

# Specification of Embedded Systems

## Summer Semester 2020

### Session 2

## Modelling Interfaces and Structure

Jan Peleska  
peleska@uni-bremen.de  
Issue 1.1  
2020-05-04

**Note.** These lecture notes are free to be used for non-commercial educational purposes. I did my best to provide scientifically sound material, but no guarantees whatsoever are given regarding correctness or suitability of the content for any specific purpose.

All rights reserved © 2020 Jan Peleska

# Chapter 1

## Preface

In this document, the material for **Session 2** of the course **Specification of Embedded Systems** is provided.

This document is structured as follows.

[Overview](#)

- We give a short overview over the language structure of SysML in Chapter 2, as far as it is helpful for the material presented in Session 2.
- In Chapter 3, a typical package structure for system models is introduced.
- Structural modelling in general is introduced in Chapter 4.
- As a first application of structural modelling, the **context** of a the system to be developed, that is, its interfaces to the operational environment, are modelled by means of a block definition diagram and an internal block diagram, each diagram containing blocks, ports, and various associations. This is explained in Chapter 5.
- SysML contains a comprehensive concept for introducing physical entities (time, electrical current, ...) and their concrete units (ms, mA, ...). It is explained in Chapter 6, how this is introduced for each project in a systematic way.

# Contents

<b>1</b>	<b>Preface</b>	<b>1</b>
<b>2</b>	<b>A Short Overview of the SysML Language Structure</b>	<b>5</b>
2.1	Structural Constructs . . . . .	5
2.2	Behavioural Constructs . . . . .	6
2.3	Crosscutting Constructs . . . . .	7
<b>3</b>	<b>Package Structure for System Models</b>	<b>8</b>
<b>4</b>	<b>Structural Modelling With SysML</b>	<b>10</b>
4.1	Blocks . . . . .	10
4.2	Ports and Flows . . . . .	13
4.3	Block Definition Diagrams . . . . .	14
4.4	Internal Block Diagrams . . . . .	15
<b>5</b>	<b>The Context Submodel</b>	<b>16</b>
5.1	Context Model Ingredients . . . . .	16
5.2	Context BDD . . . . .	16
5.3	Context IBDs . . . . .	18
<b>6</b>	<b>Introducing Enumerations, Physical Entities and Units</b>	<b>20</b>
<b>7</b>	<b>Questions and Exercises</b>	<b>22</b>
7.1	Questions . . . . .	22
7.1.1	Associations Modelling Java Attributes . . . . .	22
7.2	Exercises . . . . .	22
7.2.1	Use Cases for Requirements . . . . .	22
7.2.2	Interfaces and Internal Block Diagrams for the System Context . . . . .	23

7.2.3	Block Definition Diagram for the Turn Indication Controller . . . . .	23
7.2.4	Internal Interfaces and Internal Block Diagram for the Turn Indication Controller . . . . .	23

# List of Figures

5.1	Context BDD for the turn indication function. . . . .	17
5.2	Top-level context IBD (incomplete) for the turn indication function. . . . .	18
5.3	Dashboard IBD. . . . .	19
6.1	Project types for the turn indication controller. . . . .	21

# Chapter 2

## A Short Overview of the SysML Language Structure

Like most of the system or software modelling formalisms, the SysML has language constructs for specifying structure and behaviour. An important addition which is *not* available in other formalisms is the explicit introduction of crosscutting constructs.

### 2.1 Structural Constructs

The basic element to express structure in SysML is the **block** which can be used to represent both hardware and software entities of the system to be modelled. Just like classes in the UML, blocks may represent a single instance of a system component to be developed, or represent a **type** of which several **instances** can occur inside the system. Blocks may contain other blocks, thereby allowing the representation of system structures where subparts are contained in superparts. For making interfaces explicit, **ports** may be connected to blocks, and the transport of data is modelled by means of **flows** connecting ports.

Blocks,  
ports,  
flows

A special variant of blocks are **constraint blocks**; these are used to encapsulate logical conditions – from physical laws to application-specific rules – for restricting the structural or behavioural properties in a well-defined and re-usable way.

Constraint  
blocks

To facilitate the analysis of system structure, the modelling elements named above can be graphically depicted and related to each other in 3

Diagrams

types of diagrams.

**Block definition diagram (BDD)** This diagram contains blocks and associated relationships and is used to depict the top-down decomposition of a system in an intuitive way.

**Internal block diagram (IBD)** This diagram depicts how instances of various blocks exchange data via ports and flows.

**Parametric diagram** This diagram relates general constraints specified in constraint blocks to concrete parameters of behavioural constructs, visualising where the constraints are applied to.

It should be noted that SysML diagrams do not really add structural or behavioural semantics to a model, because this is completely expressed by the textual representation of model constructs.<sup>1</sup>

Diagrams  
do not  
add  
semantics

In this session, we will study blocks, ports, and flows in detail, together with bdd's and IBDs. Constraint blocks and parametric diagrams will be discussed in Session 4.

## 2.2 Behavioural Constructs

Behaviour, that is, the transformation of input data to outputs or the reaction to events, is modelled by four sub-languages.

**Activities** generalise the concept of classical flow charts by introducing parallel threads, complex interface specifications, and constructs to activate other behaviours during flow chart execution.

Four sub-  
languages  
to specify  
behaviour

**Interactions** are used for specifying messaged-based behaviour of communicating system components.

**State machines** specify reactive behaviour in terms of states and transitions.

---

<sup>1</sup>From the tool perspective, this means that all model semantics can be understood from analysing the model explorer and the textual information associated with each element visible in the explorer. The creation of diagrams is helpful, but not mandatory.

**Use cases** describe behaviour in terms of the high level functionality and uses of a system, that are further specified in the other behavioural diagrams referred to above. They are used to support requirements modelling and the description of system tests, but do not carry sufficient information to develop a system on this basis alone. Typically, use cases are **refined** by activities, interactions, or state machines.

Each of these behavioural sub-languages is associated with its own diagram type: **activity charts**, **sequence charts**, **state charts** (or **state machine diagrams**), and **use case diagrams**. [Diagrams](#)

We will study activities in Session 4 and state machines in Session 3. Interactions will not be covered during this course, since they can always be modelled by concurrent state machines. Use cases will only be touched briefly; there is one exercise concerning use cases in Chapter 7.

## 2.3 Crosscutting Constructs

The term **crosscutting construct** denotes language elements that are neither structural nor behavioural, but may relate to both structural and behavioural elements. The crosscutting constructs of the SysML are

**Allocations** are language elements specifying relationships between other elements. The most important application is to allocate behaviour (e.g. a state machine) on structure (e.g. a block representing a controller). [Allocations, requirements, Profiles, Libraries](#)

**Requirements** are specifications for structural or behavioural model elements to be created later on, or about non-functional system properties to be “implemented” by means of combinations of structural and behavioural model elements.

**Profiles&Model Libraries** help to introduce domain-specific re-usable modelling constructs and model fragments. The former introduces specialised elements from more general SysML constructs. For example, blocks might be specialised with some stereotype, say, «DoorController» to represent door controllers. The latter may contain, for example physical entity specifications with physical units that were modelled in another project and should be reused in the current one.

Requirements concerning allocation, profiles, and libraries are non-functional requirements.



# Chapter 3

## Package Structure for System Models

As mentioned before, models are structured into packages.<sup>1</sup> A suitable model structure which fits to most modelling tasks in the cyber-physical systems domain is as follows.

**requirements** This package contains the requirements, as discussed in Session 1. [Top-level packages](#)

**context** The context package contains information about the system to be developed is embedded into its operational environment. This will be described in more detail in Chapter 5.

**System package** A package carrying the name of the system to be developed (without uppercase letters) contains the proper structural, functional, and non-functional model parts of the target system to be developed.

**systeminterfaces** This package contains information about the data types used to exchange information between the system to be developed and its environment.

---

<sup>1</sup>We will see later, when covering more of the UML/SysML language theory, that models are specialisations of packages. Therefore, it is syntactically correct to place all structural, functional, and non-functional submodels directly underneath the model. This however, is considered as bad style, since it leads to unstructured models that are difficult to maintain.

**projecttypes** Throughout the project, special data types may be used (e.g. enumerations and structured data types modelled by blocks) will be used to specify interfaces and value properties. Since different interfaces and properties may use the same data types, it is useful to put such type declarations into a separate package.

**physicalunits** SysML allows to introduce physical entities (say, electrical current I) in a systematic way and introduce associated concrete physical units (say, mA). These specification should be contained in a separate package. These entities and units can often be imported as an existing model library, it is discussed in Chapter 6 how to do this.

**test** The test package contains test design submodels and system test submodels. Typically, lower-level tests like software integration tests and unit tests are not described by a SysML submodel, since there are testing tools performing these duties in an optimised manner.

**Imported packages** Further model components existing already in reusable model libraries can be imported. At the moment, we see already libraries like `PrimitiveBlocks:ValueTypes` that are automatically imported by the Papyrus tool when creating a new SysML model.

Packages may be structured into sub-packages, if more than just the top-level model-structure shown above is needed. Typically, the system package is structured into sub-packages containing structural, functional, and non-functional information.

Sub-  
packages

# Chapter 4

## Structural Modelling With SysML

### 4.1 Blocks

The central structural modelling element of SysML is the **block** which has been derived from the UML modelling element **class**.<sup>1</sup> Blocks can be used to represent

- structural system components,
- containers of behavioural system components,
- constraints,
- interfaces,
- structured datatypes,

and other structure-related artefacts needed in the modelling process. Just like a class, a block is **a type of** a collection of similar components, data items etc. In the discussion of internal block diagrams below, it will be seen how concrete system components are represented by **parts**, that is, instances of blocks.

Just like a class, a block may be associated with the following information.

---

<sup>1</sup>The Derivation is performed by introduction of the **stereotype** `<<block>>` which will be explained later when exploring the UML/SysML language theory.

- **Attributes** (or, synonymously, **properties**),
- **Operations**,
- **Signal receptions**,

but this basic information structure is detailed further: the attributes are structured into

- **value** properties,
- **part** properties, and
- **reference** properties,
- **ports**, and
- **constraints**.

Value properties are used to model quantifiable characteristics of a block. For example, the current speed of a train on a linear track section may be specified by a value property of type speed and physical unit **km/h**. The Value properties need not be primitive<sup>2</sup>, if they are composed of several data components, the structured type is again modelled by a block. For example, the position of an aircraft flying in 3-dimensional space may be given by  $(x, y, z)$  coordinates in some coordinate system. Position would be modelled by a block with 3 value properties of primitive type **real** and physical unit **m**. Just as in programming languages, value properties may also be typed by enumerations which are already introduced in UML and used in SysML as well.

Value  
properties

Some more details regarding value properties are given in [1, 7.3.4].

Parts describe composition relationships between blocks. A part specifies an instance (or several instances) of a block **in the context of its composite block**. The composite block is the “parent block” where the part is contained in. The context is usually specified by linking the interfaces of the part to interfaces of the composite block or interfaces of other parts contained in the same composite block. These interfaces are in turn modelled by

Part  
properties

---

<sup>2</sup>Recall that a **primitive datatype** cannot be further decomposed, like **int** in programming languages, or  $\mathbb{R}$  in mathematics, or any scalar unit (weight, temperature,...) in physics.

ports to be discussed below. Parts are typically again typed by blocks.<sup>3</sup> If a part occurs with several instances inside a composite block, then its **multiplicity** is greater than 1. Array-like notation in square brackets is used to indicate the number of instances. If the number of parts should be unlimited, the asterisk “\*” is used to indicate this instead of a concrete number. As an alternative to using a multiplicity for specifying a small number of instances in a part, one can specify *several* parts of the same type, each part typed by the same block, but carrying a different part name.

Since value properties can also be typed by blocks, it is necessary to find another criterion to distinguish value properties from part properties. This distinction is made by means of **composite associations**.<sup>4</sup> Typically, these associations are not created in the model explorer, but in block definition diagrams. This is illustrated in the video accompanying Session 2, which explains tool-related aspects for structural modelling in general and, in particular, context modelling. A composite association is an asymmetric relationship: it has a “part end” pointing to the part which is inside the composite block. The composite block is referenced by the other end of the association. The part end of a composite association also specifies a multiplicity and a role played by the part in the context of the parent block. In Fig. 5.1, several context associations are shown: they end with a black diamond symbol at the composite end, and with an arrow at the part end.

Distinguish parts from values

The composite association identifies a specific **destruction semantics**: the part cannot “live” without its composite, that is, without its parent block.

**Example 1.** In the C++ programming language, this is nicely reflected by defining attributes of class type like

```
1 class C1 {
2 private:
3   C2 part1;
4 public:
5   ...
6 };
```

The attribute **part1** is automatically created (using the default constructor of **C2**) when an instance of **C1** is created. Moreover, **part1** cannot live without the declaring class **C1**: it is automatically destroyed when the **C1**-instance is destroyed. □

---

<sup>3</sup>In use case diagrams, they may be typed by actors.

<sup>4</sup>The term ‘association’ is synonymous to the term **relationship** which you may know from entity-relationship modelling in the context of databases.

Some more details regarding part properties are given in [1, 7.3.1]. We will discuss the introduction of parts in more detail in Chapter 5, when introducing the context submodel.

Reference properties have another destruction semantics: parent blocks refer to other blocks, but the latter are *not* destroyed with the former.

Reference  
properties

**Example 2.** In the C++ programming language, this is reflected by defining attributes that are pointers to instances of other class types, like

```
1 class C1 {
2 private:
3     C2* ptr1;
4 public:
5     ...
6 };
```

The pointer `ptr1` may be defined, for example, in the constructor of `C1`, when a pointer to a `C2`-instance is passed to the `C1`-constructor. The object pointed to by `ptr1` lives on until a

```
1 delete ptr1;
```

command is executed. If the `C1`-instance is deleted, this does not affect the `C2`-instance.  $\square$

To distinguish reference properties from part properties, the **reference association** is used in analogy to the composite association. Again, reference associations are typically introduced when drawing block definition diagrams.

More details regarding part properties are given in [1, 7.3.2]. We will discuss the introduction of parts in more detail in Chapter 5, when introducing the context submodel.

Typically, blocks only contain operations and/or signal receptions, if they are associated with behaviour. We will therefore skip the specification of operations and signal receptions and handle them in the sessions on behavioural modelling.

Operations  
and signal  
receptions  
are  
associated  
with  
behaviour

## 4.2 Ports and Flows

**Ports** are language elements for specifying interfaces (hardware or software). SysML distinguishes three variants of ports.

A **full port** is a “real” part of the boundary of a block. Typically, hardware interfaces are modelled as full ports.

A **proxy port** is complementary to a full port: it is not a part of its parent block, but provides a “virtual” external access point<sup>5</sup> to and from the features of the parent block or its parts.

A **port** is used whenever the classification as ‘full’ or ‘proxy’ does not fit.

Proxy ports are typed by **interface blocks** which cannot have internal parts or behaviour. A typical application for proxy ports is at a “black box” system boundary<sup>6</sup>.

**Example 3.** When modelling the context of our turn indication controller (see Chapter 5), the controller is represented as a black box, where its internal structure is invisible to the operational environment. This black box is purely virtual, since the “real” turn indication controller consists of 4 controllers, each servicing a subset of interfaces to the operational environments. Therefore, proxy ports are used to model the interfaces of the virtual black box, and full ports are used to model the hardware interfaces of 4 controllers.

□

Ports are connected to other ports by means of **connectors**, representing the flow of data. One port may be connected to more than one other port; this applies to situations where data flows from one source to several sinks, or where one sink aggregates data from several sources. A more specific version of a connector is an **item flow** which, in particular, associates a flow direction for the data to be passed between ports.

In this course, we always use item flows to connect ports, and we always pass exactly the data items across the flow that are specified by the port. Therefore, item flows need no further specification in our context. Read [1, pp. 142] for understanding the more powerful features of item flows which are not used in our course.

Please read [1, 7.6] for more details about ports.

## 4.3 Block Definition Diagrams

The BDDs are used to visualise decomposition and reference relations. Typically, they contain a tree-like structure with the block to be decomposed as

---

<sup>5</sup>similar to a pass through or relay

<sup>6</sup>If the type is atomic, it is not necessary to create an interface block with just one component; the atomic type may be used directly as the type of the proxy port.

root. Blocks that are part of the root or subordinate parts of parts closer to the root are displayed underneath their parents. The links in the composite-part decomposition tree are realised by composite associations. Part names are specified by the role names at the part-end. The number of object instances associated with a part is specified by the multiplicity at the part-end.

A typical BDD is shown in Fig. 5.1 and further discussed in Chapter 5. Please read [1, 7.2] for more details about BDDs.

When visualising the compartments of a block on a BDD, tools offer to display the block properties and operations in different **compartments** structuring the various types of attributes a block may have. The SysML standard [2] defines many compartment types, but is slightly vague about whether these are mandatory or whether tools may introduce their own compartments. Therefore, Papyrus allows for manual association of properties to compartments. The video shows how to do this. Like diagrams, compartments do not add semantics to the operations and properties of a block: the semantics is completely fixed by parameter types and associations.

Compartments  
of a block

## 4.4 Internal Block Diagrams

While BDDs are used to display the hierarchy of blocks and their parts, IBDs are used to show how parts of a block interact with each other. We can say that an IBD associated with a given block is an **exploded view** showing its internal structure up to a certain point. Please read [1, 7.3] for a detailed explanation of the various possibilities to display parts and various types of associations on an IBD. Throughout this course, we will use the following reduced set of elements on IBDs:

- Parts
- Ports
- Flows

Moreover, we use the convention that flows may only connect to ports, but not directly to parts.

An (incomplete) IBD is contained in the sample model which is provided for the this session.



# Chapter 5

## The Context Submodel

### 5.1 Context Model Ingredients

The **context submodel** specifies the interfaces between the system to be modelled and its operational environment. Moreover, it is usually helpful to impose some top-down structure on the components of the operational environment, if there are many of them. With these objectives in mind, the context should be modelled by means of blocks representing

Blocks in  
the  
context  
model

- the system to be developed,
- each component in the operational environment which has an interface to our target system, and
- some auxiliary blocks creating a sub-structure for the operation environment, if desirable.

Interfaces are modelled by means of ports and item flows connecting ports.

Ports and  
Flows  
BDDs and  
IBDs

The top-down structure of the operational environment, together with the target system are visualised in a BDD, and the detailed interface structure with connecting flows is depicted in one or more IBDs.

### 5.2 Context BDD

The context model is associated with one or more BDDs showing the target system and the components in the operational environment in a top-down

tree view. An example is shown in Fig. 5.1, where the context BDD of the turn indication controller is specified.

Since the context is a conceptual entity and not something to be actually built, the reference association introduced in Chapter 4 is used to refer to the environment elements and to the target system: a battery, for example, exists independently of the notion of the turn indication controller context. After these associations have been established, the components become reference properties of the context block.

In Fig. 5.1, additional top-down structure has been introduced by using the blocks `Dashboard` and `TurnIndicationLights` as reference properties of the context. Since the dashboard LEDs cannot exist without a dashboard (the `Dashboard` is a real physical entity), the composite association is used here. The `TurnIndicationLights`, however, are a conceptual entity. Therefore, the reference association is used again.

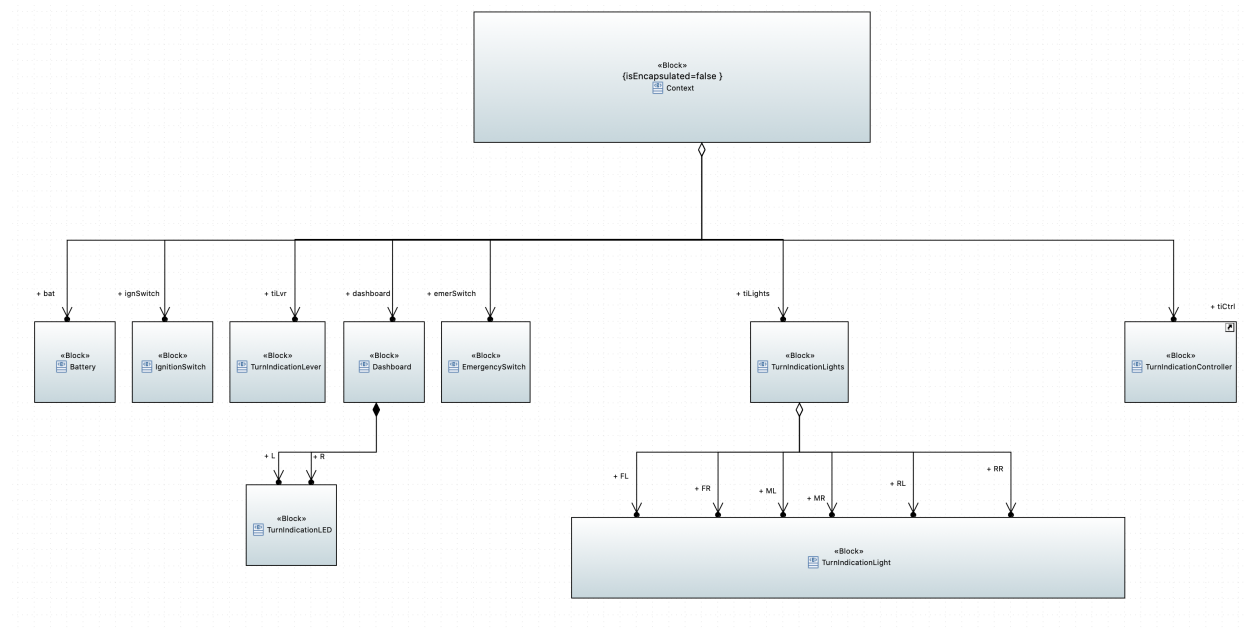


Figure 5.1: Context BDD for the turn indication function.

The association names at the part/reference property ends define the names of the corresponding instances. For example, there is one instance of `Battery` in the context, and the instance is called `bat`. There are 6 instances

of block TurnIndicationLight, denoted by FL (“forward left”), . . . , RR (“rear right”).

### 5.3 Context IBDs

In Fig. 5.2, the top-level context IBD is shown. It is still incomplete, its completion should be performed by the readers (Exercise 7.2.2). Note that the IBD has an outer frame like a block, labelled by `context`. This is to indicate that this IBD is the explosive view of the internal structure of the context. From Fig. 5.1 we know that the sub-components of `Context` are all reference properties. This is reflected in the IBD by letting these reference properties occur as dashed-line boxes. In contrast to this, the subordinate IBD presenting the dashboard explosive view (Fig. 5.3) has the LEDs integrated as parts via composite association. Therefore, the LED parts are drawn as solid-line boxes.

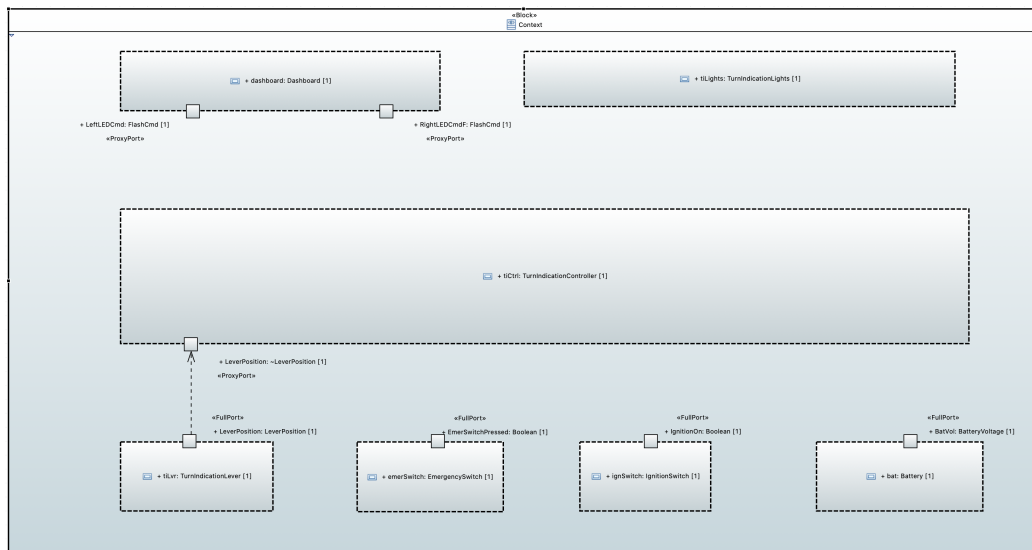


Figure 5.2: Top-level context IBD (incomplete) for the turn indication function.

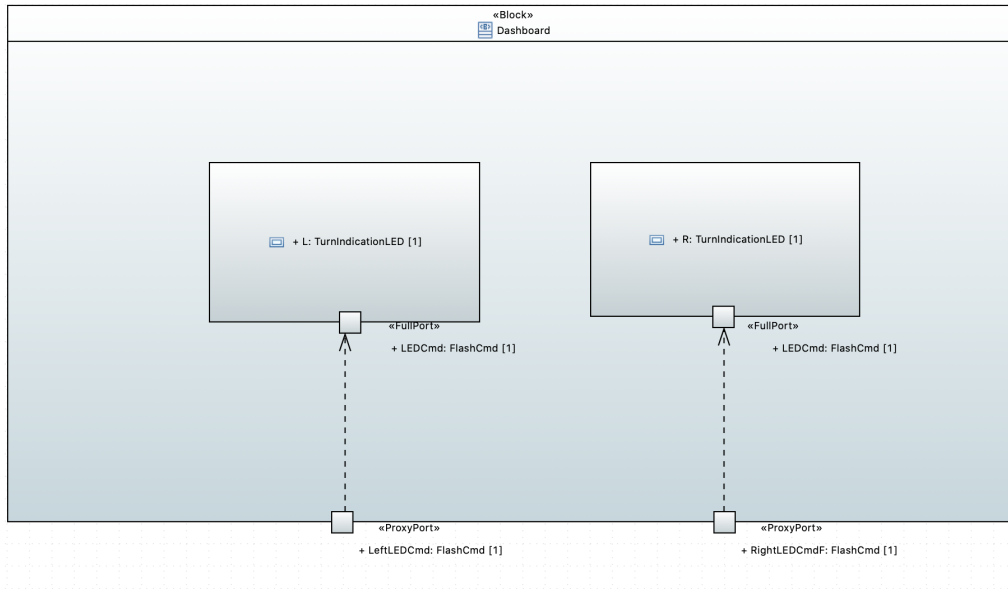


Figure 5.3: Dashboard IBD.

All interfaces of the `tiCtrl` instance are proxy ports, since they do not exist in HW, but are just used as intermediate points to connect flows from/to the outside world with “real” ports inside `tiCtrl`. For the dashboard interfaces, also proxy ports have been chosen, though the dashboard exists as a real entity. The reason is, that these interfaces just have a relay function. The full ports are connected to the `TurnIndicationLED` instances L and R.

A further sub-ordinate IBD is used to represent the 6 turn indication lights with their ports.

# Chapter 6

## Introducing Enumerations, Physical Entities and Units

It is very important to be able to introduce project-specific datatypes, because this facilitates the proper use of data. In SysML, project-specific types are created along the following lines.

1. Atomic project-specific types can be created as enumerations.
2. Further atomic project-specific types can be created as **ValueTypes** that are derivations from the primitive value types
  - Boolean,
  - Complex,
  - Integer,
  - Number,
  - Real,
  - String

provided by the SysML.

3. Composite data types (e.g. vectors of components with primitive types) can be specified as blocks whose components have known types.
4. Physical units can be specified by creating instances of **Unit**<sup>1</sup>.

---

<sup>1</sup>These are just blocks labelled with the stereotype «Unit ».

5. Physical entities (Speed, temperature, voltage, ...) are introduced by creating instances of `QuantityKind`.
6. Primitive or composite types can now be typed by means of a value type which has a certain value range and a physical unit.

A video shows the details how to create new physical units, quantity kinds, and value types.

The resulting project-specific types for the turn indication controller are shown in Fig. 6.1.

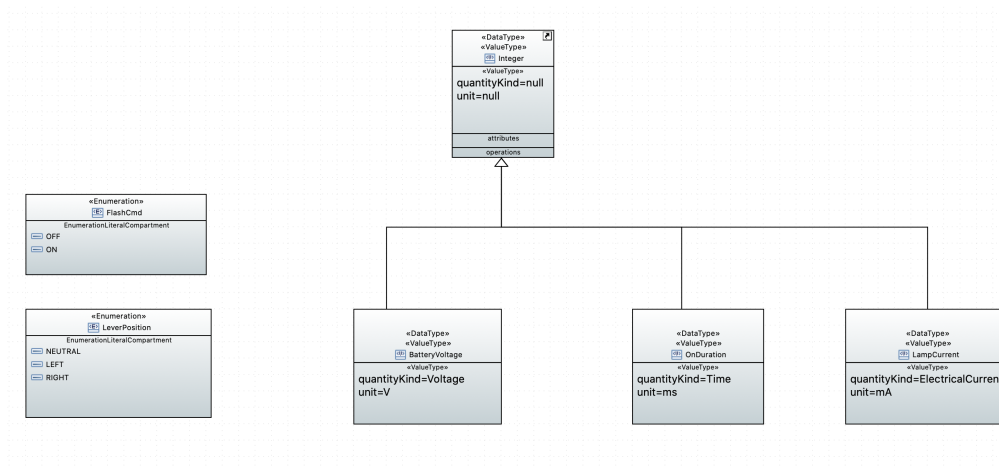


Figure 6.1: Project types for the turn indication controller.

# Chapter 7

## Questions and Exercises

### 7.1 Questions

#### 7.1.1 Associations Modelling Java Attributes

In Example 1 and Example 2, we have explained how either composite associations or reference associations should be used to model the integration of objects as class attributes in C++. Now consider a Java program which is analogous to the C++ program shown in Example 1.

```
1 public class C1 {  
2     ...  
3     public C2 part1;  
4     ...  
5 }
```

When modelling C1 and C2 in a BDD, would you use a composite association or a reference association? Explain your decision.

### 7.2 Exercises

#### 7.2.1 Use Cases for Requirements

Study Chapter 12, Sections 12.1 — 12.4 in [1] about use cases. Extend your requirements package by use cases illustrating the following requirements.

1. Turn indication flashing on left-hand side and right-hand side, and the effect of the ignition switch.

2. Emergency flashing, overridden by turn indication flashing.

Insert the use cases and associated use case diagrams as “illustrations” underneath the related requirement.

## **7.2.2 Interfaces and Internal Block Diagrams for the System Context**

In the context IBD which already exists in the sample model provided for Session 2, specify the missing ports for all system interfaces, together with suitable data types and associate them with their blocks in the operational environment and the turn indication controller itself.

In the existing IBD, the `dashboard` and the `tiLights` appear as black boxes, containing LEDs and lamps, respectively. Create the missing IBDs for `dashboard` and `tiLights`, together with the required ports and connectors.

## **7.2.3 Block Definition Diagram for the Turn Indication Controller**

Create a BDD showing the internal top-down decomposition of the turn indication controller, down to the level of the four controllers that together implement the turn indication controller hardware.

## **7.2.4 Internal Interfaces and Internal Block Diagram for the Turn Indication Controller**

Create the necessary ports and flows so that the interfaces between the four controllers of the turn indication function, as well as their connections to the proxy ports can be modelled. Create an IBD where the four controllers, their ports, and their flows are shown.



# Bibliography

- [1] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML, Third Edition: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2014.
- [2] Object Management Group. *OMG Systems Modeling Language (OMG SysML), Version 1.6*. Technical report, Object Management Group, 2019. <http://www.omg.org/spec/SysML/1.4>.