EEBus Initiative e.V.
Butzweilerhof Allee 4
50829 Cologne
GERMANY

Rue d'Arlon 25
1050 Brussels
BELGIUM

Phone: +49 221 / 47 44 12 - 20
Fax: +49 221 / 47 44 12 - 1822

info@eebus.org
www.eebus.org

District court: Cologne
VR 17275

EEBus SPINE Technical Report

Introduction

Version 1.1.1

Cologne, 2018-12-17

The EEBus concept was developed as part of E-Energy.

Supported by: Federal Ministry of Economics and Technology on the basis of a decision by the German Bundestag

Table of contents

List of figures

46
47

1 Overview

1.1 What is SPINE

SPINE (**S**mart **P**remises **I**nteroperable **N**eutral-message **E**xchange) defines a neutral layer which helps connecting different communications technologies to build a smart home / smart grid system. As SPINE only defines messages and procedures on application level (ISO-OSI layer 7), it is completely independent from the used transport protocol. Any technology that supports the bi-directional exchange of arbitrary data can be used more or less directly (e.g. SHIP [SmartHome IP] – that is also created by the EEBus Initiative e.V. – or Thread). For each communications technology a mapping is needed, but for some this mapping is very simple (this is esp. the case for SHIP). Within this mapping, the possible data points are mapped as well as some protocol definitions. The capabilities of the existing technologies are very diverse, so each SPINE-to-technology mapping is different and (possibly) not all SPINE functions are supported. However, SPINE follows a modular concept that helps incorporating other technologies as much as possible. This finally leads to the possibility to create SPINE-based applications on one hand and then connect them to one or more technology mapping components on the other hand.

SPINE covers use cases concerning control and monitoring of smart appliances like White Goods, HVAC systems and devices like batteries, electrical vehicles, etc. with a focus on the sectors of smart energy, smart home & building, connected devices and E-Mobility. Domains like AAL (ambient assisted living), home entertainment, etc. are currently not in the focus, although some use cases from these domains can be realised with the existing data model as well. As the SPINE data model evolves continuously, further domains will be added.

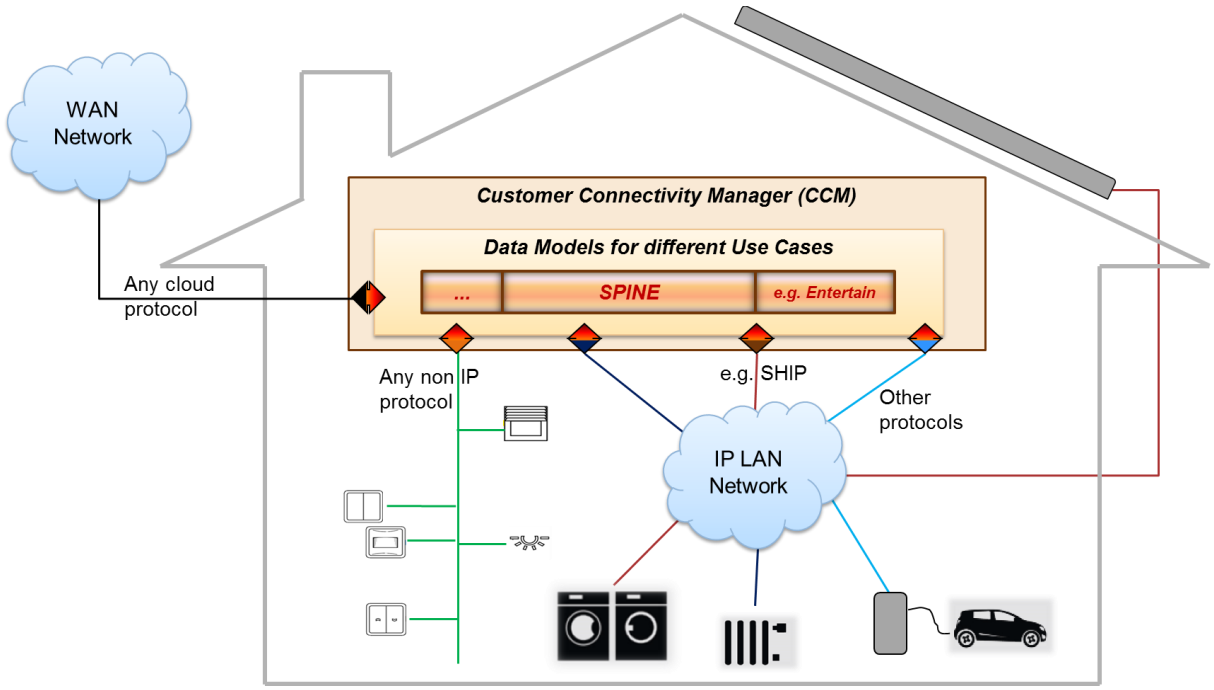

Figure 1: SPINE embedment within Smart Home environment

1.2 How to start

This document will give you an overview of the concepts behind SPINE. They are not fully described here, but references to certain chapters in other SPINE documents are given. It is recommended to proceed in the order denoted in the chapters of this introductive documentation.

XML Schema Definition (XSD) files with the file extension ".xsd" are used to specify the SPINE data model. Many parts of the SPINE documentation contain graphical representations of XSDs. The section "Graphical representation" in the document [ResourceSpecification] explains how to interpret these graphics.

The application layer interface of SPINE is modelled as message-based protocol. A SPINE message is separated into header and payload as can be seen in Figure 2. The protocol related aspects of SPINE (including structure of header and payload of messages) are described in [ProtocolSpecification], together with some parts that occur in the payload. Most part of the payload (including the complete SPINE resource model) is specified within [ResourceSpecification].

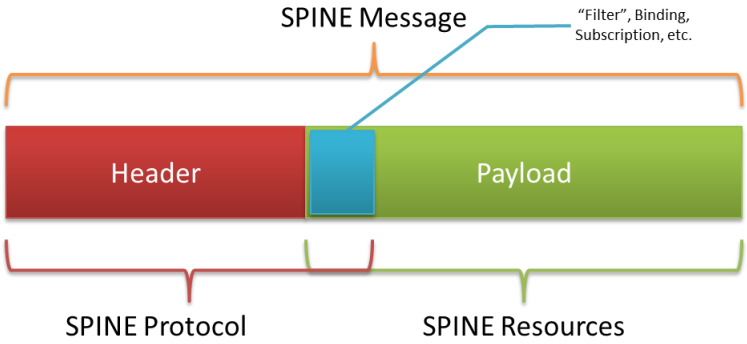

Figure 2: SPINE Message Structure

The following graphic shows different documentation categories in more detail. The "lowest" part (as in ISO-OSI model) is the SHIP (Smart Home IP) specification [SHIPSpecification]. It is one possible way to transport SPINE messages between devices, but it is not the only one. The next "level" is described in the [ProtocolSpecification] (including functional commissioning concepts like binding, subscription and detailed discovery). The resources are described in [ResourceSpecification]. These are defined based upon requirements from use cases and user stories.

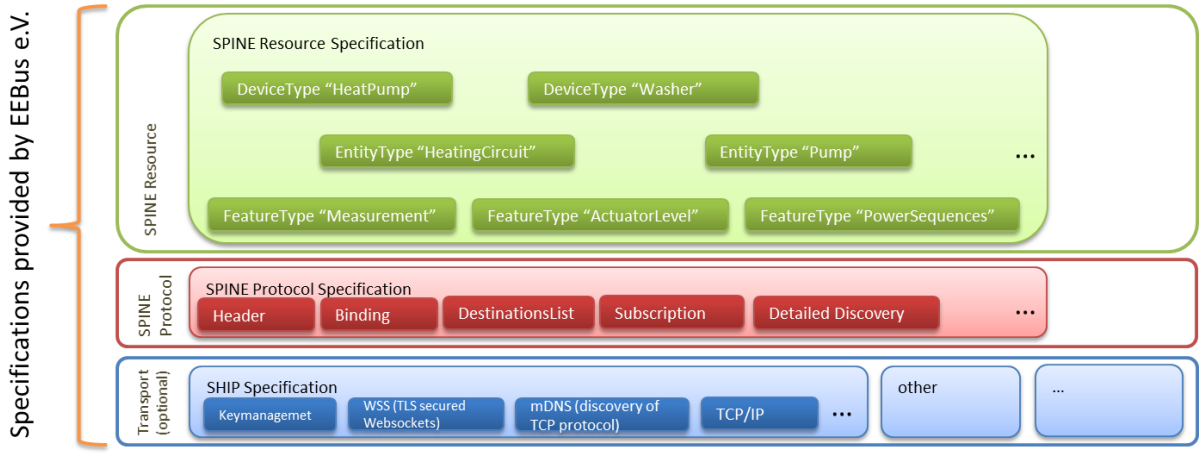

Figure 3: EEBus Specifications in detail (1)

Within the EEBus working-groups, Use Cases are also developed with a modular approach: High-Level Use Cases define a particular functionality in detail (containing a description, a list of scenarios, diagrams like sequence diagrams and example XMLs). They are created with the help of the Generic Use Case Toolbox that provides reusable blocks of messages. However, all information necessary for implementing and testing is given within the High-Level Use Cases. The Generic Use Case Toolbox is only used to ease the initial creation of the High-Level Use Cases by reusing existing functionalities.

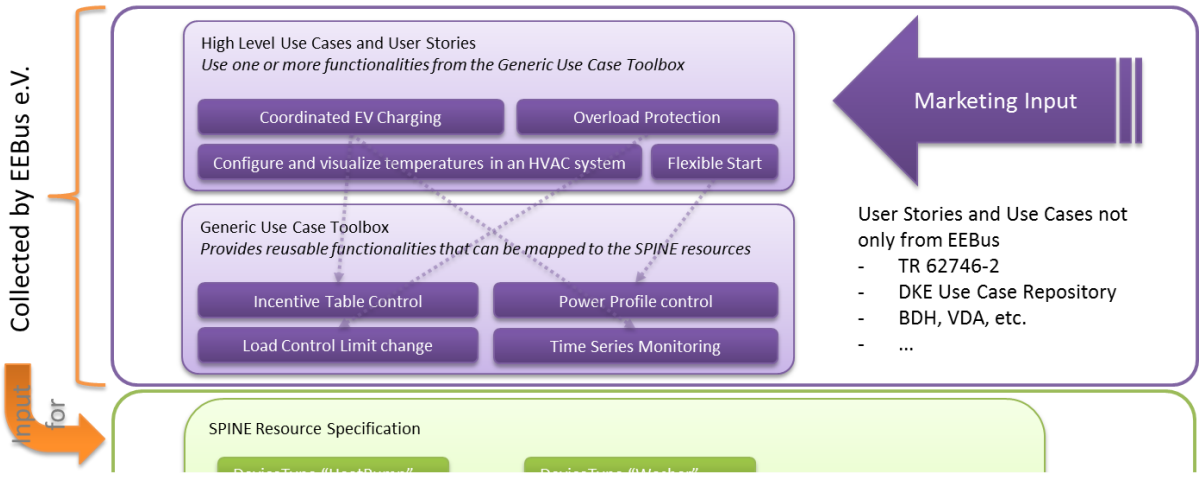

Figure 4: EEBus Specifications in detail (2)

SPINE is defined in a very modular way, including metadata (e.g. the kind of device, the constraints of the functionality, etc.). This enables one (e.g. an energy manager) to even communicate with devices that were unknown during development and to use them in a meaningful way (an important advantage, especially for energy management).

The modularity of SPINE leads to a more extensive specification than just modelling each and every information as a single data point. But considering the fast-evolving world of the Internet of Things, a specification that is easy to extend and that can be adapted to new market needs without breaking interoperability and compatibility is a great benefit.

Some wordings may be used before they are explained in detail. Due to the complexity of the concepts this cannot be prevented.

If you prefer reading documents from the beginning all the way to the end, the following order is recommended:

- This document
- [ProtocolSpecification]
- [ResourceSpecification]

Please note that the sequence in this document with the references to specific chapters of the documentations helps to get a better and quicker understanding of SPINE than just reading the documents from the beginning.

1.3 References

1.3.1 EEBus SPINE documents

[ProtocolSpecification] EEBus_SPINE_TS_ProtocolSpecification.pdf

[ResourceSpecification] EEBus_SPINE_TS_ResourceSpecification.pdf

[DataModelXSDs] EEBus_SPINE_TS_ActuatorLevel.xsd, ..., EEBus_SPINE_TS_Version.xsd

1.3.2 Other EEBus documents

[SHIPSpecification] SHIP_Specification_V1.0.0.pdf

[SPINEUseCaseXSDs] XSDs delivered within the EEBus Use Case Package
EEBus_UC_TS_UseCaseTypes.xsd,
EEBus_UC_TS_UseCaseWrapper.xsd

1.3.3 Websites

[EEBus]	http://www.eebus.org	*Official EEBus Initiative e.V. website.*
[W3C]	http://www.w3.org/	*Official World Wide Web Consortium website.*
[W3Schools]	http://www.w3schools.com/	*Tutorials and examples for XML and others.*

2 First steps

2.1 General concepts

As stated above, SPINE itself does not define any layers below ISO-OSI layer 7. So, the SPINE data model and protocol specifications can be used with many technologies that support bi-directional communication with any payload.

The SPINE device model helps finding appropriate communication partners (see section 2.2). It consists of three layers: (physical) device, logical (sub-)device (entity), functionality (feature).

The SPINE communication uses its own header and payload definitions within a SPINE message (hence within layer 7, see section 3.1).

Messages can be transferred with different classifiers (e.g. read/reply/write, see section 3.2), enabling polling-based communication as well as trigger or event based notifications.

The data model is very modular. For different use cases, the SPINE functions can be combined and used in different ways. This may seem more complex than just modelling each needed data as a separate element like it is common practice in most technologies. But in fact, the modularity enhances reusability and recognition of well-known definitions. Especially as the list of requirements grows, and so the list of different data points, the modular concept of SPINE shows its strength.

In some cases, the combination of several (modular) SPINE functions into a single atomic message simplifies the implementation of complex functionalities. For this purpose, SPINE complex classes can be defined that reuse the modular SPINE functions (see section 3.3).

Each feature has a role: *client* or *server* (and, in some cases, *special*) (see section 3.4). The server provides information that may be read or (if allowed) written. The client requests information from a server or changes it.

Whether a message could be transmitted / evaluated or not can be communicated with acknowledgement and error messages (see section 3.5).

Binding and subscription mechanisms enable easy connecting of matching functionality (see section 3.6) for specific communication purposes. The usage of binding is feature type specific (it may be used to determine which communication partner is permitted to change something on the feature or where to send commands to). Subscription is used wherever a client wants to receive updates from some server functionality automatically.

To be able to communicate with another device, one needs to know what functionality is implemented there. The device discovery concept (see section 3.7) delivers the relevant information about a device (including hierarchy levels *device*, *entity* and *feature*).

For every communication, it is essential to have an address where the message shall be sent to. SPINE has its own addressing schema (see section 3.8) for all three hierarchy levels.

To communicate with devices that are just accessible via another device, the destination list of the intermediate device may be requested. Devices included in this list may be used as partner in the further communication (see section 3.9).

In some cases, not the complete data of a resource is of interest. A filter may then be applied together with a read command, to cut down the payload to transmit (see section 3.10). This concept is also used when a notification only includes changed data or together with a write command to change just parts of the destination data container.

Further introductory sections with some more general information can be found in the following documents and sections:

- [ProtocolSpecification], "Introduction", "General notations", "Architecture requirements" and "Compatibility considerations"
- [ResourceSpecification], "Introduction"

2.2 SPINE device model (devices / entities / features)

A device model helps to find appropriate devices for communication as well as clustering information in a logical manner. The SPINE device model consists of three levels: device, entity and feature. Each level holds a type: *deviceType*, *entityType* and *featureType*.

- The device type denotes which physical device is described.
- The entity type states the logical device. A physical device may consist of more than one logical device.
- The feature type describes rules for exactly one class (simple or complex, see section 3.3). In the device discovery, the feature type is stated together with a list of supported functions and the possible operations on each function.

The address levels of the described device-model parts are as depicted in Figure 5. Please refer to section 3.8 for further information.

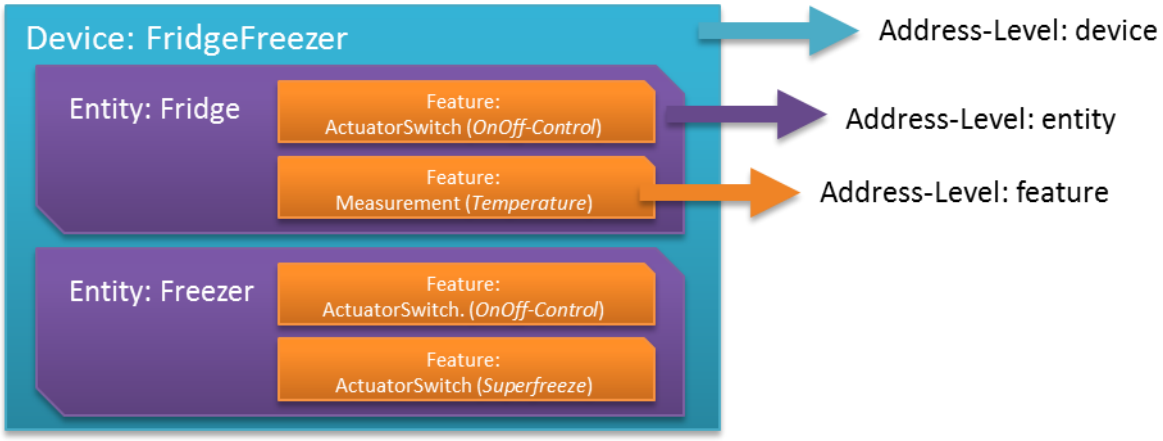

Figure 5: SPINE address levels

Please read the following chapter for details:

- [ResourceSpecification], chapter "Overall model hierarchy concept overview"

2.3 SPINE classes, functions and elements

The SPINE data model is divided into classes. For each functional domain supported by SPINE, a related *standard class* is defined. For example, *Measurement*, *Sensing*, *TimeInformation*, etc. Class names are capitalised. Additionally, *complex classes* exist that allow the combination of functionality defined by *standard classes*.

Detailed descriptions on the SPINE classes can be found in [ResourceSpecification], sections "Class descriptions" -> "Complex Classes" (with one sub-section for every complex class) and "Class descriptions" -> "Standard Classes" (with one sub-section for every standard class, e.g. "ActuatorLevel", "Measurement", etc.). It is recommended to read the class description of only one class now (e.g. "Class descriptions" -> "Standard Classes" -> "ActuatorLevel") and the other ones only if needed. Afterwards, a quick view on the chapter "Common technical details" describes some general concepts and the common used data types. No detailed knowledge is needed at this point of reading.

SPINE classes have a collection of functions that can be used for communication. Function names are lowercased. Example from the *Measurement* class:

- measurementListData
- measurementConstraintsListData
- measurementDescriptionListData
- measurementThresholdRelationListData

There is always either one *...Data* function or one *...ListData* function (also called the "resource"). In case of a *...ListData* function, a *...ListDataSelectors* is defined, used for filtering desired information.

As an example, in the ActuatorLevel class the *actuatorLevel* function and the *actuatorLevelDescription* function are the only defined information. In contrast, the information specified within the Measurement class covers the above-mentioned functions together with the selectors (as all functions are of type *...ListData*):

- measurementListDataSelectors
- measurementConstraintsListDataSelectors
- measurementDescriptionListDataSelectors
- measurementThresholdRelationListDataSelectors

In the second example, the *...ListData* function is denoted, not the *...Data* function, because the *Measurement* class supports lists and does not allow to send the single *...Data* function as so-called “cmd” payload (but the *...Data* function is a sub-element of *...ListData*).

This example also shows that not every root element that exists in the XSDs may be used for the basic “cmd” payload communication. The [ResourceSpecification] explains this in more detail: For each class there is a dedicated section ("ActuatorLevel" to "Version") with its possible functions.

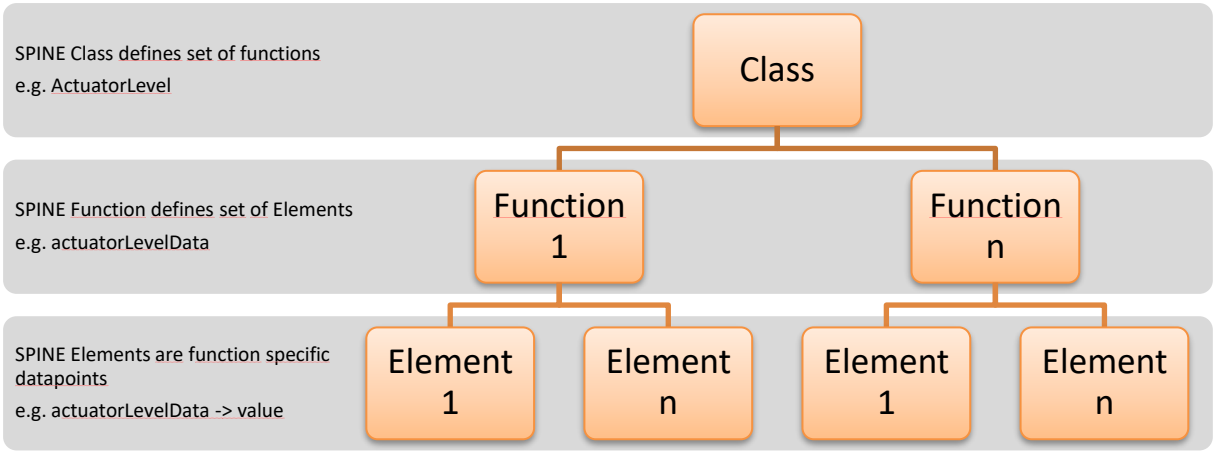

Figure 6: SPINE class hierarchy

Functions contain elements. Each element represents a single information. E.g. a timestamp, a value, a unit, etc. In the data model, elements are initially marked as optional. This is due to reuse-considerations, particularly for the complex class concept (see section 3.3). Of course, some values need to be mandatory, to fulfil a meaningful and interoperable context. Such restrictions are entailed with the use of the so-called feature types (see section 2.2).

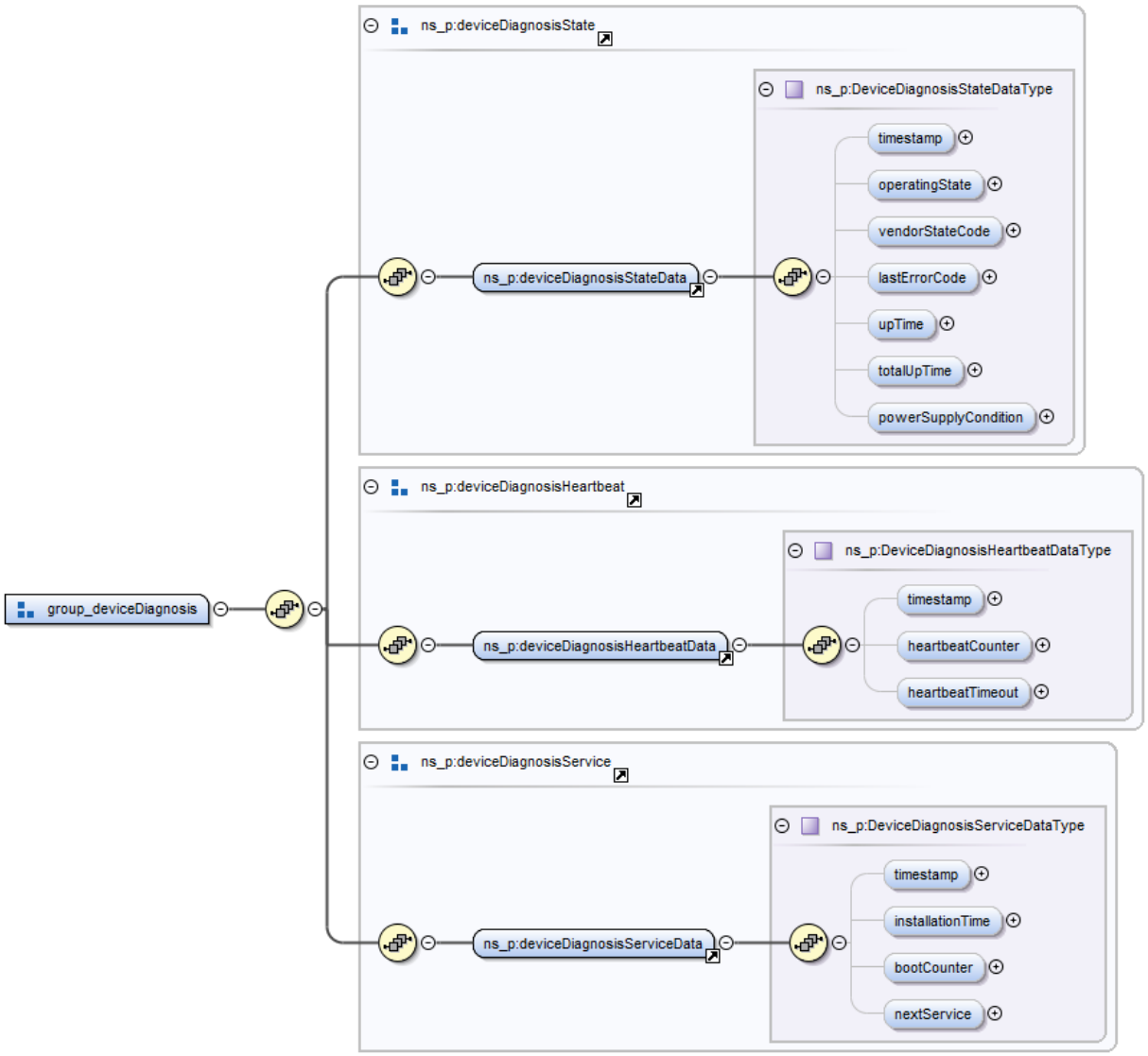

Figure 7: SPINE class example (DeviceDiagnosis)

The XSDs and some further documents describe how an XML must be structured and filled in order to be “valid” within the SPINE specification. That does not mean that only XML can be used for communication. In [SHIP], e.g., JSON is used. Dependent on the communications technology, respective formats (e.g. ASN1-PER) may be used.

Figure 8: XSD to XML example (DeviceDiagnosis)

The next sections explain some more details. Please keep in mind, that this introductive document does not cover all SPINE concepts, but gives an overview. To get every detail on SPINE, please read the related documents and sections carefully.

2.4 Link between device and function vs. datagram and function

SPINE functions are, as already denoted earlier, the central component in all SPINE concepts. In the resource hierarchy, they are defined as the data containing part behind the lowest level ("FeatureType"). Figure 9 explains in a graphical way where the SPINE functions are "located" in the concepts. On the left side, the graphic starts with the entry-point to the typical SPINE message (element tag "datagram"). On the right side, the resource model is depicted (Device, Entity, Feature), together with the functions and the elements. In which documents the different parts are defined, is visualized, too.

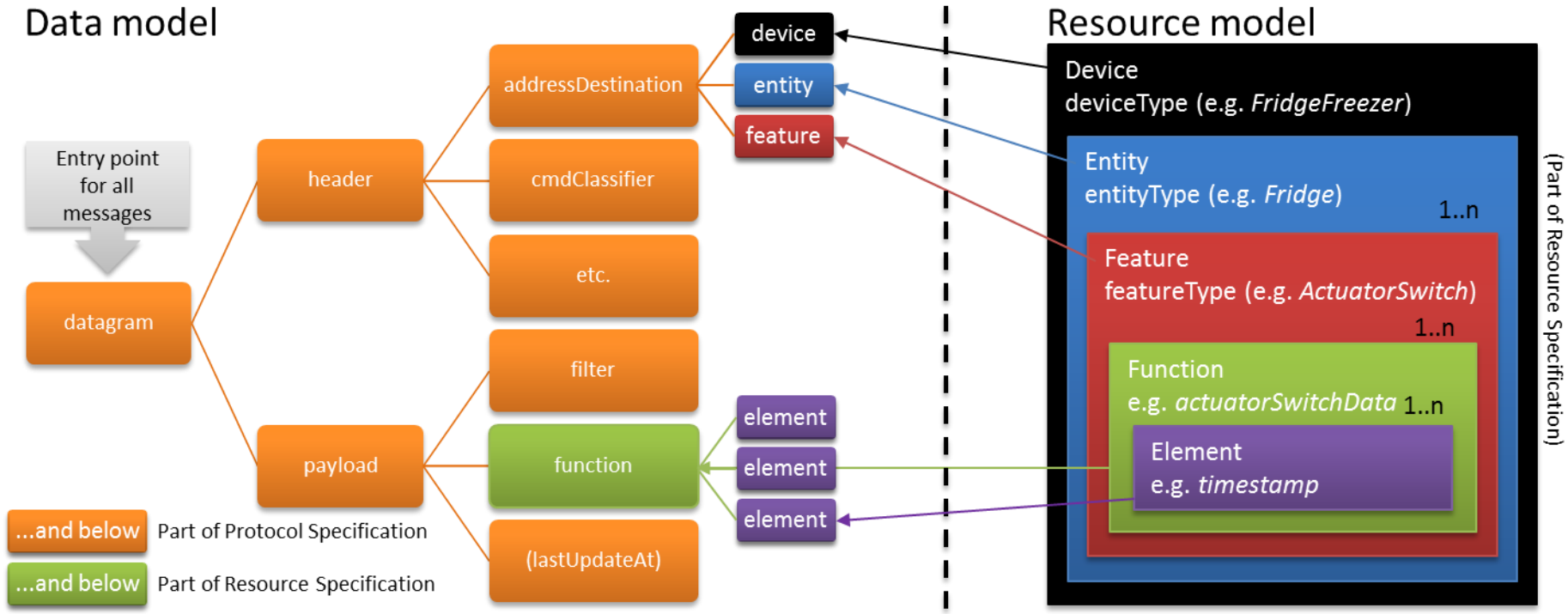

Figure 9: Link between datagram and function vs. device and function

Please note that the concept of sub-entities is not visualized in Figure 9. For details see section 3.8.

3 Detailed information

3.1 Details on header and payload

A SPINE datagram consists of the header and the payload and represents a SPINE message.

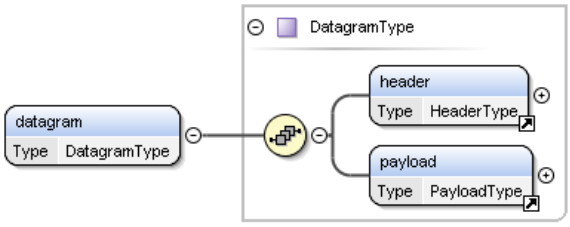

Figure 10: Datagram structure

The SPINE header is the first part of a SPINE message. It contains elements for source (sender), destination (recipient), classifier, etc.

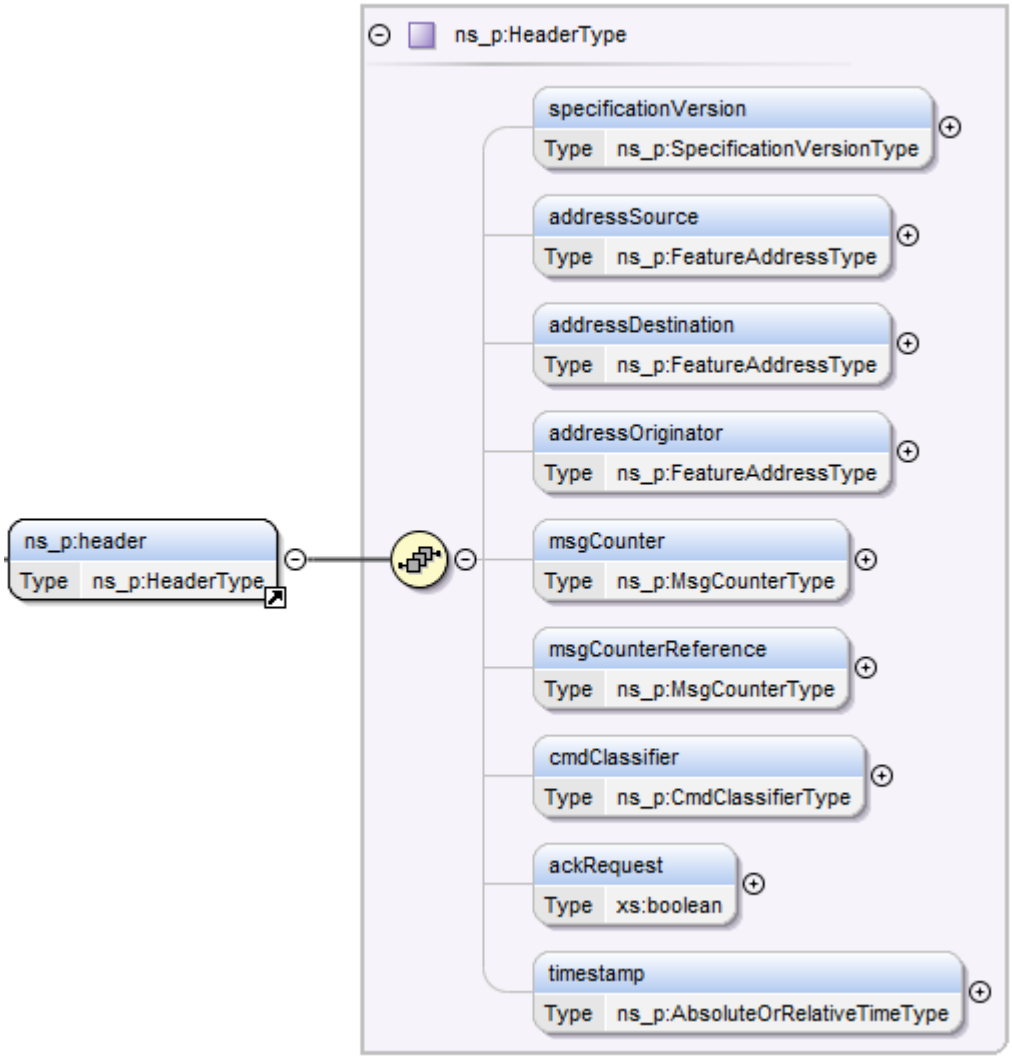

Figure 11: Header structure

The payload contains the actual data. As it needs to embrace quite different data it can be considered as some kind of “container”. This makes the formal definition more complex, nevertheless its constituents can be grouped as Figure 12 briefly shows.

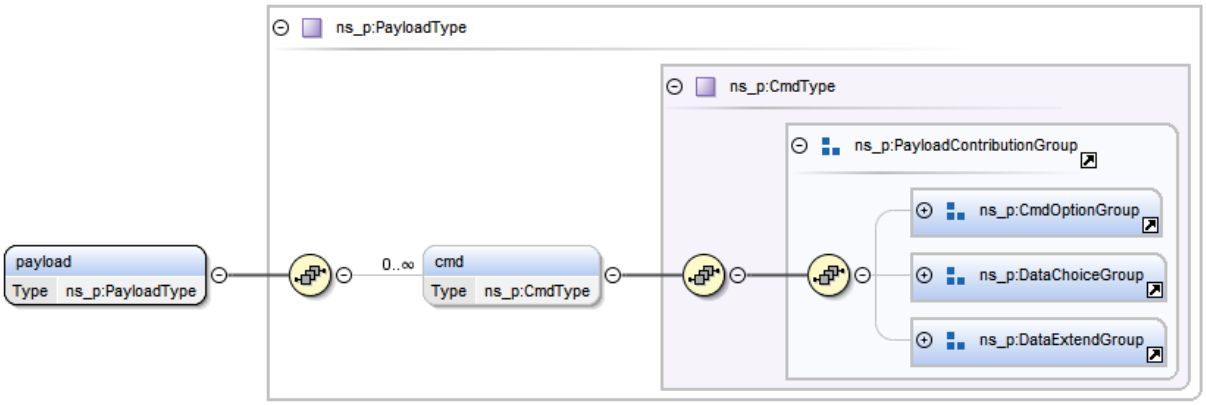

Figure 12: Payload structure

Please read the following sections for details:

- [ProtocolSpecification], chapter "SPINE Datagram" and sub-sections "Introduction", "Header" and "Payload"

3.2 Message classifiers

To determine whether a function is used for a read/reply/notify/write/call/result/ack operation, the classifier is used.

To request some information, the **read** classifier is used (together with a "filter" (see section 3.10) if needed). If the request could be processed, a proper **reply** is sent back.

If a client wants to change some data on a server (= data owner, see section 3.4) he sends a **write** command to the server. Subscribed clients (see section 3.6) are informed about changes on data of the server with a **notify** command.

To initiate some remote procedure on a communications partner, a **call** command is sent.

The **result** classifier is used within acknowledge and error messages (see section 3.5).

Please read the following sections for details:

- [ProtocolSpecification], section "SPINE Datagram" -> "Header" -> "Message classifiers"

3.3 Standard class vs. complex class

The normal SPINE functions are very modular and rather “self-sufficient”. Some scenarios are a bit more complex and require a relation between single functions or information. This can be achieved with references from the different functions (e.g. with the *measurementId*; some examples are visualized here (see [ResourceSpecification], section "Common technical details" -> "Relations between thematically connected SPINE classes").

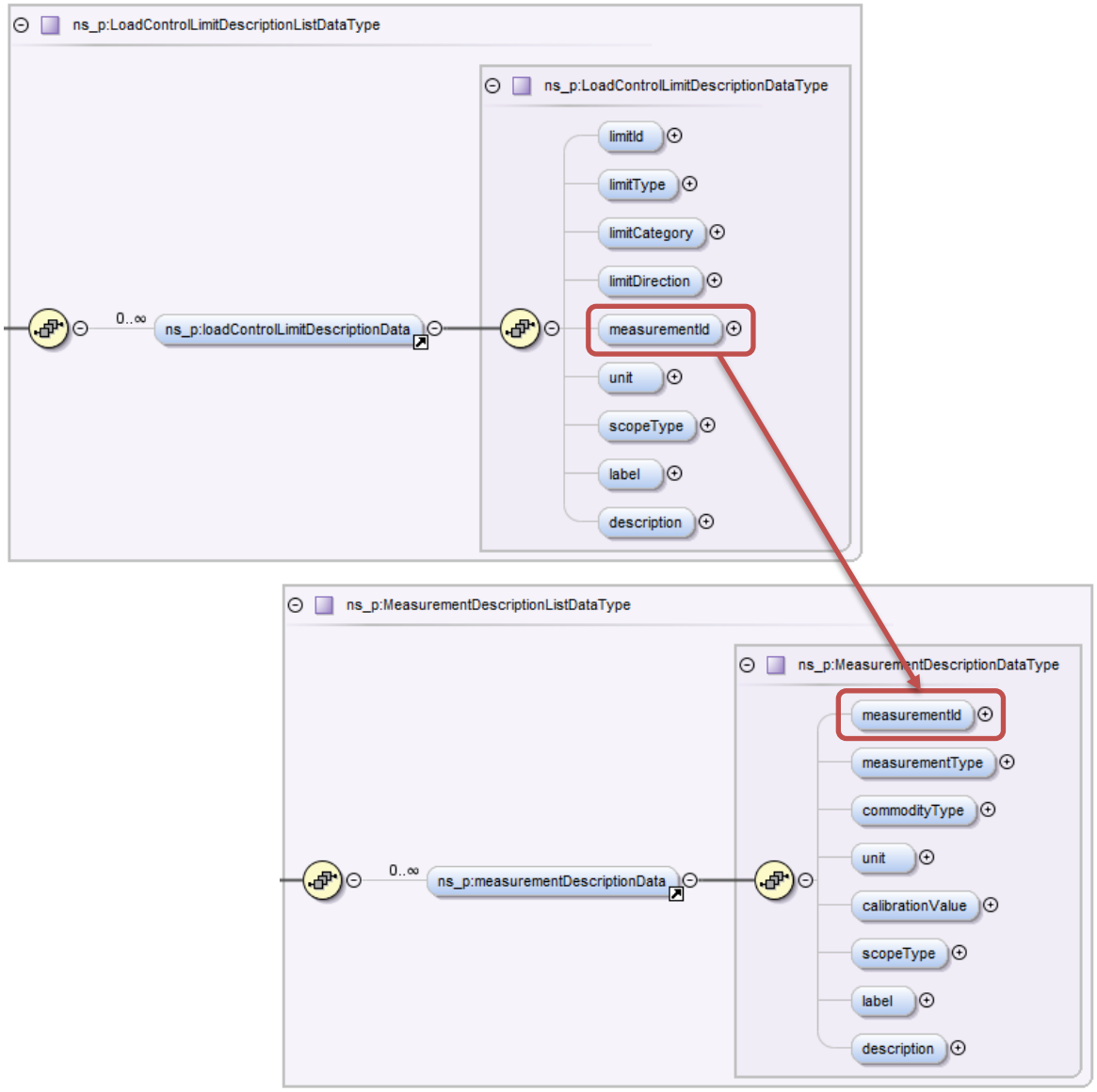

Figure 13: Reference from loadControlLimitDescriptionListData to measurementDescriptionListData via the measurementId identifier

An example for a complete SPINE message with the identification element “measurementId” for reference:

```
<?xml version="1.0" encoding="UTF-8"?>
<datagram xmlns="http://docs.eebus.org/spine/xsd/v1"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://docs.eebus.org/spine/xsd/v1
file:../../EEBus_SPINE_TS_Datagram.xsd">
    <header>
        <specificationVersion>1.1.1</specificationVersion>
        <addressSource>
            <device>d:_i:46925_TestTempSensor_1</device>
            <entity>2</entity>
            <entity>1</entity>
            <entity>1</entity>
            <feature>1</feature>
        </addressSource>
        <addressDestination>
```



```
            <device>d:_i:46925_TestSHM_1</device>
            <entity>1</entity>
            <feature>1</feature>
        </addressDestination>
        <msgCounter>15</msgCounter>
        <msgCounterReference>8</msgCounterReference>
        <cmdClassifier>reply</cmdClassifier>
    </header>
    <payload>
        <cmd>
            <function>measurementListData</function>
            <filter>
                <cmdControl>
                    <partial/>
                </cmdControl>
            </filter>
            <measurementListData>
                <measurementData>
                    <measurementId>1</measurementId>
                    <valueType>value</valueType>
                    <timestamp>2015-11-05T10:14:00.0Z</timestamp>
                    <value>22.5</value>
                    <valueSource>measuredValue</valueSource>
                </measurementData>
            </measurementListData>
        </cmd>
    </payload>
</datagram>
```

For a use case where very complex data is needed, which has to be sent in one "atomic" message (a single message that cannot be split into several (smaller) messages), *complex classes* can be used. Each complex class is developed and released like “normal classes” and described in [ResourceSpecification], too.

An excerpt of the XSD of the "SmartEnergyManagementPs" complex class:

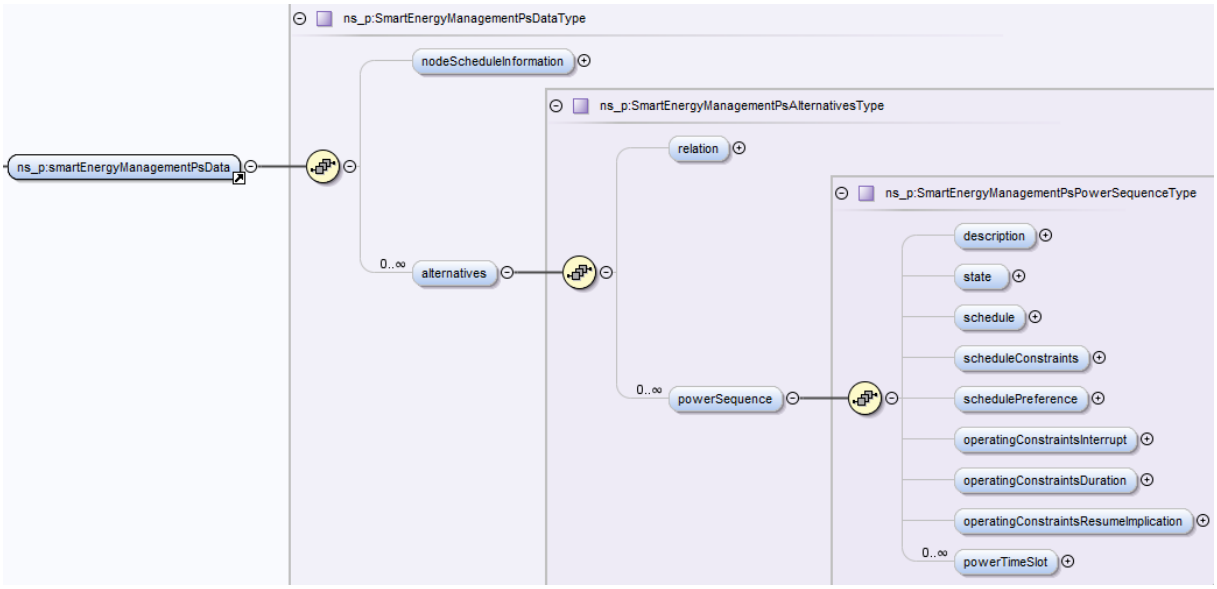

Figure 14: Complex class example (SmartEnergyManagementPs) - Overview

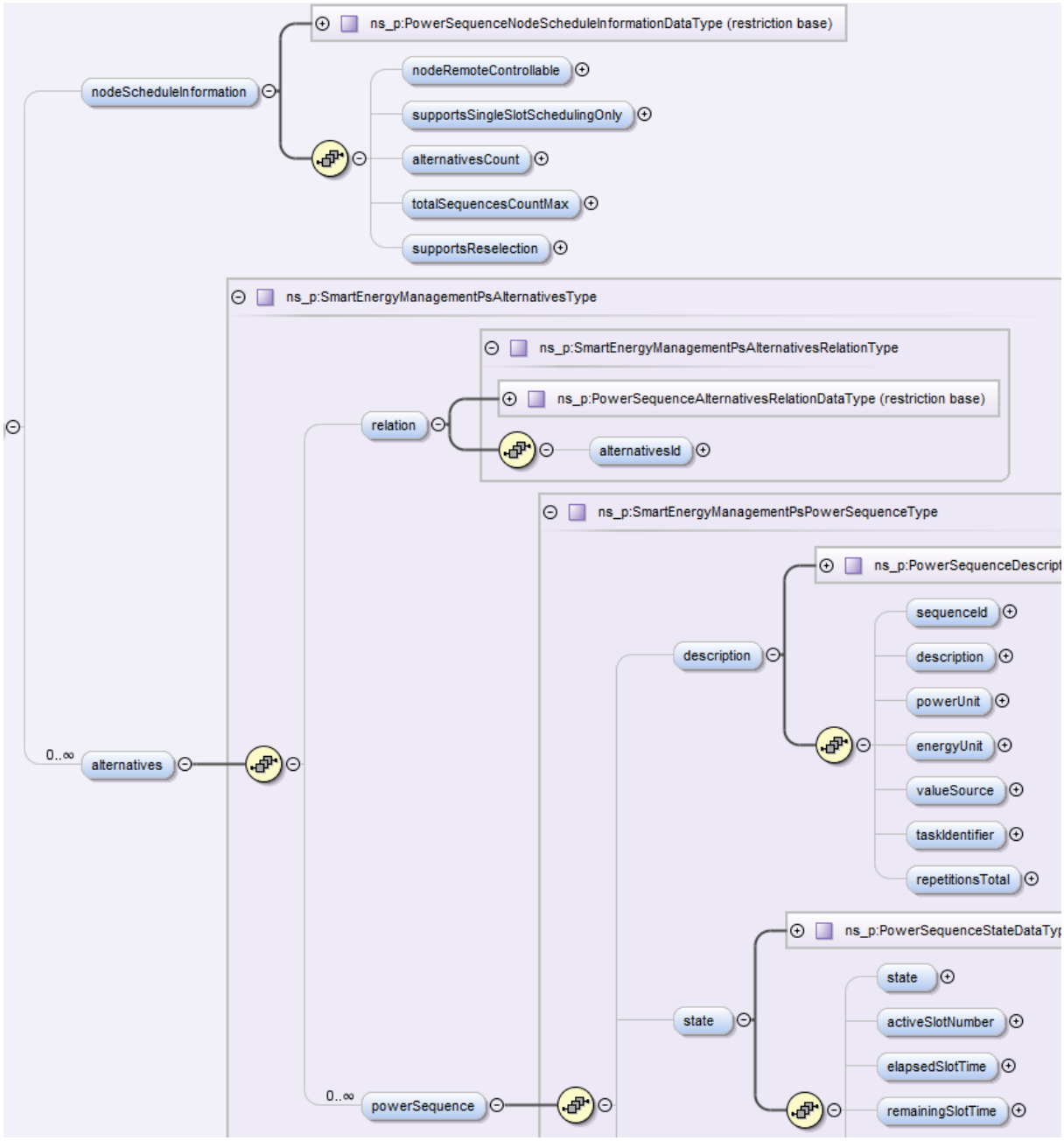

Figure 15: Complex class example (SmartEnergyManagementPs) - Detail excerpt

It can be seen that only already in standard classes defined data is used (as *restriction base*) and arranged in a way that fits the specific requirements of the complex class.

Please read [ResourceSpecification], sections "Class descriptions" -> "Complex Classes" and "Standard Classes" for more information.

3.4 Role concept

Each feature has a role for its context (either "server" or "client", or in some cases "special").

The owner of some data is always the "server". It can:

- inform clients about updates of its data,

- reply to a client with requested information,
- accept (or decline) changes (via the "write" classifier) or actions (“call” classifier).

The client can do the above listed things in the other direction (receiving updates, requesting information, etc.).

The role "special" is reserved for very specific functionalities only (e.g. in the *NodeManagement* complex class, [ProtocolSpecification], chapter "Functional commissioning"; you may skip this for the first reading).

Please read [ProtocolSpecification], chapter "Architecture requirements" and section "SPINE Datagram" -> "Payload" -> "Ownership" for some more information.

3.5 Acknowledge and error messages

Messages may be acknowledged by the receiver, so the sender knows that the message was successfully sent to its destination. The acknowledgement message may be requested by the sender. If an error occurred during reception or analysis of a SPINE message, an error message is sent by the receiver of a message to the original sender. Furthermore, delays of acknowledgements and responses can be communicated.

Please read [ProtocolSpecification], section "SPINE Datagram" -> "Header" -> "Acknowledgement concept" for details.

3.6 Binding and Subscription

There are several ways to "tell" devices where to send their messages (e.g. for updates of their status, new sensor values, etc.). One way is to configure the device statically. This presumes some kind of interface to the device, which is often not practicable. Another way is to use binding and subscription mechanisms. To which device a binding or subscription shall be established can be determined via device discovery mechanisms (see section 3.7).

Binding is rather a process-related concept to specify dedicated interactions between two devices (e.g. a light accepts requests to switch on or off only from a “known” (bound) light switch). This is also a way for a device to restrict access from other devices for specific purposes (e.g. only one energy manager may shift the demand of a device). Please read the following section for details:

- [ProtocolSpecification], section "Functional commissioning" -> "Binding"

Subscription is used by clients to receive unsolicited updates from some server instance (e.g. new temperature values). Please read the following section for details:

- [ProtocolSpecification], section "Functional commissioning" -> "Subscription"

3.7 Device discovery

Every device has some functionality that it can announce to other devices via the device discovery mechanisms of SPINE. Some automatism regarding the finding of appropriate communication partners are possible as soon as the detailed discovery has been done at the communication partner. However, manufacturers are advised to complement this with some user interaction (e.g. some kind of push-button method or configuration procedure, e.g. via web-interface).

In general, server functionality shall be announced via the detailed discovery, whereas client functionality may be announced (but at least one client feature should be announced, e.g. with a feature type "Generic", to enable a server to recognise a communications partner as a possible client).

After a successful connection establishment via the applied communications technology (e.g. SHIP), a device performs a detailed discovery of the communications partner (if not already known from a prior connection). After the detailed discovery, a device may decide to close the connection (e.g. because there is no matching feature type available) or to keep the connection open.

Please read the following section for details:

- [ProtocolSpecification], section "Functional commissioning" -> "Detailed discovery"

However, not every aspect of a device may be discoverable with detailed discovery (e.g. client functionality or currently not available (dynamic) server functionality). In this case, the concept of "Use Case Discovery" can be used.

Please read the following section for details:

- [ProtocolSpecification], section "Functional commissioning" -> "Use Case discovery"

3.8 Addressing schema

The SPINE address is composed of three address levels:

- device: physical device, e.g. a fridge/freezer combination
- entity: logical (sub) device, e.g. a freezer
- feature: functionality, e.g. superfreeze

The following relations apply:

- 1 device has N entities
- 1 entity belongs to 1 device
- 1 entity may have 0..N sub-entities
- 1 sub-entity belongs to 1 entity
- 1 entity has M features
- 1 feature belongs to 1 entity

The address elements in the SPINE header have appropriate sub-elements for the three address levels. The device element is a string (see below), the entity and feature elements are each a 32-bit integer.

The device string may contain the following (notated in ERE: Extended Regular Expression):

```
d:_(i:[1-9][0-9]*|n:[a-zA-Z0-9-]+)_[^\p{Cc}\p{Cf}\p{Z}]+
```

The pattern requires the address to first state "d:", then the pattern of the vendor specific extensions follows to give the address the needed uniqueness. After that, the vendor states a string containing a vendor-wide unique address that MAY consist of the following characters: Any Unicode character EXCEPT for such Unicode characters belonging to the so-called "general category" definitions "Control", "Format", and "Separator". Among others, this means white space characters are not permitted.

Examples for possible device strings:

- d:_i:46925_ABCabc-123
- d:_i:46925_0123456789

Please read the following sections for details:

- [ProtocolSpecification], section "SPINE Datagram" -> "Header" -> "Address information"
- [ProtocolSpecification], section "Functional commissioning" -> "Detailed discovery" -> "Basic definitions and rules" (and sub-sections)

3.9 Destination List

The previous sections basically focus on the communication between two devices. This is also the focus of the document [ProtocolSpecification]. However, a device may also provide the possibility to forward messages to another device. This can especially be the case for devices that can bridge devices of a specific communications technology into the SPINE concept. In order to provide a list of devices where message forwarding is offered for, the so-called “DestinationList” concept can be used. You can find detailed information in the following section:

- [ProtocolSpecification], section "Functional commissioning" -> "Destination list"

3.10 Partial data concept

The most common (or “default”) method to get data from a server or to modify it is to exchange a complete SPINE function instance. To give an example, a client can send a “read” command to a server’s "deviceDiagnosisStateData" function and the server then replies with a full copy of this function. However, there are situations where it is preferable or even required to exchange only a part of a function. In order to explain this briefly we consider as example a server feature that has a function with a number of child elements:

1. The server permits a client to read it. In this example the server also permits the client to modify the function – but only two of the function’s child elements. I.e. the client is not permitted to send a complete function with a "write" command to the server. Rather, it can just send a "write" command that only contains those child elements the server permits for modification.
2. The server notifies changes of the function to subscribed client features. In this example we assume the function is quite large, i.e. it has many child elements. In this example it may be

482 favourable to notify just those parts of the function that have changed – in contrast to notifying
483 the whole function every time even if just a single child element changed.

484 The concept to exchange only parts of a function is called “restricted function exchange”. You can
485 find detailed information in the following sections:

486 - [ProtocolSpecification], sections "SPINE Datagram" -> "Payload" -> "Restricted function
487 exchange with cmdOptions" and Annex "Recommendations for Restricted Function
488 Exchange"

489

