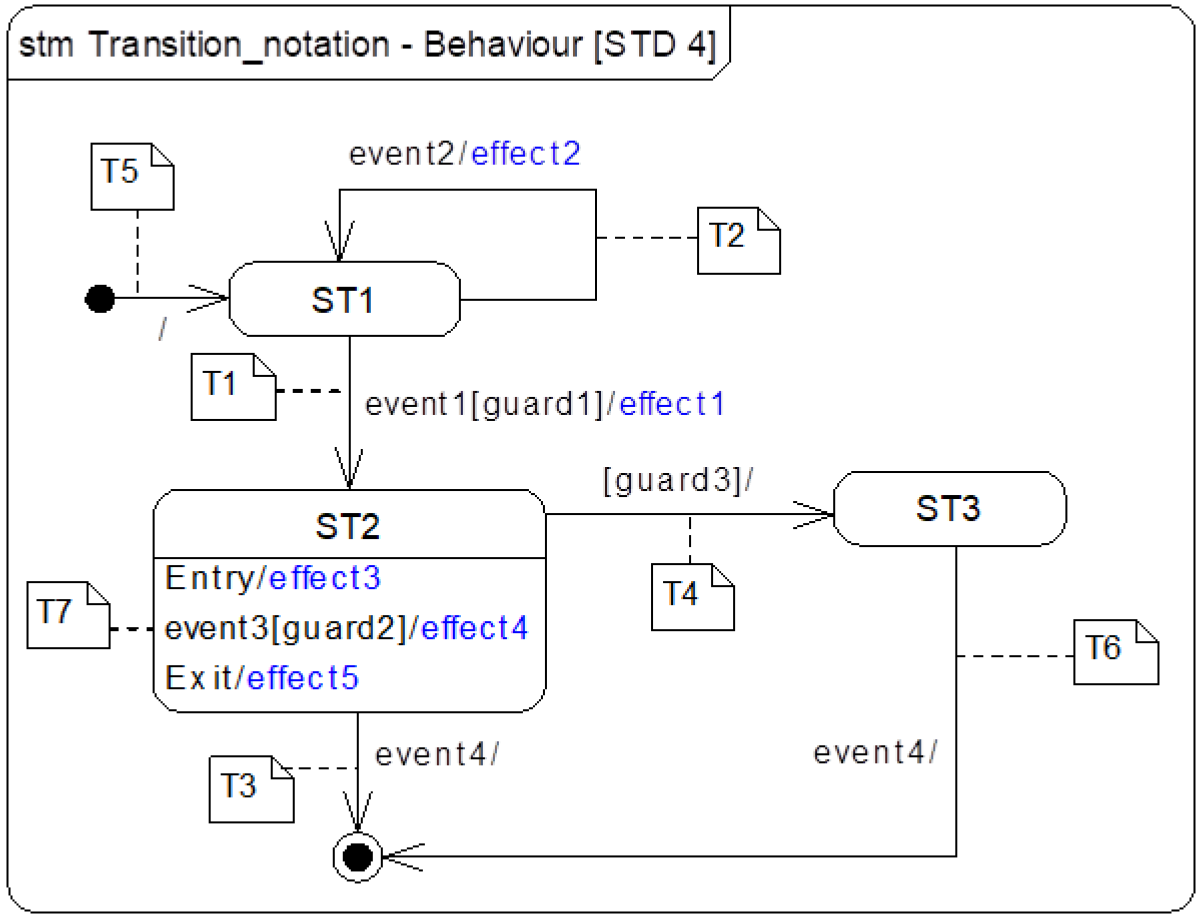
Eu.ModIn.594	**6.1.5.8 While loops**
Eu.ModIn.595	The while loop provides a mechanism for executing code while a condition is true. The syntax is as follows: while <condition> ... //code to execute end while Where: · <condition> - specifies the condition that is being tested. **Example:** while A < 100 A := A + 1; end while
Eu.ModIn.596	**6.1.5.9 Case selection**
Eu.ModIn.597	The case selection provides a mechanism for executing code when a case is true. The syntax is as follows (note that there can be many cases): select case <condition> case <condition>: ... //code to execute case else: ... //code to execute end select Where: · <condition> - specifies the condition that is being tested. **Example:** select case A + B case 200: ResultIs200 := True; case else: ResultIs200 := False; end select
Eu.ModIn.598	**6.1.5.10 Return statement**

ID	Requirements
Eu.ModIn.599	The Return statement can return the result of an expression. The syntax is as follows: return <expression> ; Where: · <expression> - specifies the expression that returns the result. **Example:** return A + B;
Eu.ModIn.317	**6.2 Concept and interpretation of state machines**
Eu.ModIn.739	In the following, the term "Functional Entity" and the corresponding abbreviation "FE" stand for both a Functional Entity and a Technical Functional Entity (TFE).
Eu.ModIn.416	A FE is always in a state that abstracts a combination of values given in the FE. Events arriving at the FE lead to reactions - depending on the state - that change values of SysML out ports or SysML block properties, invoke a local trigger or a call operation or send a signal via a port and result in new states.
Eu.ModIn.417	The state machine diagrams (STD) are children of the state machine and illustrate its behaviour (see *figure 32*), i.e. they describe the stimulus-response behaviour of a FE. The state machine contains states and state transitions that are triggered by trigger in ports, data in ports, internal broadcast events as well as timing events. The state transitions represent the binding functional requirements of the system to be specified.
Eu.ModIn.865	For each STD, a description "Requirements Part 2" that corresponds to the following defined schema: • The SUS or SIUS receives a stimulus and responds with the result to....
Eu.ModIn.866	A possible application of the schema is shown below using the example of the subsystem LS: Information: This state machine diagram describes the requirements for the following functionalities: • receives the observed Signal Aspect and reports this to the Subsystem - Electronic Interlocking • receives the observed intentionally dark state and reports this to the Subsystem - Electronic Interlocking • receives the observed Luminosity and reports this to the Subsystem - Electronic Interlocking
Eu.ModIn.707	Figure 32 Example of a state machine diagram 
Eu.ModIn.418	**6.2.1 Region**
Eu.ModIn.419	Each state machine contains at least one region, which itself can contain a number of states and pseudostates, as well as the transitions between them. During execution of a state machine, each of its regions has a single active state that determines the transitions that are currently viable in that region. A region must have an initial pseudostate and can have a final state that correspond to its beginning and completion, respectively.
Eu.ModIn.420	If a state machine contains a single region, it is represented by the area inside the frame of the state machine diagram and it is not to be named. Multiple regions are named and shown separated by dashed lines. A state machine with multiple regions may describe some concurrent behaviour happening within the state machine's owning block.

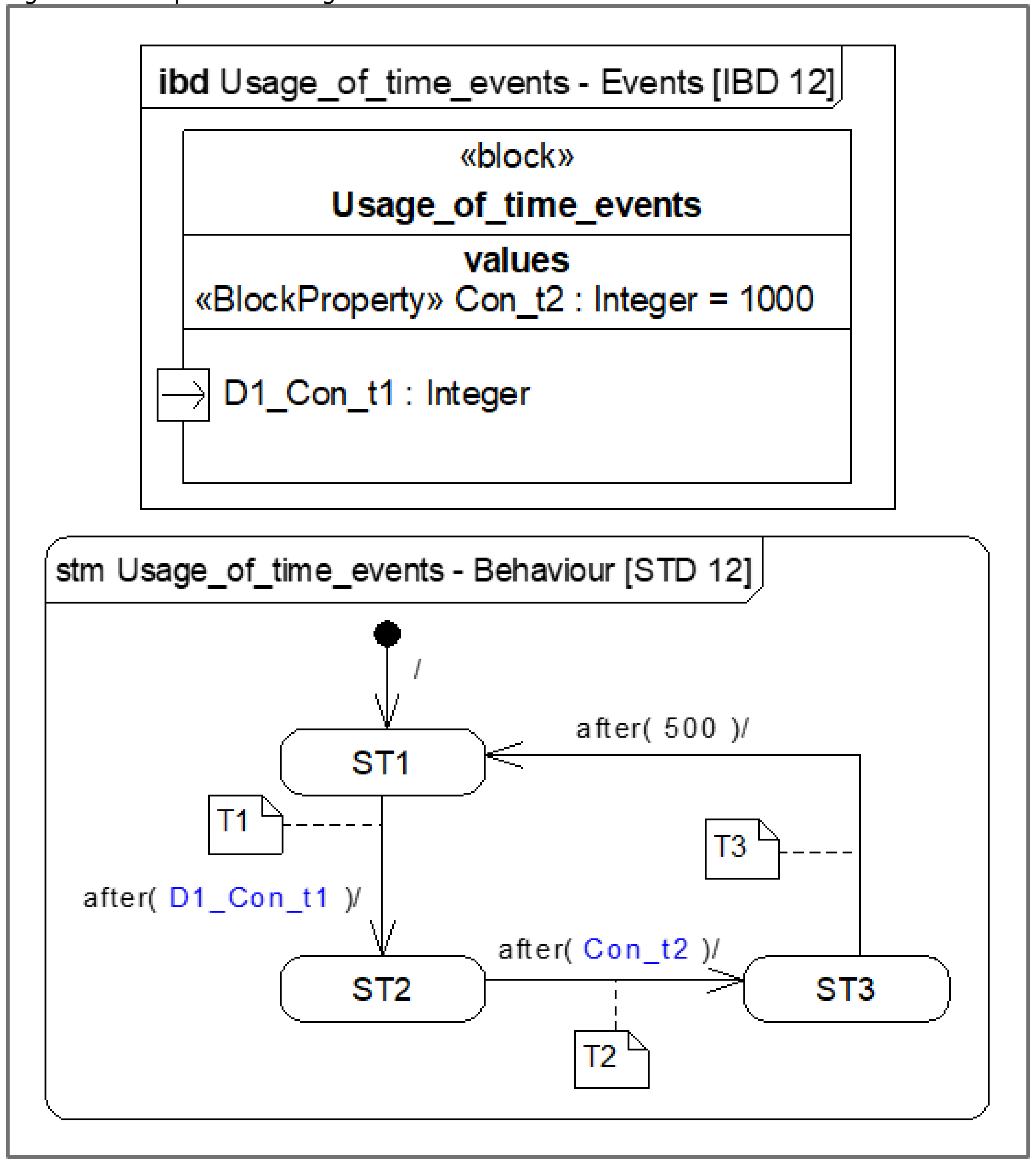
ID	Requirements
Eu.ModIn.421	**6.2.2 State**
Eu.ModIn.422	The UML specification defines a state as „a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external or internal event to occur“. The „object“, in the present case the FE, is waiting for a stimulus from its environment or for an internal stimulus such as a time event or a local trigger.
Eu.ModIn.423	Thus, a state represents a "between stimuli" condition of the external observable stimulus-response behaviour of a FE. In other words, it specifies the responses to incoming stimuli.
Eu.ModIn.424	It is helpful to use the analogy that a block, i.e. the FE, is controlled by a switch. Each state corresponds to a switch position. The state machine defines all valid switch positions (i.e. states) and transitions between switch positions (i.e. state transitions). If there are multiple regions, each region is controlled by its own switch with its switch positions corresponding to its states. The switch positions can be specified by a form of truth table - similar to how logic gates can be specified - in which the current states and transitions define the next state.
Eu.ModIn.425	In the example depicted in *figure 33*, the state ST2 represents a "between stimuli condition", i.e. it constitutes the precondition for triggering a response in the form of Effect_1. Following the analogy that the FE is controlled by a switch, the switch would be positioned to ST2. When Event_3 occurs Effect_1 is executed while the FE changes to state ST3.
Eu.ModIn.427	Figure 33 Example of a state specifying a response 
Eu.ModIn.426	In the EULYNX requirements specification documents there are below the depicted state machine diagrams (as for example depicted in *figure 33*) the corresponding state transitions listed as atomic mandatory functional requirements: **Info | Initial** **Req | {Initial - ST1}** **Info | ST1** **Req | Event_1/{ST1 -ST2}** **Info | ST2** **Req | Event_2/{ST2 -ST1}** **Req | Event_3/Effect_1; {ST2 - ST3}** **Info | ST3**
Eu.ModIn.437	A state is represented on the state machine diagram by a round-cornered box containing its name.

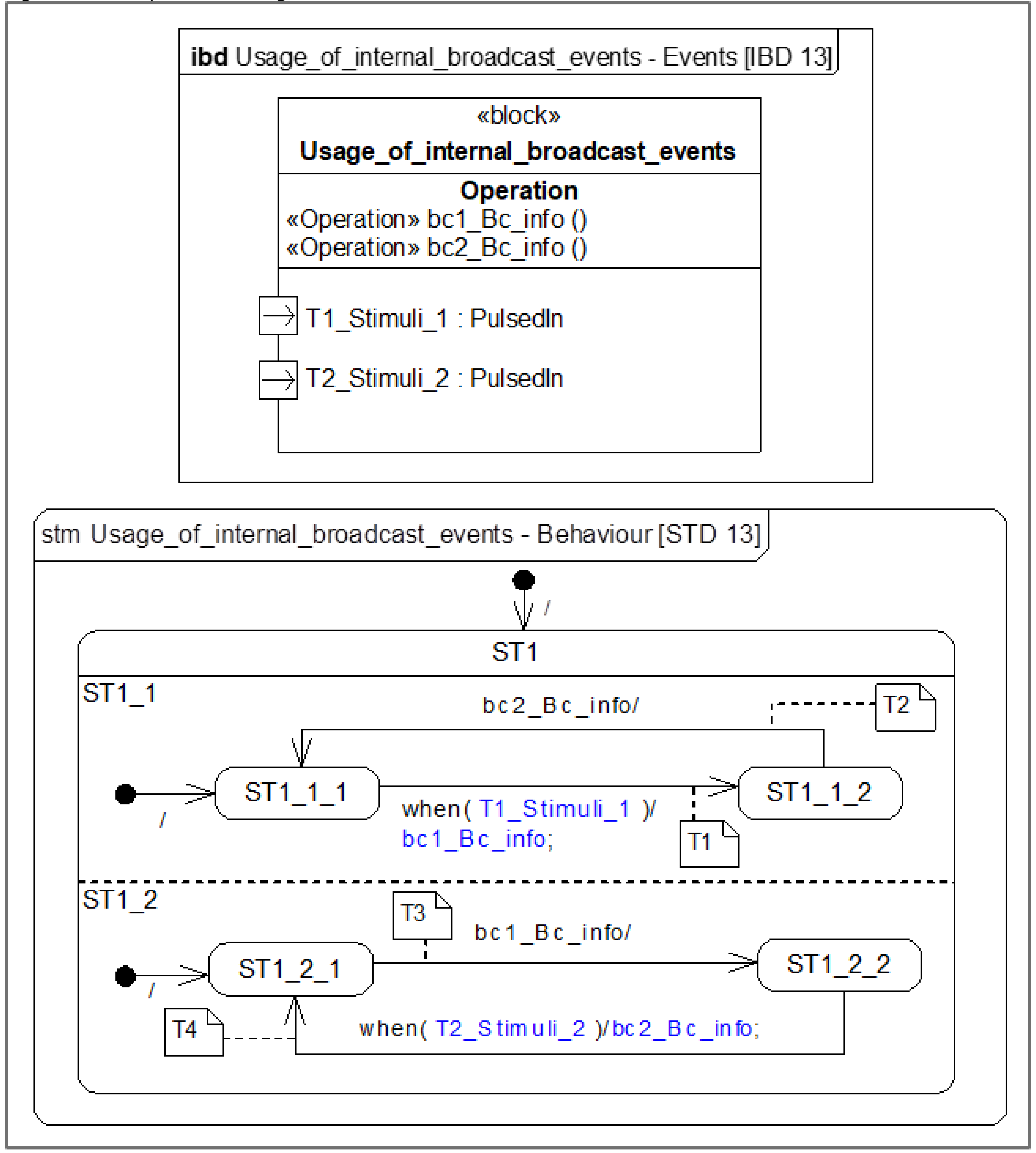
ID	Requirements
Eu.ModIn.431	**Kinds of states:** The following three kinds of states are distinguished: • **simple state** (state with no regions and therefore without nested states), • **sequential state** (state with exactly one region) and • **concurrent state** (state with at least two regions)
Eu.ModIn.428	Each state may contain entry and exit behaviour that are performed whenever the state is entered or exited respectively. Entry and exit behaviour are described as text expressions using the chosen action language preceded by the keywords entry or exit and a forward slash.
Eu.ModIn.429	A state machine can contain transitions, called internal transitions, which do not effect a change in state. An internal transition has the same source and destination and, if triggered, simply executes the transition effect.
Eu.ModIn.430	By contrast, an external transition with the same source and destination state - sometimes called a transition-to-self - triggers the execution of that state's exit and entry behaviour as well as the transition effect.
Eu.ModIn.438	Additional to the states, SysML includes a number of pseudostates to provide additional semantics. The difference between a state and a pseudostate is that a region can never stay in a pseudostate, which merely exists to help determine the next active state.
Eu.ModIn.439	The EULYNX methodology adopts the following SysML pseudostates: • **initial pseudostate**, • **final state**, • **choice pseudostate**, • **fork pseudostate** and • **join pseudostate**.
Eu.ModIn.614	Pseudostates have a defined name, that may be visible on the diagrams.
Eu.ModIn.440	**6.2.3 Initial pseudostate and final state**
Eu.ModIn.441	An initial pseudostate is shown as a filled circle. It is used to determine the initial state of a region (see *figure 34*). The outgoing transition from an initial pseudostate may include an effect. Such effects are often used to set the initial values of properties used by the state machine (e.g. call operation cOp1_init() shown in *figure 34*).
Eu.ModIn.442	A final state is shown as a bulls-eye (i.e. a filled circle surrounded by a larger hollow circle). It indicates that a region has completed execution (see *figure 31*. When the active state of a region is the final state, the region has completed, and no more transitions take place within it. Hence, a final state can have no outgoing transitions.
Eu.ModIn.443	**6.2.4 Choice pseudostate**
Eu.ModIn.444	A choice pseudostate is shown as a white diamond with one transition arriving and two or more transitions leaving. It is used to construct a compound transition path between states. The compound transition allows more than one alternative path between states to be specified, although only one path can be taken in response to any single event.
Eu.ModIn.445	Multiple transitions may either converge on or diverge from the choice pseudostate. When there are multiple outgoing transitions from a choice pseudostate, the selected transition will be one of those whose guard evaluates to true at the time after the choice pseudostate has been reached. This allows effects executed on the prior transition to affect the outcome of the choice.
Eu.ModIn.446	When a choice pseudostate is reached in the execution of a state machine, there must always be at least one valid outgoing transition. If not, the state machine is invalid.
Eu.ModIn.447	If a compound transition contains choice pseudostates, any possible compound transition must contain only one trigger, normally on the first transition in the path.
Eu.ModIn.448	**6.2.5 Fork pseudostate**
Eu.ModIn.452	A fork pseudostate is shown as a vertical or horizontal bar with transition edges either starting or ending on the bar.
Eu.ModIn.449	It has a single incoming transition and as many outgoing transitions as there are orthogonal regions in the target state. Unlike choice pseudostates, all outgoing transitions of a fork are part of the compound transition. When an incoming transition is taken to the fork pseudostate, all the outgoing transitions are taken.
Eu.ModIn.451	Because all outgoing transitions of the fork pseudostate have to be taken, they may not have triggers or guards but may have effects.
Eu.ModIn.450	**6.2.6 Join pseudostate**
Eu.ModIn.456	A join pseudostate is shown as a vertical or horizontal bar with transition edges either starting or ending on the bar.
Eu.ModIn.453	The coordination of outgoing transitions from a concurrent state is performed using a join pseudostate that has multiple incoming transitions and one outgoing transition. The rules on triggers and guards for join pseudostates are the opposite of those for fork pseudostates.

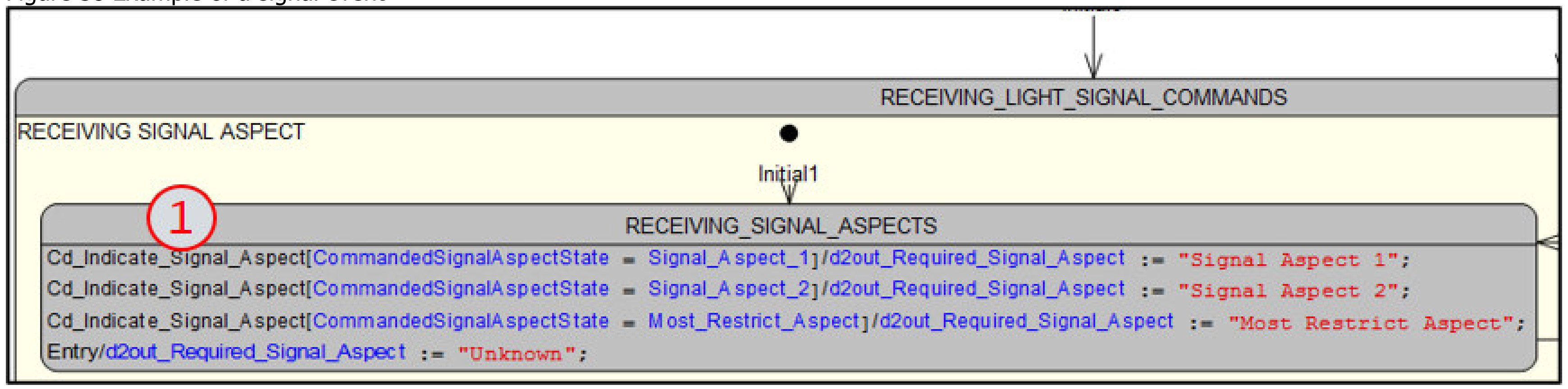
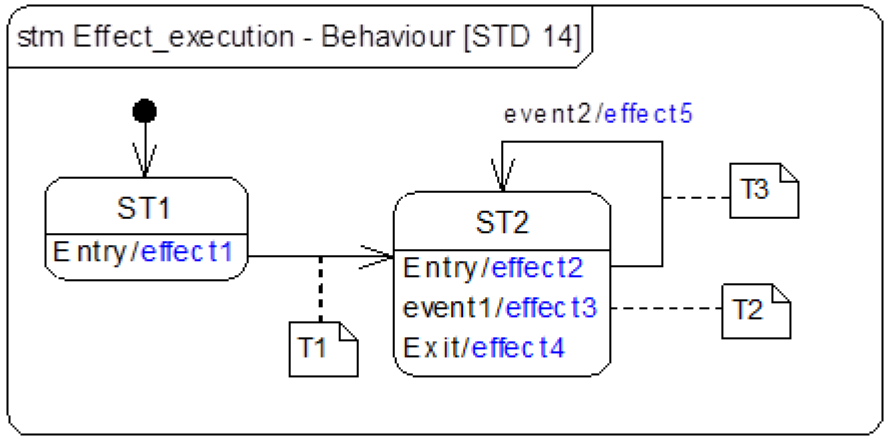
ID	Requirements
Eu.ModIn.454	Incoming transitions of the join pseudostate may not have triggers or a guard but may have an effect. The outgoing transition may have triggers, a guard and an effect.
Eu.ModIn.455	When all the incoming transitions can be taken and the join's outgoing transition is valid, the compound transition can occur. Incoming transitions occur first followed by the outgoing transition.
Eu.ModIn.432	**6.2.7 Simple state**
Eu.ModIn.433	As shown in the examples depicted in *figure 33* (states ST1, ST2, ST3) and *figure 34* (state "OPERATIONAL"), a simple state has no regions and therefore no nested states.
Eu.ModIn.434	A simple state may, like any kind of state, contain entry behaviour, that is executed immediately upon entering the state, exit behaviour, that is executed immediately before exiting the state, and behaviour executed during internal transitions. (see *figure 34*). All three kinds of behaviour are not interruptible.
Eu.ModIn.435	*Figure 34* shows a simple example of a FE defining the functionality "Indicate signal aspect" of a light signal (LS) with a single OPERATIONAL state in its single region. A transition from the region's initial pseudostate goes to the OPERATIONAL state. On entry, the light signal indicates that it is operational, setting the value of the out port "D3_Operational" to true, and on exit it indicates a non operational status, setting the value of "D3_Operational" to false. While the light signal is in the state OPERATIONAL, it may receive commands to indicate a transmitted signal aspect (T1_Cd_Indicate_signal_aspect) and indicate it (D2_Signal_aspect). When in the OPERATIONAL state, the intrasystem event "T4_SIL_not_fulfiled" triggers a transition to the final state, and because there is only one single region, the state machine terminates.
Eu.ModIn.436	Figure 34 Example of a simple state 
Eu.ModIn.457	**6.2.8 Transition**
Eu.ModIn.458	A transition specifies a change of state within a state machine. It is a directed relationship between a source and a destination state, and defines an event (trigger) and a guard (condition) that both lead to the state transition, as well as an effect (behaviour) that is executed during the transition. Source and destination can be the same state (see T2 in *figure 35*).
Eu.ModIn.460	**Run to completion:** State machines always run to completion, which means that they are not able to consume another event until the state machine has completed the processing of the current event. Thus, the next event will be consumed only if all effects (behaviour) of the previous event have been completed.
Eu.ModIn.559	Run to completion does not mean that a state machine owned by a FE interconnected with neighbouring FE monopolises all FEs in this network until the run to completion step is complete. The preemption restriction only applies to the context of the corresponding FE.
Eu.ModIn.461	An event that cannot be consumed, for example because there is no matching transition, is discarded.
Eu.ModIn.462	**Transition notation:** A transition is shown as an arrow between two states, with the head pointing to the target state.
Eu.ModIn.463	Transitions-to-self are shown with both ends of the arrow attached to the same state (see T2 in *figure 35*).
Eu.ModIn.464	Internal transitions are not shown as graphical paths but are listed on separate lines within the state symbol (see T7 in *figure 35*).

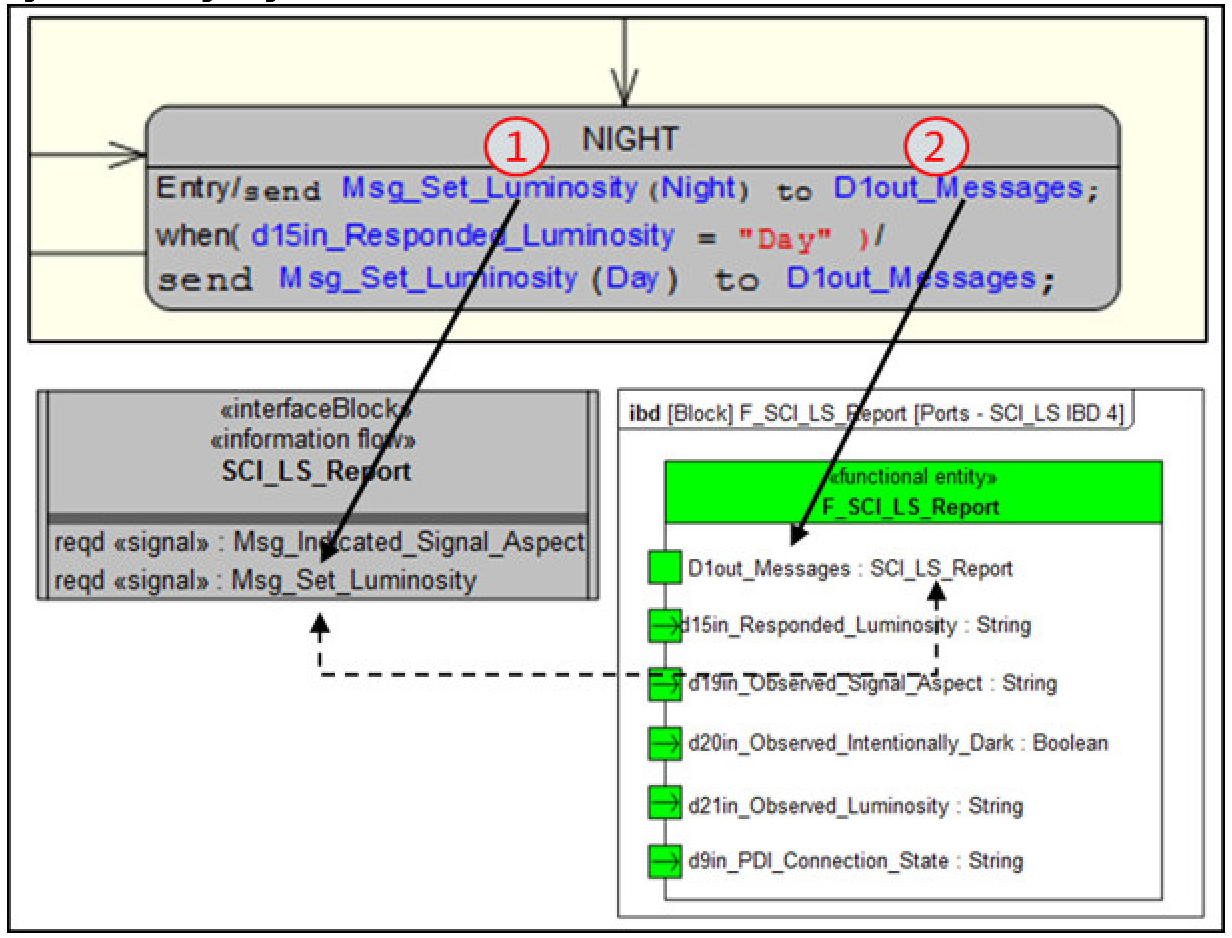
ID	Requirements
Eu.ModIn.465	The definition of the transition's behaviour is shown in a formatted string on the transition with the event first, followed by a guard in square brackets, and finally the transition effect preceded by a forward slash (event-effect block or even-action block). As shown in *figure 35*, any or all of the behavioural elements as event, guard and effect may be omitted. In T5 for example, all the behavioural elements are omitted. Transition **T3**, to give another example, is only triggered by an event without guard and effect.
Eu.ModIn.466	**Event:** An event specifies some occurrence that can be measured with regard to location and time and causes a transition to occur. Descriptions of the triggering events are provided in chapter 6.2.9 "Event"**.**
Eu.ModIn.467	**Guard:** The transition guard contains an expression that must evaluate true in the moment of the triggering event so that the transition is performed (see T1, T4 and T7 in *figure 35*). The guard is specified using a constraint which includes an expression formulated in the applied action language to represent the guard condition. If preceded by an event (see T1 and T7 in *figure 35*) and if the event satisfies a trigger, the guard on the transition is evaluated. If the guard evaluates to true, the transition is triggered; if the guard evaluates to false, then the event is consumed with no effect.
Eu.ModIn.468	Transitions can also be triggered by internally generated completion events. For a simple state a completion event is generated when the entry behaviour (for example Entry/effect3 in *figure 35*) has completed.
Eu.ModIn.469	Thus, where a guard is shown without a preceding event (see T4 in *figure 35*), the guard condition is evaluated immediately after entering the source state, i.e. after its entry behaviour has completed, and a transition takes place if true, triggered by the generated completion event of the source state.
Eu.ModIn.470	**Please note:** if the guard condition of a transition without trigger changes to true while the state machine is already in the source state (for example in state ST2), the guard condition won't be evaluated and no transition will take place.
Eu.ModIn.471	**Effect:** The effect is a behaviour executed when entering or exiting a state (entry and exit behaviour, respectively), during an internal transition (see T7 in *figure 35*) and during the external transition from one state to another (see T1 in *figure 35*). If an external transition is triggered, first the exit behaviour of the current (source) state, then the transition effect and finally the entry behaviour of the target state are executed. Descriptions of the effects used in the methodology underlying this Modelling standard are provided in chapter 6.2.14 "Effect".
Eu.ModIn.472	A transition may also be formulated textually as atomic functional requirement: **Event [Guard]/Effect {Source state - Target state}**.
Eu.ModIn.459	Figure 35 Transition notation 
Eu.ModIn.473	**6.2.9 Event**

ID	Requirements
Eu.ModIn.474	An event specifies some occurrence that can be measured with regard to location and time and causes a transition to occur.
Eu.ModIn.475	In the EULYNX methodology, the following types of events are used: • **Change event**, • **Time event** • **Internal broadcast event** • **Signal event**.
Eu.ModIn.476	**6.2.9.1 Change event**
Eu.ModIn.477	A change event indicates that some condition has been satisfied, that is, the value of a specified Boolean expression holds. A defined change event occurs during system operation each time the specified Boolean expression toggles from false to true. Change events are continuously evaluated.
Eu.ModIn.478	According to the EULYNX methodology, the Boolean expression of a change event may contain the following arguments: • **Data In Port,** • **block property** • **block operation.**
Eu.ModIn.479	**Notation of change events:** Change events use the term „when“ followed by the Boolean expression that has to be met in parenthesis. Like other constraint expressions, the Boolean expression is to be expressed in text using the applied action language: **when(boolean expression)[guard]/effect;**
Eu.ModIn.480	**6.2.9.2 Time event**
Eu.ModIn.481	A time event indicates that a given time interval has passed since the current state was entered.
Eu.ModIn.482	**Notation of time events:** Time events use the term "after“ followed by the time period (in milliseconds by default) in parenthesis, e.g. after(D1_Con_t1) as depicted in *figure 36*.
Eu.ModIn.484	"after" indicates that the time is relative to the moment the state is entered. The transition T1 shown in *figure 36* is, for example, triggered after the time D1_Con_t1 has expired. The time starts on entering the state ST1.

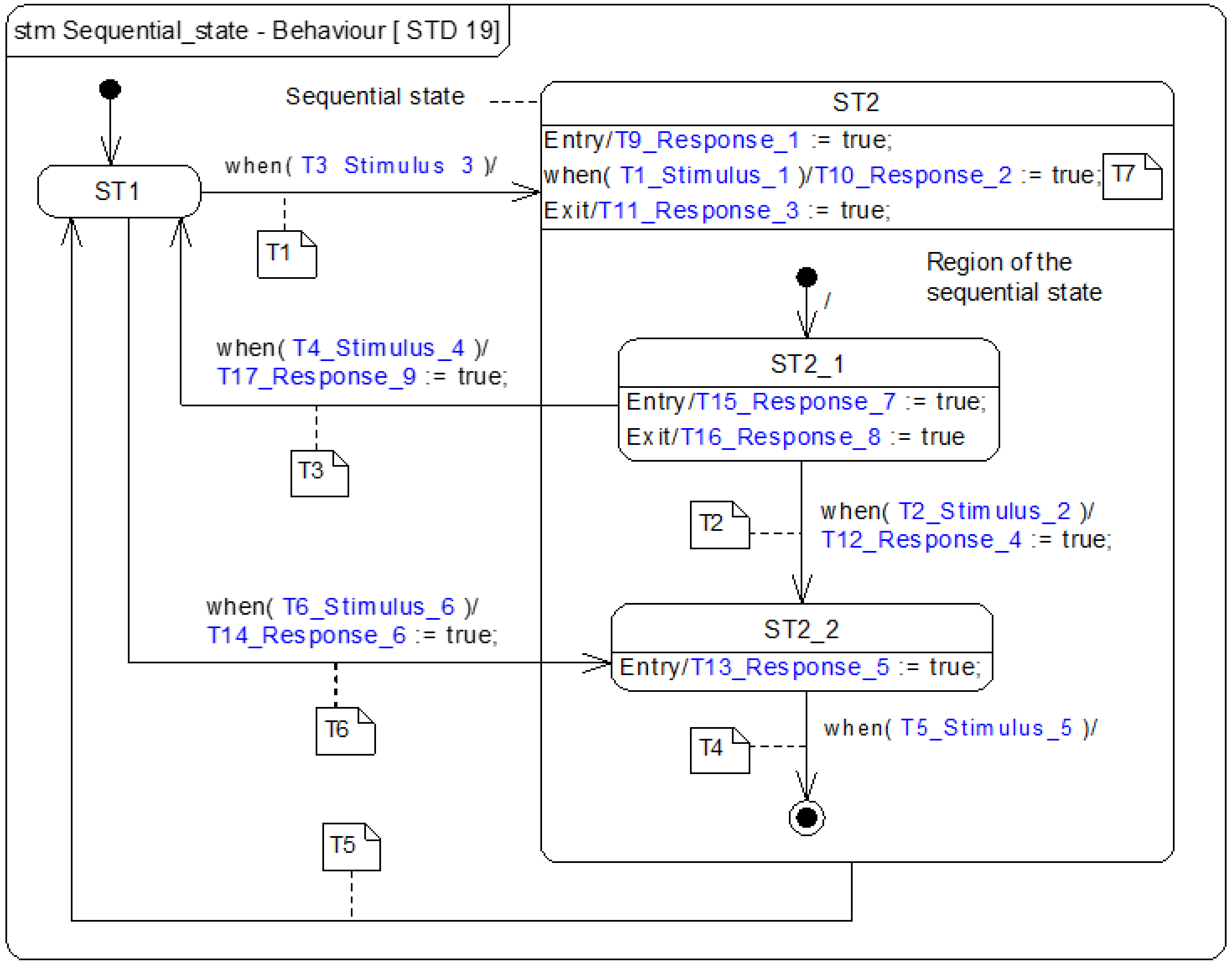
ID	Requirements
Eu.ModIn.483	Figure 36 Example of the usage of time events 
Eu.ModIn.485	**6.2.9.3 Internal broadcast event**
Eu.ModIn.487	Internal broadcast events occur when corresponding SysML block operations are invoked. They are supposed to submit broadcasts within the state machine of a FE.
Eu.ModIn.488	In *figure 37* for example, the SysML block operations bc1_Bc_info() and bc2_Bc_info() represent internal broadcast events. During transition T1, the internal broadcast event bc1_Bc_info() is invoked in order to trigger transition T3. Furthermore, during transition T4, the internal broadcast event bc2_Bc_info() is invoked to trigger transition T2.

ID	Requirements
Eu.ModIn.489	Figure 37 Example of the usage of internal broadcast events 
Eu.ModIn.699	**6.2.9.4 Signal event**
Eu.ModIn.700	A signal event is generated when a reception of an interface block receives a signal. This is then used in the state model to trigger a state transition **(1)**.

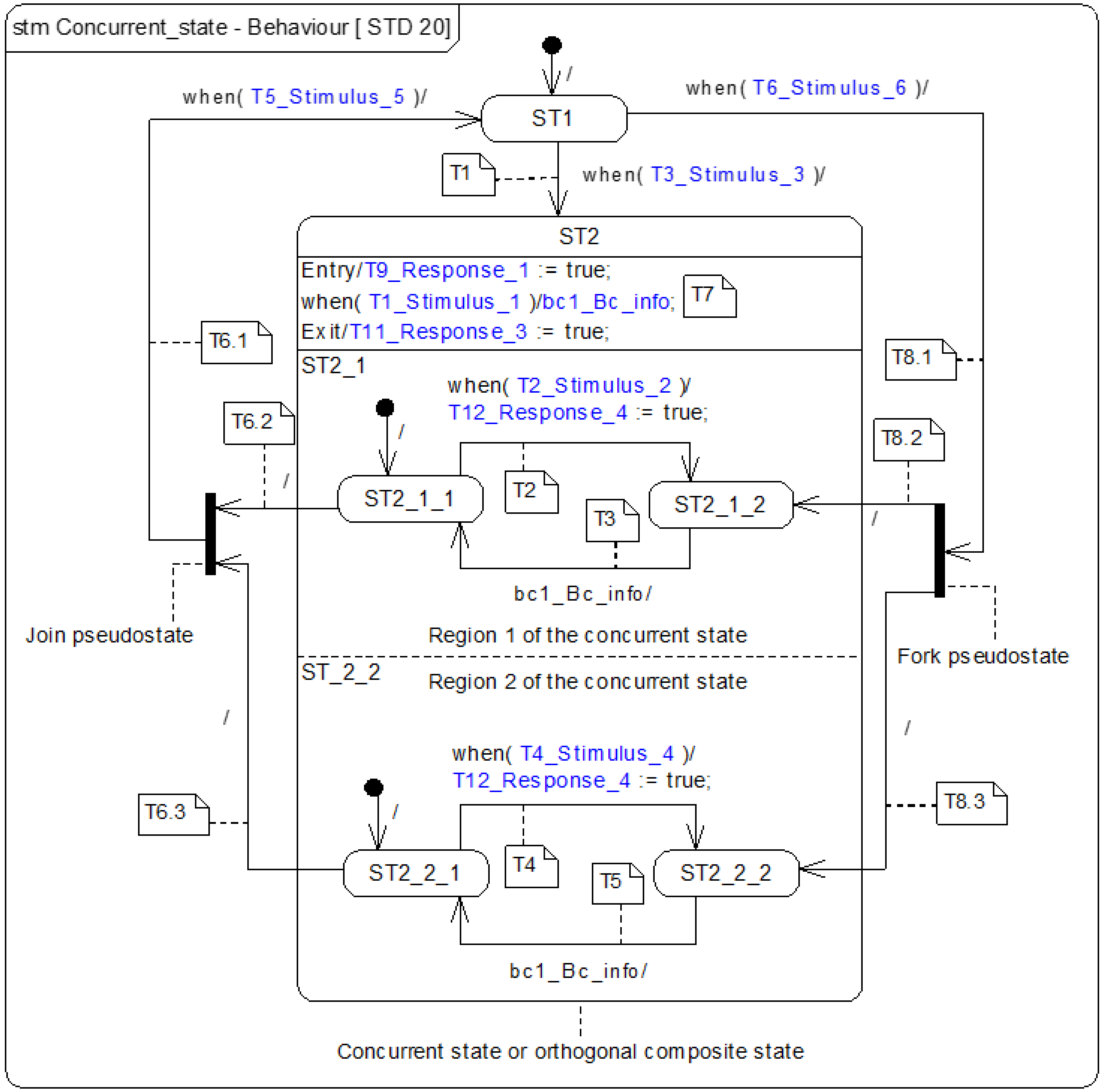
ID	Requirements
Eu.ModIn.701	Figure 38 Example of a signal event 
Eu.ModIn.490	**6.2.10 Effect**
Eu.ModIn.492	An effect is a behaviour executed when entering or exiting a state (entry and exit behaviour, respectively), during an internal transition or during an external transition from one state to another.
Eu.ModIn.493	The sequence of effect execution is demonstrated in *figure 39*. Transition T1 is taken immediately on completion of effect1. The sequence of effect execution when event2 occurs (T3) is: effect4, then effect5, then effect2. Event1 generates only one effect (T2): effect3.
Eu.ModIn.494	Figure 39 Sequence of effect execution 
Eu.ModIn.495	The following elements of behaviour may be represented as **effect**: • **Event-driven responses using signals,** • **Responses in form of continuous flows,** • **Call behaviour.**
Eu.ModIn.703	**6.2.10.1 Event-driven responses using signals**
Eu.ModIn.496	As shown in Figure 40, signals **(1)** are sent as an effect of a state transition or triggered in a block operation via the corresponding port **(2)** of the respective FE.

ID	Requirements
Eu.ModIn.702	Figure 40 Sending a signal 
Eu.ModIn.704	**6.2.10.2 Responses in form of continuous flows**
Eu.ModIn.503	A response is sent in form of a continuous flow by assigning the desired value to a data out port, e.g. D1out_Temperature := 40.
Eu.ModIn.504	All out ports are initialised. The initialisation can be carried out in the body of the init-block operation systematically named cOp1_init(). Alternatively it can be carried out directly in the transition effect of the transition outgoing from initial state of the state machine.
Eu.ModIn.505	Furthermore, the sender of a response must always configure the current value of the Data Out Port.
Eu.ModIn.705	**6.2.10.3 Call behaviour**
Eu.ModIn.506	Call behaviour is invoked on demand, executed and terminated after execution. It is supposed to define event-driven data transformations. The algorithm of the data transformations is to be described in the body of the corresponding block operation.
Eu.ModIn.507	**6.2.11 Composite state**
Eu.ModIn.508	States can have regions. Such states are called composite states or hierarchical states. They allow state machines to scale to represent state-based behaviour of any complexity. A composite state may have one single region (sequential state) but also multiple orthogonal regions (concurrent state or orthogonal composite state).
Eu.ModIn.509	Instead of using a region to decompose the behaviour of a state, a state machine diagram may be assigned to the corresponding state alternatively, defining its behaviour.
Eu.ModIn.510	Each region or state machine diagram assigned to a state has a set of mutually exclusive disjoint subvertices and a set of transitions. In other words, it typically will contain an initial pseudostate and a final state, a set of pseudostates, and a set of substates, which may themselves be composite states.
Eu.ModIn.511	Any state enclosed within a region of a composite state is called a substate of that composite state.
Eu.ModIn.512	**6.2.12 Sequential state**
Eu.ModIn.513	A sequential state, such as ST2 shown in the example depicted in *figure 41,* is a composite state that has one region.

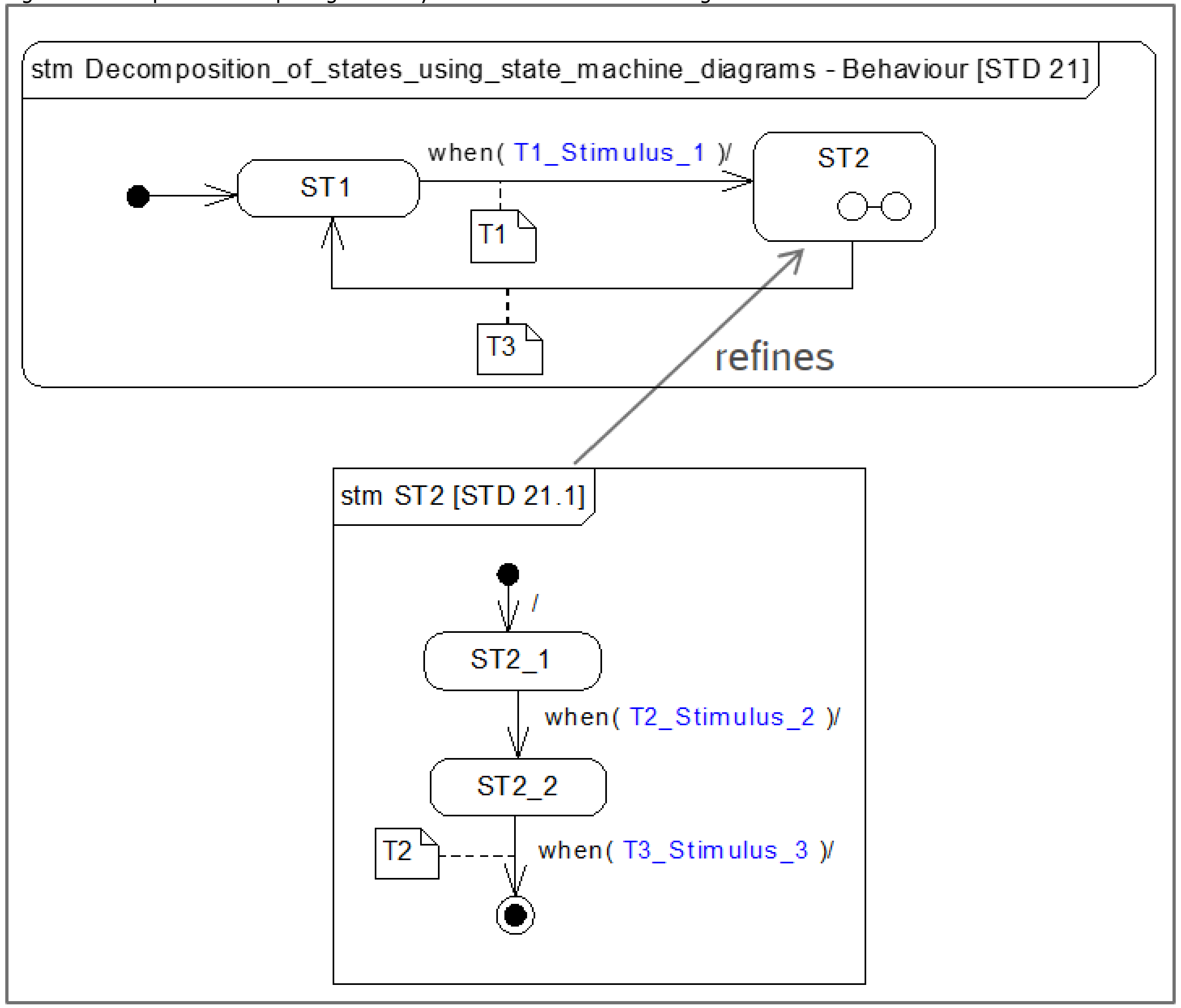
ID	Requirements
Eu.ModIn.515	*Figure 41* shows the decomposition of the state ST2 into the substates ST2_1 and ST2_2. On entry to the state ST2, two entry behaviours are executed: the entry behaviour of ST2, T9_Response_1 := true and then the entry behaviour of ST2_1, T15_Response_7 := true. This is because on entry, as indicated by the initial pseudostate, the initial substate of ST2 is ST2_1.
Eu.ModIn.516	When in state ST2_1, T2_Stimulus_2 will cause the transition T2 to the state ST2_2 and will successively process T16_Response_8 := true, T12_Response_4 := true and T13_Response_5 := true. If T5_Stimulus_5 is received while in state ST2_2, the change event will trigger the transition T4 to the final state. A completion event is generated when the final state is reached, triggering the transition T5 to state ST1. When leaving ST2, T11 _Response_3 := true is executed.
Eu.ModIn.517	A composite state (sequential state or concurrent state) may be porous, which means transitions such as transition T3 and T6 shown in *figure 41* may cross the state boundary, starting or ending on states within its regions.
Eu.ModIn.518	In the case of a transition ending on a nested state, such as transition T6 shown in *figure 41*, the behaviours are executed in this order: **1.** the effect T14_Response_6 := true of the transition T6, **2.** the entry behaviour T9_Response_1 := true of the composite state, **3.** the entry behaviour T13_Response_5 := true of the transition's target nested state.
Eu.ModIn.519	In the opposite case, such as transition T3 shown in *figure 41*, the behaviours are exited in this order: **1.** the exit behaviour T16_Response_8 := true of the source nested state, **2.** the exit behaviour of the composite state T11_Response_3 := true is executed, **3.** the transition effect T17_Response_9 := true.
Eu.ModIn.520	In the case of more deeply nested state hierarchies, the same rule can be applied recursively to all the composite states whose boundaries have been crossed.
Eu.ModIn.521	If T1_Stimulus_1 is received while in state ST2, the change event will trigger the internal transition T7 and the effect T10_Response_2 := true will be executed without a change of state.
Eu.ModIn.514	Figure 41 Example of a sequential state

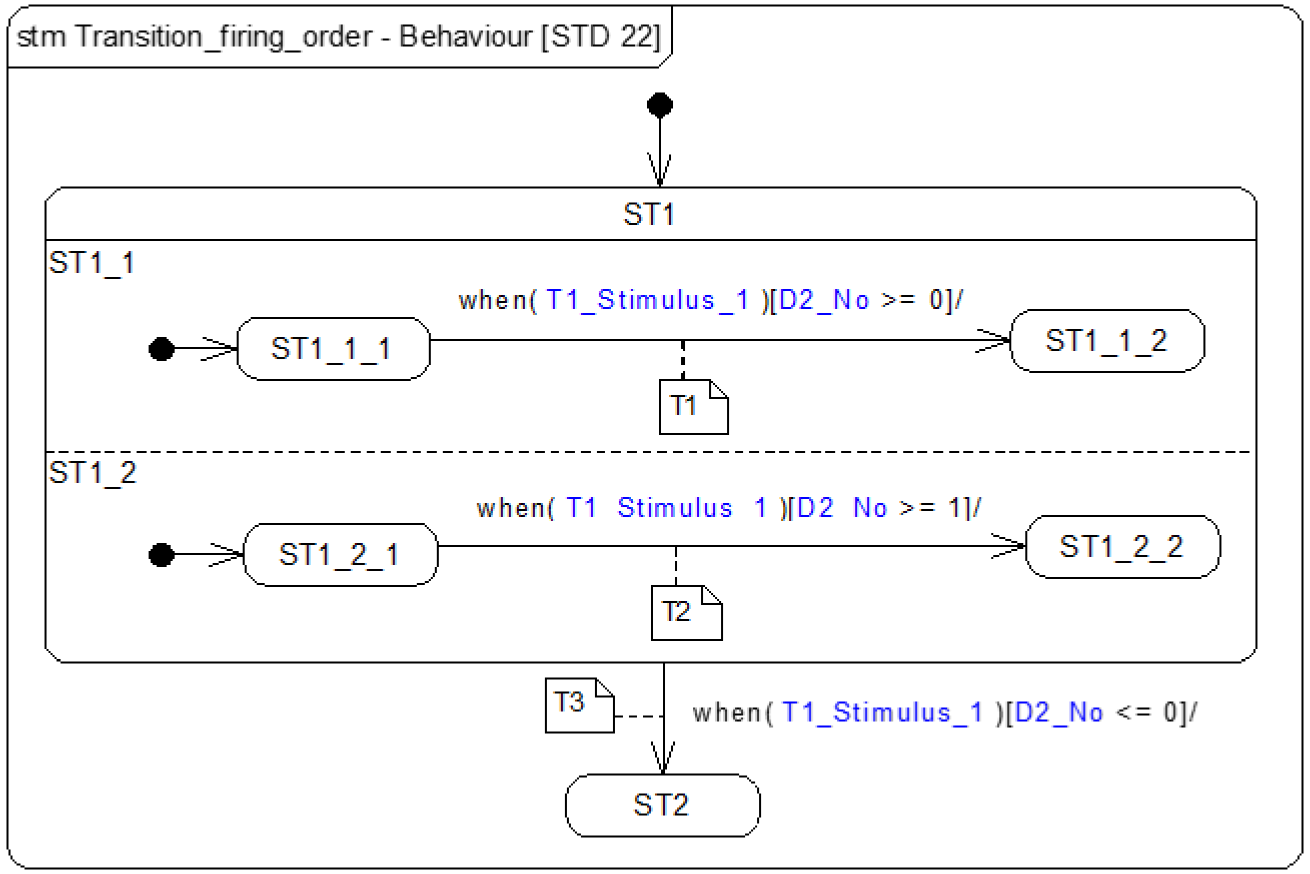


ID	Requirements
Eu.ModIn.522	**6.2.13 Concurrent state**
Eu.ModIn.524	A concurrent state as shown in *figure 42*, sometimes also called an orthogonal composite state, contains at least two regions.
Eu.ModIn.526	When a concurrent state is active, each region has its own active state that is independent of the others, and any incoming event is independently analysed within each region.
Eu.ModIn.527	A transition that ends on the concurrent state, such as transition T1 in *figure 42*, will trigger transitions from the initial pseudostate of each region, so there must be an initial pseudostate in each region for such a transition to be valid.
Eu.ModIn.528	Similarly, a completion event for the concurrent state will occur when all the regions are in their final state.
Eu.ModIn.529	When an event, as for example the internal broadcast event bc1_Bc_info shown in *figure 42*, is associated with triggers in multiple orthogonal regions, the event may trigger a transition in each region (e.g. transitions T3 and T5), assuming the transition is valid based on the other usual criteria.
Eu.ModIn.530	**Please note:** a transition can never cross the boundary between two regions of the same concurrent state.
Eu.ModIn.531	In addition to transitions that start or end on the concurrent state, such as transition T1 in *figure 42*, transitions from outside the concurrent state may start or end on the nested states of its regions. In this case, one state in each region must be the start or end of one of a coordinated set of transitions. This coordination is performed by a fork pseudostate in the case of incoming transitions, such as T8.1, T8.2 and T8.3 in *figure 42*, and a join pseudostate for outgoing transitions, such as T6.1, T6.2 and T6.3 in *figure 42*.

ID	Requirements
Eu.ModIn.525	Figure 42 Example of a concurrent state 

ID	Requirements
Eu.ModIn.532	**6.2.14 Decomposition of states using state machine diagrams**
Eu.ModIn.533	Instead of decomposing the behaviour of a state within a region of a sequential state or multiple regions of a concurrent state, the behaviour may alternatively be specified by a state machine diagram assigned to the corresponding state (see *figure 43*).
Eu.ModIn.534	The region of the corresponding state machine diagram typically will contain an initial pseudostate and a final state, a set of pseudostates, and a set of substates, which themselves may be decomposed by state machine diagrams.
Eu.ModIn.535	As illustrated in *figure 43*, a transition (e.g. transition T1) ending on a state (e.g. state ST2) that is refined by a state machine diagram will trigger the transition from the initial pseudostate of the diagram to its initialising state (e.g. state ST2_1).

ID	Requirements
Eu.ModIn.536	Similarly, when the behaviour specified on the state machine diagram completes (e.g. the final state is entered after triggering the transition T2), it will generate a completion event that can trigger transitions (e.g. transition T3) whose source is the state (e.g. state ST2) the state machine diagram is assigned to.
Eu.ModIn.537	Figure 43 Principle of decomposing states by means of state machine diagrams 
Eu.ModIn.538	**6.2.15 Transition firing order in nested state hierarchies**
Eu.ModIn.539	The same event may trigger transitions at several levels in a state hierarchy, and with the exception of concurrent regions, only one of the transitions can be taken at a time. Priority is given to the transition whose source state is innermost in the state hierarchy.
Eu.ModIn.540	Suppose the state machine depicted in *figure 44* is in its initial state (i.e. in state ST1_1_1 and ST1_2_1). The stimulus T1_Stimulus_1 is associated with the triggers of the transitions T1, T2 and T3, each with guards based on the value of D2_No.
Eu.ModIn.541	The following list shows the transitions that will fire upon receipt of T1_Stimulus_1 based on values of D18_No from -1 to 1 if the system is in the states ST1_1_1 and ST1_2_1: • D2_No equals -1: transition T3 will be triggered because it is the only transition with a valid guard; • D2_No equals 0: transition T1 will be triggered because, although transition T3 also has a valid guard, state ST1_1_1 is the innermost of the two source states; or • D2_No equals 1: both transitions T1 and T2 will be triggered because both their guards are valid.
Eu.ModIn.542	The normal rules for execution of exit behaviour apply, so, before the transition from state ST1 to state ST2 can be taken, any exit behaviour of the active nested states of state ST1, as well as the exit behaviour of state ST1, must be executed.

ID	Requirements
Eu.ModIn.543	Figure 44 Illustration of transition firing order 
Eu.ModIn.830	**6.2.16 Interaction between state machines**
Eu.ModIn.831	State machines in different blocks, may interact with one another by sending stimuli and returning responses. For example, the state machine of one block can send a stimulus to another block as part of a transition effect or state behaviour. The event corresponding to the receipt of this stimulus by the receiving block can trigger a state transition in its state machine.
Eu.ModIn.832	Thus, different behaviour, each specifying a certain functionality of the system, may be encapsulated in blocks and interconnected with each other in a network of FEs or TFEs, i.e. in a Functional or Technical Functional Architecture.
Eu.ModIn.824	**6.2.17 Binding** (see *chapter 2.1*)
Eu.ModIn.825	**Diagram of model view "Functional Entity" (ibd and stm)** has a "**Def**" binding.
Eu.ModIn.826	**Diagram of model view "Technical Functional Entity" (ibd and stm)** has a "**Req**" binding.
Eu.ModIn.827	The algorithm defined in a **time advanced operations** has a "**Req**" binding. The algorithm defined in a time advanced operation represents the mandatory externally visible behaviour of a FE or TFE in place of or in cooperation with a state machine.
Eu.ModIn.828	**Transitions, states, ports, block operations and block properties** have "**Def**" bindings.