OPC 10000-100

OPC Unified Architecture

Part 100: Devices

Release 1.02

2019-04-18

OPC UA Specification

Specification Type	Industry Standard Specification	Comments:	
Document Number	**OPC 10000-100**		
Title:	OPC Unified Architecture Devices	Date:	2019-04-18
Version:	Release 1.02	Software Source:	MS-Word OPC 10000-100 - UA Specification Part 100 - Devices 1.02.docx
Author:	OPC Foundation	Status:	Release

CONTENTS

FIGURES

TABLES

OPC FOUNDATION

UNIFIED ARCHITECTURE –

FOREWORD

This specification is the specification for developers of OPC UA applications. The specification is a result of an analysis and design process to develop a standard interface to facilitate the development of applications by multiple vendors that shall inter-operate seamlessly together.

Revision 1.2 Highlights

This revision contains extensions to Version 1.1.

The following table includes the Mantis issues resolved with this revision.

Mantis ID	Summary	Resolution
2708	Inconsistent BrowseNames	Already fixed in 1.01.09 and IEC version.
3096	Need inverse ref from connection point to device	Added requirement to ConnectionPoint.
3148	NamespaceMetaData Object missing	Added to this version.
3288	BrowseNames for optional Placeholder inconsistent	Fixed as suggested.
3221	Spelling error	Fixed as suggested.
3193	NetworkSet only mandatory as part of DeviceCommunication facet	Removed “mandatory” from heading.
3191	Broken reference	Fixed as suggested.
3419	It would be good to have something like a SoftwareType	SoftwareType was incorporated in this revision.
4049	New element to structure different aspects of a device	• Defined a "ComponentType" that can be used to model any HW or SW element of a device. • Annex B provides guidelines on how to structure such a device
3873	Clarification of subtypes of BaseVariableType	Fixed UIElement to derive from BaseDataVariableType.
4533	Add “DeviceFeatures” folder in DI	Specified “DeviceFeatures” Object underneath “DeviceSet”

1 Scope

This part of the OPC UA specification is an extension of the overall OPC Unified Architecture specification series and defines the information model associated with *Devices*. This specification describes three models which build upon each other as follows:

- The (base) Device Model is intended to provide a unified view of devices and their hardware and software parts irrespective of the underlying device protocols.
- The Device Communication Model adds Network and Connection information elements so that communication topologies can be created.
- The Device Integration Host Model finally adds additional elements and rules required for host systems to manage integration for a complete system. It allows reflecting the topology of the automation system with the devices as well as the connecting communication networks.

2 Reference documents

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments and errata) applies.

OPC 10000-1*, OPC Unified Architecture - Part 1: Overview and Concepts*
http://www.opcfoundation.org/UA/Part1/

OPC 10000-3, *OPC Unified Architecture - Part 3: Address Space Model*
http://www.opcfoundation.org/UA/Part3/

OPC 10000-4, *OPC Unified Architecture - Part 4: Services*
http://www.opcfoundation.org/UA/Part4/

OPC 10000-5, *OPC Unified Architecture - Part 5: Information Model*
http://www.opcfoundation.org/UA/Part5/

OPC 10000-6, *OPC Unified Architecture - Part 6: Mappings*
http://www.opcfoundation.org/UA/Part6/

OPC 10000-7, *OPC Unified Architecture - Part 7: Profiles*
http://www.opcfoundation.org/UA/Part7/

OPC 10000-8, *OPC Unified Architecture - Part 8: Data Access*
http://www.opcfoundation.org/UA/Part8/

OPC 10000-9, *OPC Unified Architecture - Part 9: Alarms and Conditions*
http://www.opcfoundation.org/UA/Part9/

OPC 10001-5, *OPC Unified Architecture V1.04 - Amendment 5: Dictionary Reference*

OPC 10001-7, *OPC Unified Architecture V1.04 - Amendment 7: Interfaces and AddIns*

[OPC 10020 - ADI], *OPC UA Companion Specification for Analyser Devices*

[OPC 30000 - PLCopen], *OPC UA Companion Specification for PLCopen*

IEC 62769, *Field Device Integration (FDI)*

NAMUR Recommendation NE107*:* Self-monitoring and diagnosis of field devices

3 Terms, definitions, abbreviations, and used data types

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments and errata) applies.

OPC 10000-1, OPC 10000-3, and OPC 10000-8 as well as the following apply.

3.1.1
block
functional *Parameter* grouping entity

Note 1 to entry: It could map to a function block (see IEC 62769) or to the resource parameters of the *Device* itself.

3.1.2
blockMode
mode of operation (target mode, permitted modes, actual mode, and normal mode) for a *Block*

Note 1 to entry: Further details about *Block* modes are defined by standard organisations.

3.1.3
Communication Profile
fixed set of mapping rules to allow unambiguous interoperability between *Devices* or Applications, respectively

Note 1 to entry: Examples of such profiles are the “Wireless communication network and communication profiles for WirelessHART” in IEC 62591 and the Protocol Mappings for OPC UA in OPC 10000-6.

3.1.4
Connection Point
logical representation of the interface between a *Device* and a *Network*

3.1.5
device
independent physical entity capable of performing one or more specified functions in a particular context and delimited by its interfaces

Note 1 to entry: See IEC 61499-1.

Note 2 to entry: *Devices* provide sensing, actuating, communication, and/or control functionality. Examples include transmitters, valve controllers, drives, motor controllers, PLCs, and communication gateways.

Note 3 to entry: A *Device* can be a system (topology) of other *Devices*, components, or parts.

3.1.6
Device Integration Host
Server that manages integration of multiple *Devices* in an automation system

3.1.7
Device Topology
arrangement of *Networks* and *Devices* that constitute a communication topology

3.1.8
fieldbus
communication system based on serial data transfer and used in industrial automation or process control applications

Note 1 to entry: See IEC 61784.

Note 2 to entry: Designates the communication bus used by a *Device*.

3.1.9
Parameter
variable of the *Device* that can be used for configuration, monitoring or control purposes

Note 1 to entry: In the information model it is synonymous to an OPC UA *DataVariable*.

3.1.10
Network
means used to communicate with one specific protocol

3.2 Abbreviations

ADI	Analyser Device Integration
CP	Communication Processor (hardware module)
CPU	Central Processing Unit (of a *Device*)
DA	Data Access
DI	Device Integration (the short name for this specification)
ERP	Enterprise Resource Planning
IRDI	International Registration Data Identifiers
UA	Unified Architecture
UML	Unified Modelling Language
XML	Extensible Mark-up Language

3.3 Conventions used in this Document

3.3.1 Conventions for Terms

Terms in this document are written in CamelCase and italicized.

3.3.2 Conventions for Node Descriptions

Node definitions are specified using tables (see Table 2).

Attributes are defined by providing the *Attribute* name and a value, or a description of the value.

References are defined by providing the *ReferenceType* name, the *BrowseName* of the *TargetNode* and its *NodeClass*.

- If the *TargetNode* is a component of the *Node* being defined in the table the *Attributes* of the composed *Node* are defined in the same row of the table.
- The *DataType* is only specified for *Variables*; "[<number>]" indicates a single-dimensional array, for multi-dimensional arrays the expression is repeated for each dimension (e.g. [2][3] for a two-dimensional array). For all arrays the *ArrayDimensions* is set as identified by <number> values. If no <number> is set, the corresponding dimension is set to 0, indicating an unknown size. If no number is provided at all the *ArrayDimensions* can be omitted. If no brackets are provided, it identifies a scalar *DataType* and the *ValueRank* is set to the corresponding value (see OPC 10000-3). In addition, *ArrayDimensions* is set to null or is omitted. If it can be Any or ScalarOrOneDimension, the value is put into "{<value>}", so either "{Any}" or "{ScalarOrOneDimension}" and the *ValueRank* is set to the corresponding value (see OPC 10000-3) and the *ArrayDimensions* is set to null or is omitted. Examples are given in Table 1.

Table 1 – Examples of DataTypes

Notation	Data-Type	Value-Rank	Array-Dimensions	Description
Int32	Int32	-1	omitted or null	A scalar Int32.
Int32[]	Int32	1	omitted or {0}	Single-dimensional array of Int32 with an unknown size.
Int32[][]	Int32	2	omitted or {0,0}	Two-dimensional array of Int32 with unknown sizes for both dimensions.
Int32[3][]	Int32	2	{3,0}	Two-dimensional array of Int32 with a size of 3 for the first dimension and an unknown size for the second dimension.
Int32[5][3]	Int32	2	{5,3}	Two-dimensional array of Int32 with a size of 5 for the first dimension and a size of 3 for the second dimension.
Int32{Any}	Int32	-2	omitted or null	An Int32 where it is unknown if it is scalar or array with any number of dimensions.
Int32{ScalarOrOneDimension}	Int32	-3	omitted or null	An Int32 where it is either a single-dimensional array or a scalar.

- The *TypeDefinition* is specified for *Objects* and *Variables*.
- The *TypeDefinition* column specifies a symbolic name for a *NodeId*, i.e. the specified *Node* points with a *HasTypeDefinition Reference* to the corresponding *Node*.
- The *ModellingRule* of the referenced component is provided by specifying the symbolic name of the rule in the *ModellingRule* column. In the *AddressSpace*, the *Node* shall use a *HasModellingRule Reference* to point to the corresponding *ModellingRule Object*.

If the *NodeId* of a *DataType* is provided, the symbolic name of the *Node* representing the *DataType* shall be used.

Nodes of all other *NodeClasses* cannot be defined in the same table; therefore only the used *ReferenceType*, their *NodeClass* and their *BrowseName* are specified. A reference to another part of this document points to their definition.

Table 2 illustrates the table. If no components are provided, the *DataType*, *TypeDefinition* and *ModellingRule* columns may be omitted and only a Comment column is introduced to point to the *Node* definition.

Table 2 – Type Definition Table

Attribute	Value				
Attribute name	Attribute value. If it is an optional Attribute that is not set "--" will be used.				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
ReferenceType name	*NodeClass* of the *TargetNode*.	*BrowseName* of the target Node. If the *Reference* is to be instantiated by the server, then the value of the target Node's *BrowseName* is "--".	*DataType* of the referenced *Node*, only applicable for *Variables*.	*TypeDefinition* of the referenced *Node*, only applicable for *Variables* and *Objects*.	Referenced *ModellingRule* of the referenced *Object*.
NOTE Notes referencing footnotes of the table content.					

Components of *Nodes* can be complex that is containing components by themselves. The *TypeDefinition*, *NodeClass*, *DataType* and *ModellingRule* can be derived from the type definitions, and the symbolic name can be created as defined in Annex A. Therefore, those containing components are not explicitly specified; they are implicitly specified by the type definitions.

3.3.3 NodeIds and BrowseNames

3.3.3.1 NodeIds

The *NodeIds* of all Nodes described in this standard are only symbolic names. Annex A defines the actual *NodeIds*.

The symbolic name of each *Node* defined in this specification is its *BrowseName*, or, when it is part of another *Node*, the *BrowseName* of the other *Node*, a ".", and the *BrowseName* of itself. In this case "part of" means that the whole has a *HasProperty* or *HasComponent Reference* to its part. Since all *Nodes* not being part of another *Node* have a unique name in this specification, the symbolic name is unique.

The *NamespaceUri* for all *NodeIds* defined in this specification is defined in Annex A. The *NamespaceIndex* for this *NamespaceUri* is vendor-specific and depends on the position of the *NamespaceUri* in the *Server* namespace table.

Note that this specification not only defines concrete *Nodes*, but also requires that some *Nodes* shall be generated, for example one for each *Session* running on the *Server*. The *NodeIds* of those *Nodes* are vendor-specific, including the *NamespaceUri*. But the *NamespaceUri* of those *Nodes* cannot be the *NamespaceUri* used for the *Nodes* defined in this specification, because they are not defined by this specification but generated by the *Server*.

3.3.3.2 BrowseNames

The text part of the *BrowseNames* for all *Nodes* defined in this specification is specified in the tables defining the *Nodes*. The *NamespaceUri* for all *BrowseNames* defined in this specification is defined in Annex A.

If the *BrowseName* is not defined by this specification, a namespace index prefix like '0:EngineeringUnits' is added to the *BrowseName*. This is typically necessary if a *Property* of another specification is overwritten or used in the OPC UA types defined in this specification. Clause 11 provides the namespaces used in this specification.

3.3.4 Common Attributes

3.3.4.1 General

The *Attributes* of *Nodes*, their *DataTypes* and descriptions are defined in OPC 10000-3. *Attributes* not marked as optional are mandatory and shall be provided by a *Server*. The following tables define if the *Attribute* value is defined by this specification or if it is vendor-specific.

For all *Nodes* specified in this specification, the *Attributes* named in Table 3 shall be set as specified in the table.

Table 3 – Common Node Attributes

Attribute	Value
DisplayName	The DisplayName is a LocalizedText. Each server shall provide the DisplayName identical to the BrowseName of the Node for the LocaleId “en”. Whether the server provides translated names for other LocaleIds is vendor-specific.
Description	Optionally a vendor-specific description is provided.
NodeClass	Shall reflect the NodeClass of the Node.
NodeId	The NodeId is described by BrowseNames as defined in 3.3.3.1
WriteMask	Optionally the WriteMask Attribute can be provided. If the WriteMask Attribute is provided, it shall set all non-vendor-specific Attributes to not writable. For example, the Description Attribute may be set to writable since a Server may provide a vendor-specific description for the Node. The NodeId shall not be writable, because it is defined for each Node in this specification.
UserWriteMask	Optionally the UserWriteMask Attribute can be provided. The same rules as for the WriteMask Attribute apply.
RolePermissions	Optionally vendor-specific role permissions can be provided.
UserRolePermissions	Optionally the role permissions of the current Session can be provided. The value is vendor-specific and depend on the RolePermissions Attribute (if provided) and the current Session.
AccessRestrictions	Optionally vendor-specific access restrictions can be provided.

3.3.4.2 Objects

For all *Objects* specified in this specification, the *Attributes* named in Table 4 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 4 – Common Object Attributes

Attribute	Value
EventNotifier	Whether the Node can be used to subscribe to Events or not is vendor-specific.

3.3.4.3 Variables

For all *Variables* specified in this specification, the *Attributes* named in Table 5 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 5 – Common Variable Attributes

Attribute	Value
MinimumSamplingInterval	Optionally, a vendor-specific minimum sampling interval is provided.
AccessLevel	The access level for Variables used for type definitions is vendor-specific, for all other Variables defined in this specification, the access level shall allow reading; other settings are vendor-specific.
UserAccessLevel	The value for the UserAccessLevel Attribute is vendor-specific. It is assumed that all Variables can be accessed by at least one user.
Value	For Variables used as InstanceDeclarations, the value is vendor-specific; otherwise it shall represent the value described in the text.
ArrayDimensions	If the ValueRank does not identify an array of a specific dimension (i.e. ValueRank <= 0) the ArrayDimensions can either be set to null or the Attribute is missing. This behaviour is vendor-specific. If the ValueRank specifies an array of a specific dimension (i.e. ValueRank > 0) then the ArrayDimensions Attribute shall be specified in the table defining the Variable.
Historizing	The value for the Historizing Attribute is vendor-specific.
AccessLevelEx	If the AccessLevelEx Attribute is provided, it shall have the bits 8, 9, and 10 set to 0, meaning that read and write operations on an individual Variable are atomic, and arrays can be partly written.

3.3.4.4 VariableTypes

For all *VariableTypes* specified in this specification, the *Attributes* named in Table 6 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 6 – Common VariableType Attributes

Attributes	Value
Value	Optionally a vendor-specific default value can be provided.
ArrayDimensions	If the ValueRank does not identify an array of a specific dimension (i.e. ValueRank <= 0) the ArrayDimensions can either be set to null or the Attribute is missing. This behaviour is vendor-specific. If the ValueRank specifies an array of a specific dimension (i.e. ValueRank > 0) then the ArrayDimensions Attribute shall be specified in the table defining the VariableType.

3.3.4.5 Methods

For all *Methods* specified in this specification, the *Attributes* named in Table 7 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 7 – Common Method Attributes

Attributes	Value
Executable	All Methods defined in this specification shall be executable (Executable Attribute set to “True”), unless it is defined differently in the Method definition.
UserExecutable	The value of the UserExecutable Attribute is vendor-specific. It is assumed that all Methods can be executed by at least one user.

4 Introduction to OPC UA

4.1 What is OPC UA?

OPC UA is an open and royalty free set of standards designed as a universal communication protocol. While there are numerous communication solutions available, OPC UA has key advantages:

- A state of art security model (see OPC UA Part 2).
- A fault tolerant communication protocol.
- An information modelling framework that allows application developers to represent their data in a way that makes sense to them.

OPC UA has a broad scope which delivers economies of scale for application developers. This means that a larger number of high quality applications at a reasonable cost are available. When combined with semantic models such as OPC UA for IO-Link, OPC UA makes it easier for end users to access data via generic commercial applications.

The OPC UA model is scalable from small devices to ERP systems. OPC UA Servers process information locally and then provide that data in a consistent format to any application requesting data - ERP, MES, PMS, Maintenance Systems, HMI, Smartphone or a standard Browser, for examples. For a more complete overview see The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments and errata) applies.

OPC 10000-1.

4.2 Basics of OPC UA

As an open standard, OPC UA is based on standard internet technologies, like TCP/IP, HTTP, Web Sockets.

As an extensible standard, OPC UA provides a set of Services (see OPC 10000-4) and a basic information model framework. This framework provides an easy manner for creating and exposing vendor defined information in a standard way. More importantly all OPC UA Clients are expected to be able to discover and use vendor-defined information. This means OPC UA users can benefit from the economies of scale that come with generic visualization and historian applications. This specification is an example of an OPC UA Information Model designed to meet the needs of developers and users.

OPC UA Clients can be any consumer of data from another device on the network to browser based thin clients and ERP systems. The full scope of OPC UA applications is shown in Figure 1.

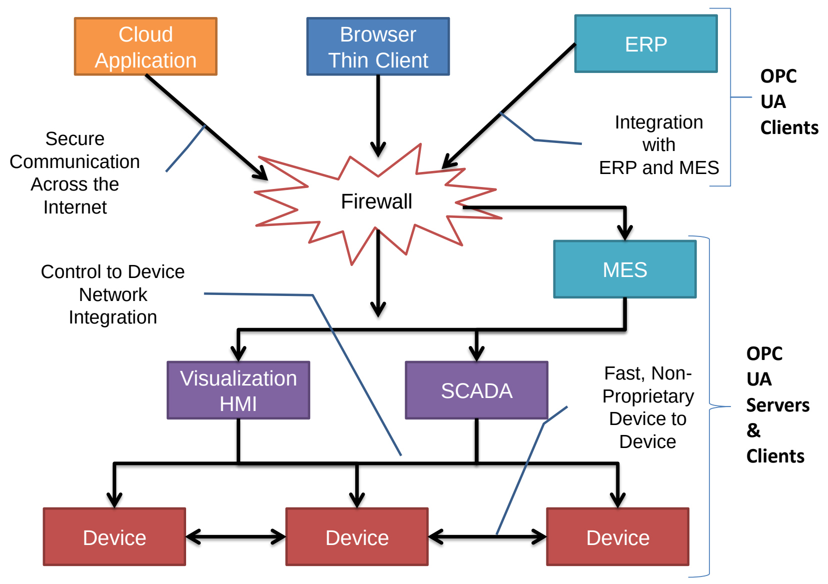
Cloud Application
Browser Thin Client
ERP
OPC UA Clients
Secure Communication Across the Internet
Integration with ERP and MES
Firewall
MES
Control to Device Network Integration
Visualization HMI
SCADA
Fast, Non-Proprietary Device to Device
OPC UA Servers & Clients
Device
Device
Device

Figure 1 – The Scope of OPC UA within an Enterprise

OPC UA provides a robust and reliable communication infrastructure having mechanisms for handling lost messages, failover, heartbeat, etc. With its binary encoded data, it offers a high-performing data exchange solution. Security is built into OPC UA as security requirements become more and more important especially since environments are connected to the office network or the internet and attackers are starting to focus on automation systems.

4.2.1 Information Modelling in OPC UA

4.2.1.1 Concepts

OPC UA provides a framework that can be used to represent complex information as *Objects* in an *AddressSpace* which can be accessed with standard services. These *Objects* consist of *Nodes* connected by *References*. Different classes of *Nodes* convey different semantics. For example, a *Variable Node* represents a value that can be read or written. The *Variable Node* has an associated *DataType* that can define the actual value, such as a string, float, structure etc. It can also describe the *Variable* value as a variant. A Method Node represents a function that can be called. Every *Node* has a number of *Attributes* including a unique identifier called a *NodeId* and non-localized name called as *BrowseName*.

Object and *Variable Nodes* represent instances and they always reference a *TypeDefinition* (*ObjectType* or *VariableType*) Node which describes their semantics and structure. Figure 2 illustrates the relationship between an instance and its *TypeDefinition*.

The type *Nodes* are templates that define all the children that can be present in an instance of the type. In the example in Figure 2 the SomeType *ObjectType* defines two *Properties*: Property1 and Property2. All instances of SomeType are expected to have the same children with the same *BrowseNames*. Within a type the *BrowseNames* uniquely identify the children. This means *Client* applications can be designed to search for children based on the *BrowseNames* from the type instead of *NodeIds*. This eliminates the need for manual reconfiguration of systems if a *Client* uses types that multiple *Servers* implement.

OPC UA also supports the concept of sub-typing. This allows a modeller to take an existing type and extend it. There are rules regarding sub-typing defined in OPC 10000-3, but in general they allow the extension of a given type or the restriction of a *DataType*. For example, the modeller may decide that the existing *ObjectType* in some cases needs an additional Variable. The modeller can create a subtype of the *ObjectType* and add the *Variable*. A *Client* that is expecting the parent type can treat the new type as if it was of the parent type. Regarding *DataTypes*, subtypes can only restrict. If a *Variable* is defined to have a numeric value, a subtype could restrict it to a float.

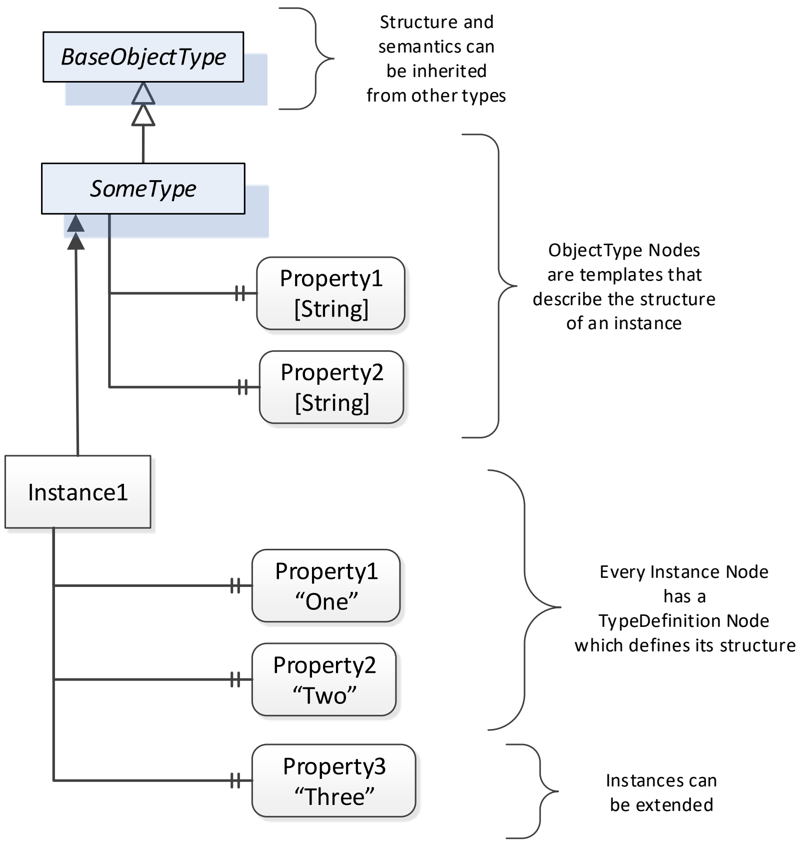
BaseObjectType
Structure and semantics can be inherited from other types
SomeType
Property1 [String]
Property2 [String]
ObjectType Nodes are templates that describe the structure of an instance
Instance1
Property1 "One"
Property2 "Two"
Every Instance Node has a TypeDefinition Node which defines its structure
Property3 "Three"
Instances can be extended

Figure 2 – The Relationship between Type Definitions and Instances

References allow *Nodes* to be connected in ways that describe their relationships. All *References* have a *ReferenceType* that specifies the semantics of the relationship. *References* can be hierarchical or non-hierarchical. Hierarchical references are used to create the structure of *Objects* and *Variables*. Non-hierarchical are used to create arbitrary associations. Applications can define their own *ReferenceType* by creating subtypes of an existing *ReferenceType*. Subtypes inherit the semantics of the parent but may add additional restrictions.

4.2.1.2 Graphical Notation

Figure 2 uses a notation that was developed for the OPC UA specification. The notation is summarized in Figure 3. UML representations can also be used; however, the OPC UA notation is less ambiguous because there is a direct mapping from the elements in the figures to Nodes in the *AddressSpace* of an OPC UA Server.

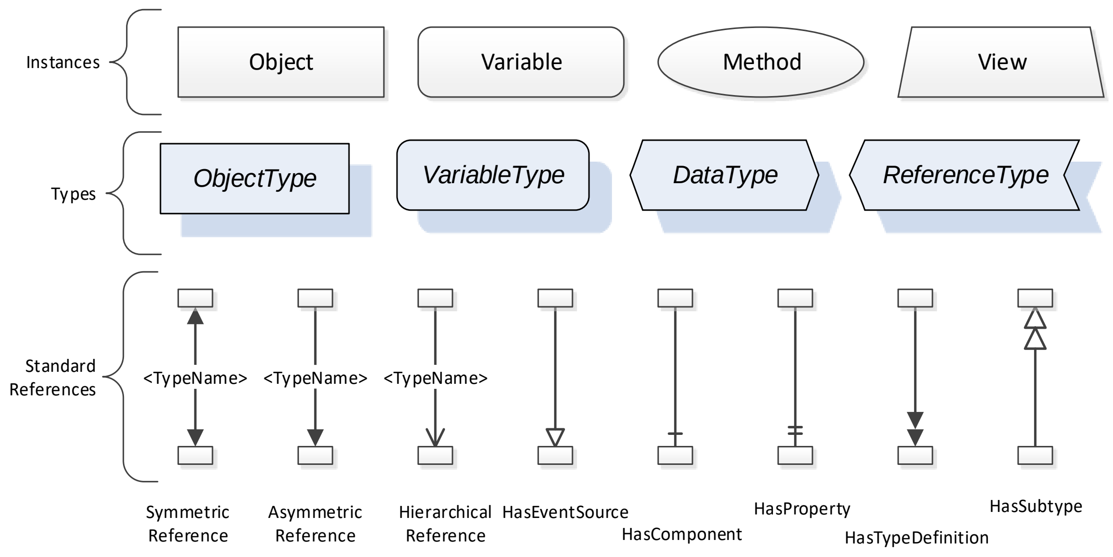

Figure 3 – The OPC UA Information Model Notation

A complete description of the different types of Nodes and References can be found in OPC 10000-3 and the base structure is described in OPC 10000-5.

4.2.2 OPC UA Profiles

OPC UA specification defines a very wide range of functionality in its base information model. It is not expected that all *Clients* or *Servers* support all functionality in the OPC UA specifications. OPC UA includes the concept of *Profiles*, which segments the functionality into testable certifiable units. This allows the definition of functional subsets (that are expected to be implemented) within a companion specification. The *Profiles* do not restrict functionality, but generate requirements for a minimum set of functionalities (see OPC 10000-7).

4.2.3 Namespaces

OPC UA allows information from many different sources to be combined into a single coherent *AddressSpace*. Namespaces are used to make this possible by eliminating naming and id conflicts between information from different sources. *Namespaces* in OPC UA have a globally unique string called a *NamespaceUri* and a locally unique integer called a *NamespaceIndex*. The *NamespaceIndex* is only unique within the context of a Session between an OPC UA *Client* and an OPC UA *Server*. The *Services* defined for OPC UA use the *NamespaceIndex* to specify the *Namespace* for qualified values.

There are two types of values in OPC UA that are qualified with *Namespaces*: *NodeIds* and *QualifiedNames*. *NodeIds* are globally unique identifiers for Nodes. This means the same *Node* with the same *NodeId* can appear in many *Servers*. This, in turn, means *Clients* can have built in knowledge of some *Nodes*. OPC UA Information Models generally define globally unique *NodeIds* for the *TypeDefinitions* defined by the *Information Model*.

QualifiedNames are non-localized names qualified with a *Namespace*. They are used for the *BrowseNames* of *Nodes* and allow the same names to be used by different information models without conflict. *TypeDefinitions* are not allowed to have children with duplicate *BrowseNames*; however, instances do not have that restriction.

4.2.4 Companion Specifications

An OPC UA companion specification describes an *Information Model* by defining *ObjectTypes*, *VariableTypes*, *DataTypes* and *ReferenceTypes* that represent a specific semantic relevant for the companion specification.

5 Device model

5.1 General

Figure 4 depicts the main *ObjectTypes* of the base device model and their relationship. The drawing is not intended to be complete. For the sake of simplicity only a few components and relations were captured to give a rough idea of the overall structure.

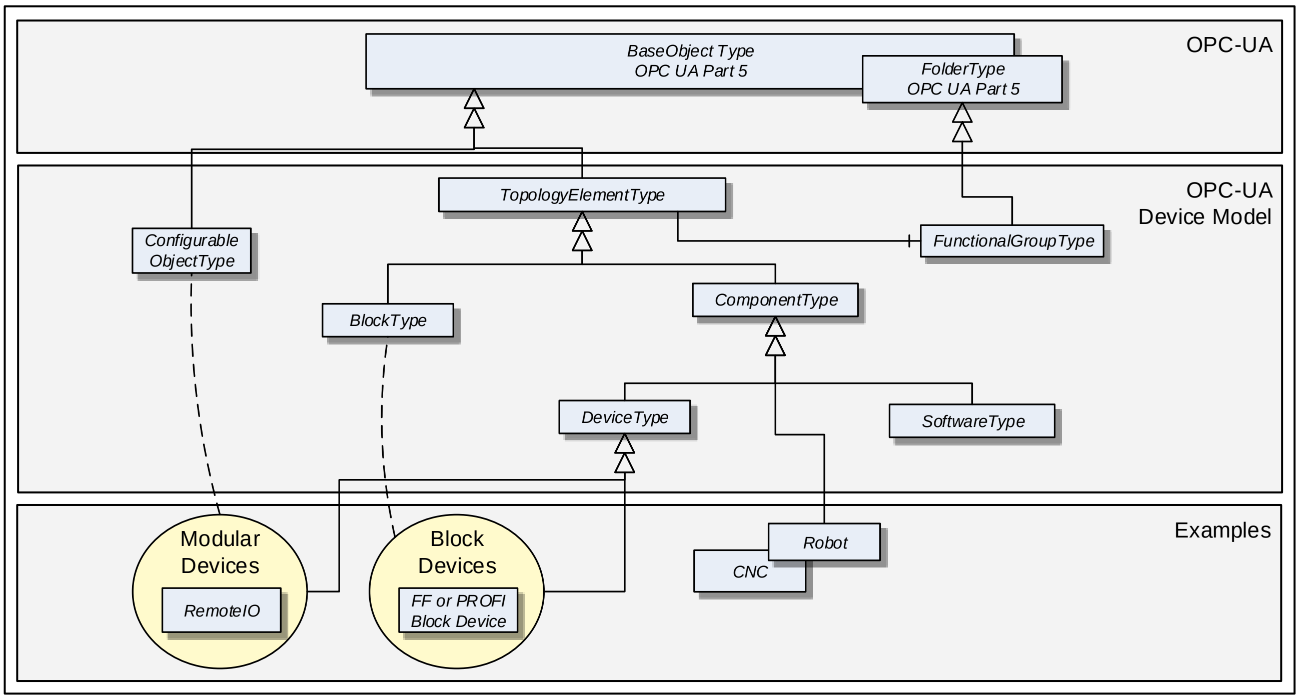

Figure 4 – Device model overview

The boxes in this drawing show the *ObjectTypes* used in this specification as well as some elements from other specifications that help understand some modelling decisions. The upper grey box shows the OPC UA core *ObjectTypes* from which the *TopologyElementType* is derived. The grey box in the second level shows the main *ObjectTypes* that the device model introduces. The components of those *ObjectTypes* are illustrated only in an abstract way in this overall picture.

The grey box in the third level shows real-world examples as they will be used in products and plants. In general, such subtypes are defined by other organizations.

The *TopologyElementType* is the base *ObjectType* for elements in a device topology. Its most essential aspect is the functional grouping concept.

The *ComponentType ObjectType* provides a generic definition for a *Device* or parts of a *Device* where parts include mechanics and/or software. *DeviceType* is commonly used to represent field *Devices*.

Modular Devices are introduced to support subdevices and *Block Devices* to support *Blocks*. *Blocks* are typically used by field communication foundations as means to organise the functionality within a *Device*. Specific types of *Blocks* will therefore be specified by these foundations.

The *ConfigurableObjectType* is used as a general means to create modular topology units. If needed an instance of this type will be added to the head object of the modular unit. Modular *Devices*, for example, will use this *ObjectType* to organise their modules. Block-oriented *Devices* use it to expose and organise their *Blocks*.

5.2 Usage guidelines

Annex C describes guidelines for the usage of the device model as base for creating companion specifications as well as guidelines on how to combine different aspects of the same device – defined in different companion specifications - in one OPC UA application.

5.3 TopologyElementType

This *ObjectType* defines a generic model for elements in a device or component topology. Among others, it introduces *FunctionalGroups*, *ParameterSet*, and *MethodSet*. Figure 5 shows the *TopologyElementType*. It is formally defined in Table 8.

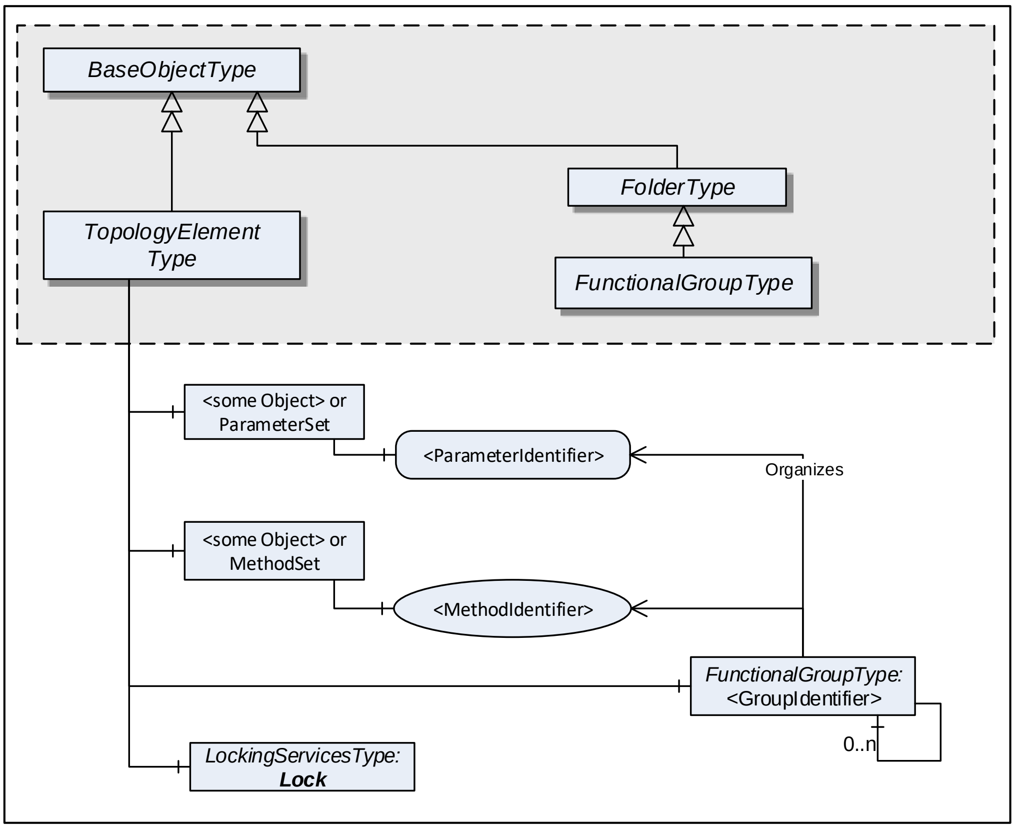
BaseObjectType
FolderType
TopologyElement Type
FunctionalGroupType
<some Object> or ParameterSet
<ParameterIdentifier>
Organizes
<some Object> or MethodSet
<MethodIdentifier>
FunctionalGroupType: <GroupIdentifier>
0..n
LockingServicesType: Lock

Figure 5 – Components of the TopologyElementType

Table 8 – TopologyElementType definition

Attribute	**Value**				
BrowseName	TopologyElementType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in OPC 10000-5					
HasSubtype	ObjectType	ComponentType	Defined in 5.6		
HasSubtype	ObjectType	BlockType	Defined in 5.11		
HasSubtype	ObjectType	ConnectionPointType	Defined in 6.4		
HasComponent	Object	<GroupIdentifier>		FunctionalGroupType	OptionalPlaceholder
HasComponent	Object	Identification		FunctionalGroupType	Optional
HasComponent	Object	Lock		LockingServicesType	Optional
HasComponent	Object	ParameterSet		BaseObjectType	Optional
HasComponent	Object	MethodSet		BaseObjectType	Optional

The *TopologyElementType* is abstract. There will be no instances of a *TopologyElementType* itself, but there will be instances of subtypes of this type. In this specification, the term *TopologyElement* generically refers to an instance of any *ObjectType* derived from the *TopologyElementType*.

FunctionalGroups are an essential aspect introduced by the *TopologyElementType*. *FunctionalGroups* are used to structure *Nodes* like *Properties*, *Parameters* and *Methods* according to their application such as configuration, diagnostics, asset management, condition monitoring and others.

FunctionalGroups are specified in 5.4.

A *FunctionalGroup* called **Identification** can be used to organise identification information of this *TopologyElement* (see 5.4.2). Identification information typically includes the *Properties* defined by the *VendorNameplate* or *TagNameplate Interfaces* and additional application specific information.

TopologyElements may also support *LockingServices* (defined in 8.3.3).

ParameterSet and *MethodSet* are defined as standard containers for systems that have a flat list of Parameters or Methods with unique names. In such cases, the *Parameters* are components of the “ParameterSet” as a flat list of *Parameters*. The *Methods* are kept the same way in the “MethodSet”. The “*ParameterSet*” *Object* is formally defined in Table 9.

Table 9 – ParameterSet definition

Attribute	**Value**			
BrowseName	ParameterSet			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**ModellingRule**
HasTypeDefinition	ObjectType	BaseObjectType		
HasComponent	Variable	<ParameterIdentifier>	BaseDataVariableType	MandatoryPlaceholder

The “*MethodSet*” *Object* is formally defined in Table 10.

Table 10 – MethodSet definition

Attribute	**Value**			
BrowseName	MethodSet			
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**ModellingRule**
HasTypeDefinition	ObjectType	BaseObjectType		
HasComponent	Method	<MethodIdentifier>		MandatoryPlaceholder

5.4 FunctionalGroupType

5.4.1 Model

This subtype of the OPC UA *FolderType* is used to structure *Nodes* like *Properties*, *Parameters* and *Methods* according to their application (e.g. maintenance, diagnostics, condition monitoring). *Organizes References* should be used when the elements are components in other parts of the *TopologyElement* that the *FunctionalGroup* belongs to. This includes *Properties*, *Variables*, and *Methods* of the *TopologyElement* or in *Objects* that are components of the *TopologyElement* either directly or via a subcomponent. The same *Property*, *Parameter* or *Method* might be useful in different application scenarios and therefore referenced from more than one *FunctionalGroup*.

FunctionalGroups can be nested.

FunctionalGroups can directly be instantiated. In this case, the *BrowseName* of a *FunctionalGroup* should indicate its purpose. A list of recommended *BrowseNames* is in 5.4.2.

Figure 6 shows the *FunctionalGroupType* components. It is formally defined in Table 11.

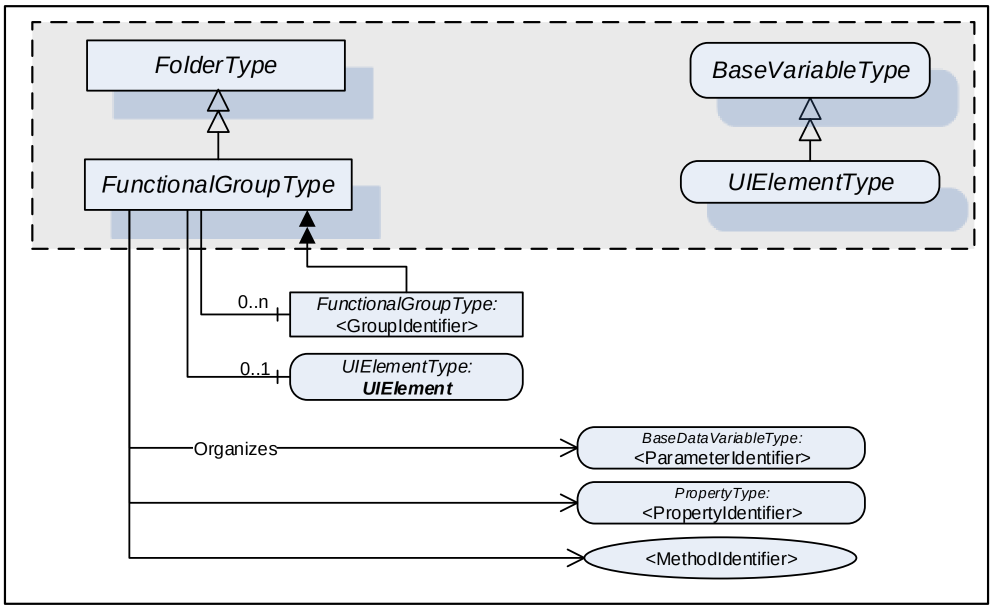

Figure 6 – FunctionalGroupType

Table 11 – FunctionalGroupType definition

Attribute	**Value**				
BrowseName	FunctionalGroupType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *FolderType* defined in OPC 10000-5					
HasComponent	Object	<GroupIdentifier>		FunctionalGroupType	OptionalPlaceholder
HasComponent	Variable	UIElement	BaseDataType	UIElementType	Optional

All *BrowseNames* for *Nodes* referenced by a *FunctionalGroup* with an *Organizes Reference* shall be unique.

The Organizes *References* may be present only at the instance, not the type. Depending on the current state of the *TopologyElement* the *Server* may decide to hide or unhide certain *FunctionalGroup*s or (part of) their *References*. If a *FunctionalGroup* may be hidden on an instance the *TypeDefinition* shall use an appropriate *ModellingRule* like "Optional".

If desirable, *Nodes* can be also children of *FunctionalGroups*. If such *Nodes* are defined, it is recommended to define a subtype of the *FunctionalGroupType*.

UIElement is the user interface element for this *FunctionalGroup*. See 5.4.3 for the definition of *UIElements*.

Examples in Annex B.1 illustrate the use of *FunctionalGroups*.

5.4.2 Recommended FunctionalGroup BrowseNames

Table 12 includes a list of *FunctionalGroups* with name and purpose. If *Servers* expose a *FunctionalGroup* that corresponds to the described purpose, they should use the recommended *BrowseName* with the Namespace of this specification.

Table 12 – Recommended FunctionalGroup BrowseNames

BrowseName	Purpose
Configuration	*Parameters* representing the configuration items of the *TopologyElement*. If the *CurrentWrite* bit is set in the *AccessLevel Attribute* they can be modified by *Clients*.
Tuning	*Parameters* and *Methods* to optimize the behavior of the *TopologyElement*.
Maintenance	*Parameters* and *Methods* useful for maintenance operations.
Diagnostics	*Parameters* and *Methods* for diagnostics.
Statistics	*Parameters* and *Methods* for statistics.
Status	*Parameters* which describe the general health of the *TopologyElement*. This can include diagnostic *Parameters*.
Operational	*Parameters* and *Methods* useful for during normal operation, like process data.
Identification	The *Properties* of the *VendorNameplate Interface*, like Manufacturer, SerialNumber or *Properties* of the TagNameplate will usually be sufficient as identification. If other *Parameters* or even *Methods* are required, all elements needed shall be organised in a *FunctionalGroup* called **Identification**. See Annex B.1 for an example.

5.4.3 UIElement Type

Servers can expose *UIElements* providing user interfaces in the context of their *FunctionalGroup* container. *Clients* can load such a user interface and display it on the *Client* side. The hierarchy of *FunctionalGroups* represents the tree of user interface elements.

The *UIElementType* is abstract and is mainly used as filter when browsing a *FunctionalGroup*. Only subtypes can be used for instances. No concrete *UIElements* are defined in this specification. FDI (Field Device Integration, see IEC 62769) specifies two concrete subtypes

- UIDs (UI Descriptions), descriptive user interface elements, and
- UIPs (UI Plug-Ins), programmed user interface elements.

The *UIElementType* is specified in Table 13.

Table 13 – UIElementType definition

Attribute	**Value**				
BrowseName	UIElementType				
IsAbstract	True				
DataType	BaseDataType				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseDataVariableType defined in OPC 10000-5.					

The *Value* attribute of the *UIElement* contains the user interface element. Subtypes have to define the *DataType* (e.g. *XmlElement* or *ByteString*).

5.5 Interfaces

5.5.1 Overview

This clause describes *Interfaces* with specific functionality that may be applied to multiple types at arbitrary positions in the type hierarchy.

Interfaces are defined in OPC 10001-7.

Figure 7 shows the *Interfaces* described in this specification.

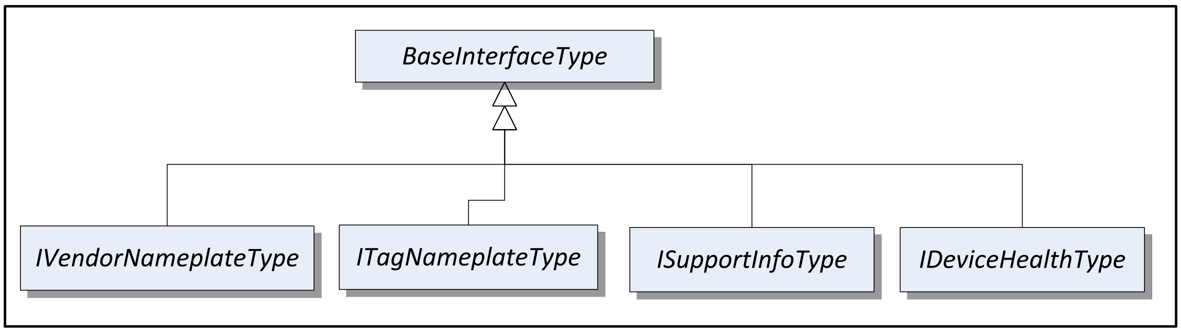

Figure 7 – Overview of Interfaces for Devices and Device components

5.5.2 VendorNameplate Interface

IVendorNameplateType includes *Properties* that are commonly used to describe a *TopologyElement* from a manufacturer point of view. They can be used as part of the identification. The *Values* of these *Properties* are typically provided by the component vendor.

The *VendorNameplate Interface* is illustrated in Figure 8 and formally defined in Table 14.

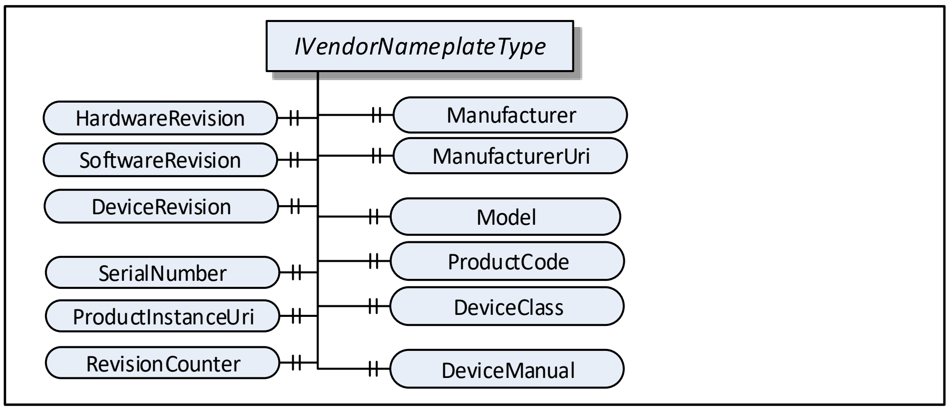

Figure 8 – VendorNameplate Interface

Table 14 – IVendorNameplateType definition

Attribute	**Value**				
BrowseName	IVendorNameplateType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseInterfaceType* defined in OPC 10001-7					
Product-specific Properties					
HasProperty	Variable	Manufacturer	LocalizedText	PropertyType	Optional
HasProperty	Variable	ManufacturerUri	String	PropertyType	Optional
HasProperty	Variable	Model	LocalizedText	PropertyType	Optional
HasProperty	Variable	ProductCode	String	PropertyType	Optional
HasProperty	Variable	HardwareRevision	String	PropertyType	Optional
HasProperty	Variable	SoftwareRevision	String	PropertyType	Optional
HasProperty	Variable	DeviceRevision	String	PropertyType	Optional
HasProperty	Variable	DeviceManual	String	PropertyType	Optional
HasProperty	Variable	DeviceClass	String	PropertyType	Optional
Product instance-specific Properties					
HasProperty	Variable	SerialNumber	String	PropertyType	Optional
HasProperty	Variable	ProductInstanceUri	String	PropertyType	Optional
HasProperty	Variable	RevisionCounter	Int32	PropertyType	Optional

Product type specific *Properties*:

Manufacturer provides the name of the company that manufactured the *TopologyElement*. *ManufacturerUri* provides a unique identifier for this company. This identifier should be a fully qualified domain name; however, it may be a GUID or similar construct that ensures global uniqueness.

Model provides the name of the product.

ProductCode provides a unique combination of numbers and letters used to identify the product. It may be the order information displayed on type shields or in ERP systems.

HardwareRevision provides the revision level of the hardware of a *TopologyElement*.

SoftwareRevision provides the version or revision level of the software component, the software/firmware of a hardware component, or the software/firmware of the *Device*.

DeviceRevision provides the overall revision level of a hardware component or the *Device*. As an example, this *Property* can be used in ERP systems together with the *ProductCode Property*.

DeviceManual allows specifying an address of the user manual for a *TopologyElement*. It may be a pathname in the file system or a URL (Web address).

DeviceClass indicates in which domain or for what purpose a certain *ComponentType* is used. Examples are “ProgrammableController”, “RemoteIO”, and “TemperatureSensor”. This standard does not predefine any *DeviceClass* names. More specific standards that utilize this *Interface* will likely introduce such classifications (e.g. IEC 62769, [OPC 30000 - PLCopen], or [OPC 10020 - ADI]).

Product instance specific *Properties*:

SerialNumber is a unique production number of the manufacturer of the *TopologyElement*. This is often stamped on the outside of a physical component and may be used for traceability and warranty purposes.

ProductInstanceUri is a globally unique resource identifier of the manufacturer of the *TopologyElement*. This is often stamped on the outside of a physical component and may be used for traceability and warranty purposes. The maximum length is 255 characters. The syntax of the *ProductInstanceUri* is: <ManufacturerUri>/<any string>. The manufacturer must ensure that the value of the field <any string> is unique among all instances using the same *ManufacturerUri*.

Examples: “some-company.com/5ff40f78-9210-494f-8206-c2c082f0609c”, “some-company.com/snr-16273849” or “some-company.com/model-xyz/snr-16273849”.

RevisionCounter is an incremental counter indicating the number of times the configuration data within a *TopologyElement* has been modified. An example would be a temperature sensor where the change of the unit would increment the *RevisionCounter* but a change of the measurement value would not affect the *RevisionCounter.*

Companion specifications may specify additional semantics for the contents of these *Properties*.

Table 15 specifies the mapping of these *Properties* to the International Registration Data Identifiers (IRDI) defined in ISO/ICE 11179-6. They should be used if a *Server* wants to expose a dictionary reference as defined in OPC 10001-5.

Table 15 – VendorNameplate Mapping to IRDIs

Property	**IRDI**
Manufacturer	0112/2///61987#ABA565 - manufacturer
ManufacturerUri	-
Model	0112/2///61987#ABA567 - name of product
SerialNumber	0112/2///61987#ABA951 - serial number
HardwareRevision	0112/2///61987#ABA926 - hardware version
SoftwareRevision	0112/2///61987#ABA601 - software version
DeviceRevision	-
RevisionCounter	-
ProductCode	0112/2///61987#ABA300 – code of product
ProductInstanceUri	-
DeviceManual	-
DeviceClass	0112/2///61987#ABA566 - type of product

5.5.3 TagNameplate Interface

ITagNameplateType includes *Properties* that are commonly used to describe a *TopologyElement* from a user point of view.

The *TagNameplate Interface* is illustrated in Figure 9 and formally defined in Table 16.

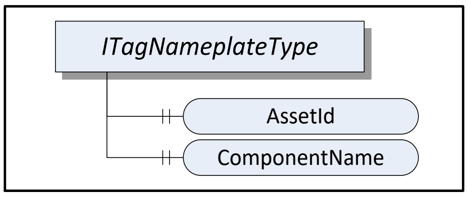

Figure 9 – TagNameplate Interface

Table 16 – ITagNameplateType definition

Attribute	**Value**				
BrowseName	ITagNameplateType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseInterfaceType* defined in OPC 10001-7					
HasProperty	Variable	AssetId	String	PropertyType	Optional
HasProperty	Variable	ComponentName	LocalizedText	PropertyType	Optional

AssetId is a user writable alphanumeric character sequence uniquely identifying a component. The ID is provided by the integrator or user of the device. It contains typically an identifier in a branch, use case or user specific naming scheme. This could be for example a reference to an electric scheme.

ComponentName is a user writable name provided by the integrator or user of the component.

Table 17 specifies the mapping of these *Properties* to the International Registration Data Identifiers (IRDI) defined in ISO/ICE 11179-6. They should be used if a *Server* wants to expose a dictionary reference as defined in OPC 10001-5.

Table 17 – TagNameplate Mapping to IRDIs

Property	IRDI
AssetId	0112/2///61987#ABA038 - identification code of device
ComponentName	0112/2///61987#ABA251 - designation of device

5.5.4 DeviceHealth Interface

The *DeviceHealth Interface* includes *Properties* and *Alarms* that are commonly used to expose the health status of a *Device*. It is illustrated in *Figure* 10 and formally defined in Table 18.

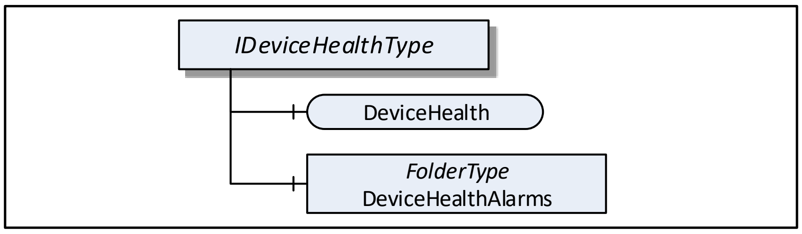

Figure 10 – DeviceHealth Interface

Table 18 – IDeviceHealthType definition

Attribute	Value				
BrowseName	IDeviceHealthType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseInterfaceType* defined in OPC 10001-7					
HasComponent	Variable	DeviceHealth	DeviceHealth Enumeration	BaseDataVariableType	Optional
HasComponent	Object	DeviceHealthAlarms		FolderType	Optional

DeviceHealth indicates the status as defined by NAMUR Recommendation NE107. *Clients* can read or monitor this *Variable* to determine the device condition.

The *DeviceHealthEnumeration DataType* is an enumeration that defines the device condition. Its values are defined in Table 19.

Table 19 – DeviceHealthEnumeration values

Value	Description
NORMAL_0	The *Device* functions normally.
FAILURE_1	Malfunction of the *Device* or any of its peripherals. Typically caused device-internal or is process related.
CHECK_FUNCTION_2	Functional checks are currently performed. Examples: change of configuration, local operation, substitute value entered.
OFF_SPEC_3	"Off-spec" means that the *Device* is operating outside its specified range (e.g. measuring or temperature range) or that internal diagnoses indicate deviations from measured or set values due to internal problems in the *Device* or process characteristics.
MAINTENANCE_REQUIRED_4	Although the output signal is valid, the wear reserve is nearly exhausted or a function will soon be restricted due to operational conditions e.g. build-up of deposits

DeviceHealthAlarms shall be used for instances of the DeviceHealth Alarm Types specified in 5.12.

5.5.5 SupportInfo Interface

5.5.5.1 General

The *SupportInfo Interface* defines a number of additional data that a commonly exposed for *Devices* and their components. These include mainly images, documents, or protocol-specific data. The various types of information is organised into different folders. Each information element is represented by a read-only *Variable*. The information can be retrieved by reading the *Variable* value.

Figure 11 Illustrates the *SupportInfo Interface.* It is formally defined in Table 20.

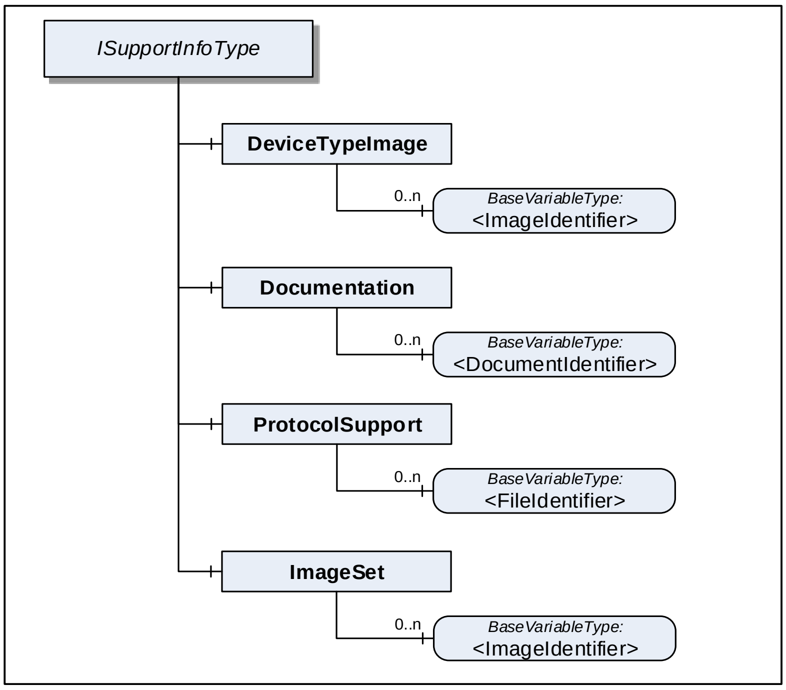

Figure 11 –Support information Interface

Table 20 – ISupportInfoType definition

Attribute	Value				
BrowseName	ISupportInfoType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseInterfaceType* defined in OPC 10001-7					
HasComponent	Object	DeviceTypeImage		FolderType	Optional
HasComponent	Object	Documentation		FolderType	Optional
HasComponent	Object	ProtocolSupport		FolderType	Optional
HasComponent	Object	ImageSet		FolderType	Optional

Clients need to be aware that the contents that these *Variables* represent may be large. Reading large values with a single Read operation may not be possible due to configured limits in either the *Client* or the *Server* stack. The default maximum size for an array of bytes is 1 megabyte. It is recommended that *Clients* use the IndexRange in the OPC UA Read Service (see OPC 10000-4) to read these *Variables* in chunks, for example, one-megabyte chunks. It is up to the *Client* whether it starts without an index and repeats with an IndexRange only after an error or whether it always uses an IndexRange.

The different types of support information are specified in 5.5.5.2 to 5.5.5.5.

5.5.5.2 Device Type Image

Pictures can be exposed as *Variables* organised in the *DeviceTypeImage* folder. There may be multiple images of different resolutions. Each image is a separate *Variable*. The "*DeviceTypeImage*" *Folder* is formally defined in Table 21.

Table 21 – DeviceTypeImage definition

Attribute	Value				
BrowseName	DeviceTypeImage				
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**DataType**	**ModellingRule**
HasTypeDefinition	ObjectType	FolderType (defined in OPC 10000-5.)			
HasComponent	Variable	<ImageIdentifier>	BaseDataVariableType	Image	MandatoryPlaceholder

All images are transferred as a *ByteString*. The *DataType* of the *Variable* specifies the image format. OPC UA defines BMP, GIF, JPG and PNG (see OPC 10000-3).

5.5.5.3 Documentation

Documents are exposed as *Variables* organized in the *Documentation* folder. In most cases they will represent a product manual, which can exist as a set of individual documents. The "*Documentation*" *Folder* is formally defined in Table 22.

Table 22 – Documentation definition

Attribute	Value				
BrowseName	Documentation				
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**DataType**	**ModellingRule**
HasTypeDefinition	ObjectType	FolderType (defined in OPC 10000-5.)			
HasComponent	Variable	<DocumentIdentifier>	BaseDataVariableType	ByteString	MandatoryPlaceholder

All documents are transferred as a *ByteString*. The *BrowseName* of each *Variable* will consist of the filename including the extension that can be used to identify the document type. Typical extensions are ".pdf" or ".txt".

5.5.5.4 Protocol Support Files

Protocol support files are exposed as *Variables* organised in the *ProtocolSupport* folder. They may represent various types of information as defined by a protocol. Examples are a GSD or a CFF file. The "*ProtocolSupport*" *Folder* is formally defined in Table 23.

Table 23 – ProtocolSupport definition

Attribute	**Value**				
BrowseName	ProtocolSupport				
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**DataType**	**ModellingRule**
HasTypeDefinition	ObjectType	FolderType (defined in OPC 10000-5)			
HasComponent	Variable	<ProtocolSupportIdentifier>	BaseDataVariableType	ByteString	MandatoryPlaceholder

All protocol support files are transferred as a *ByteString*. The *BrowseName* of each *Variable* shall consist of the complete filename including the extension that can be used to identify the type of information.

5.5.5.5 Images

Images that are used within *UIElements* are exposed as separate *Variables* rather than embedding them in the element. All image *Variables* will be aggregated by the *ImageSet* folder. The *UIElement* shall specify an image by its name that is also the *BrowseName* of the image *Variable*. *Clients* can cache images so they don't have to be transferred more than once. The "*ImageSet*" *Folder* is formally defined in Table 24.

Table 24 – ImageSet definition

Attribute	**Value**				
BrowseName	ImageSet				
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**DataType**	**ModellingRule**
HasTypeDefinition	ObjectType	FolderType (defined in OPC 10000-5.)			
HasComponent	Variable	<ImageIdentifier>	BaseDataVariableType	Image	MandatoryPlaceholder

The *DataType* of the *Variable* specifies the image format. OPC UA defines BMP, GIF, JPG and PNG (see OPC 10000-3).

5.6 ComponentType

Compared to *DeviceType* the *ComponentType* is more universal. It includes the same components but does not mandate any *Properties*. This makes it usable for representation of a *Device* or parts of a *Device*. Parts include both mechanical and software parts.

The *ComponentType* applies the Vendor*Nameplate* and the *TagNameplate Interface*. Figure 12 Illustrates the *ComponentType.* It is formally defined in Table 25.

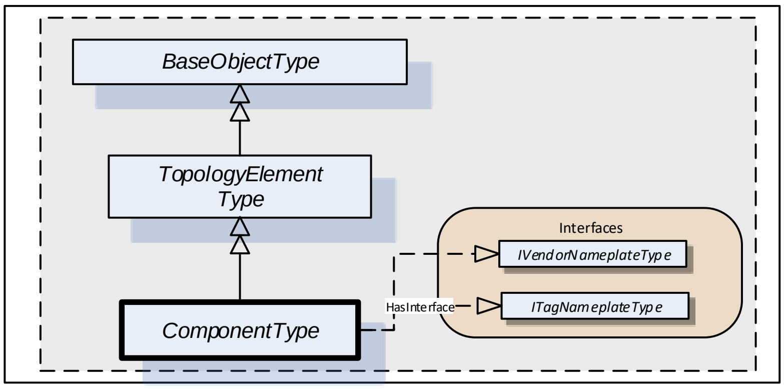

Figure 12 – ComponentType

Table 25 – ComponentType definition

Attribute	**Value**				
BrowseName	ComponentType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *TopologyElementType* defined in 5.3.					
HasSubtype	ObjectType	DeviceType	Defined in 5.7.		
HasSubtype	ObjectType	SoftwareType	Defined in 5.8.		
HasInterface	ObjectType	IVendorNameplateType	Defined in 5.5.2.		
HasInterface	ObjectType	ITagNameplateType	Defined in 5.5.3.		
Applied from *IVendorNameplateType*					
HasProperty	Variable	Manufacturer	LocalizedText	PropertyType	Optional
HasProperty	Variable	ManufacturerUri	String	PropertyType	Optional
HasProperty	Variable	Model	LocalizedText	PropertyType	Optional
HasProperty	Variable	ProductCode	String	PropertyType	Optional
HasProperty	Variable	HardwareRevision	String	PropertyType	Optional
HasProperty	Variable	SoftwareRevision	String	PropertyType	Optional
HasProperty	Variable	DeviceRevision	String	PropertyType	Optional
HasProperty	Variable	DeviceManual	String	PropertyType	Optional
HasProperty	Variable	DeviceClass	String	PropertyType	Optional
HasProperty	Variable	SerialNumber	String	PropertyType	Optional
HasProperty	Variable	ProductInstanceUri	String	PropertyType	Optional
HasProperty	Variable	RevisionCounter	Int32	PropertyType	Optional
Applied from *ITagNameplateType*					
HasProperty	Variable	AssetId	String	PropertyType	Optional
HasProperty	Variable	ComponentName	LocalizedText	PropertyType	Optional

The *ComponentType* is abstract. *DeviceType* and *SoftwareType* are subtypes of *ComponentType*. There will be no instances of a *ComponentType* itself, only of concrete subtypes.

IVendorNameplateType and its members are described in 5.5.2.

ITagNameplateType and its members are described in 5.5.3.

5.7 DeviceType

This *ObjectType* can be used to define the structure of a *Device*. Figure 13 shows the *DeviceType*. It is formally defined in Table 26.

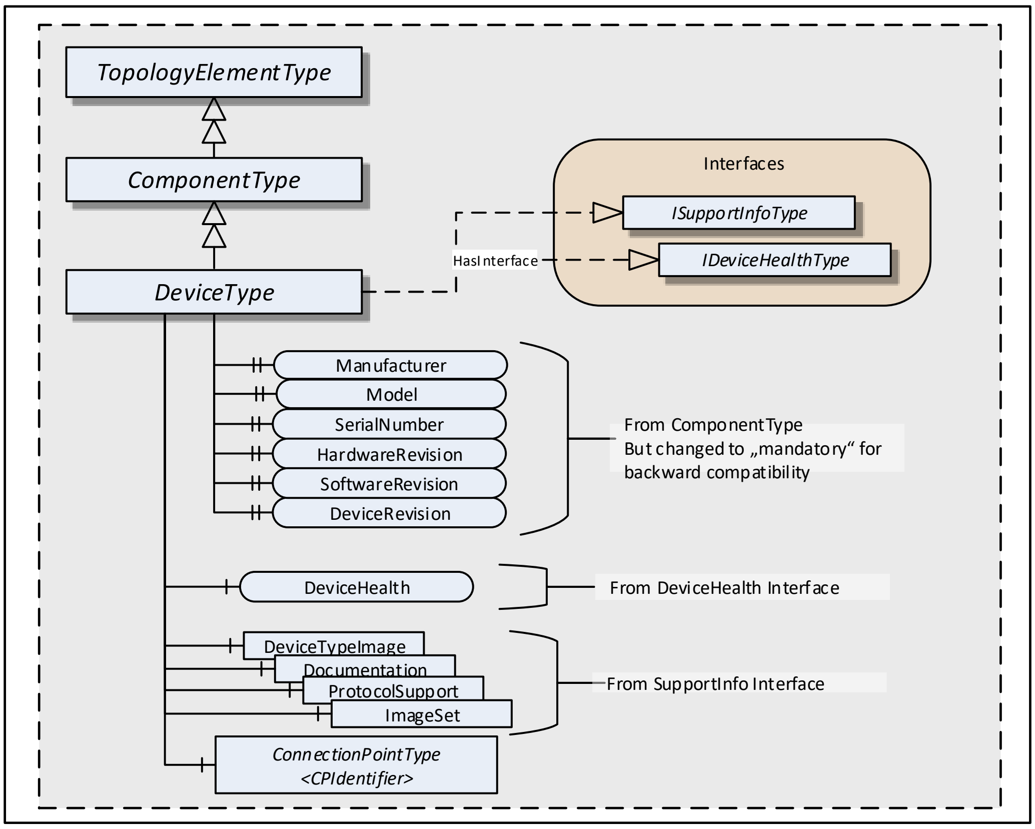

Figure 13 – DeviceType

Table 26 – DeviceType definition

Attribute	**Value**				
BrowseName	DeviceType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *ComponentType* defined in 5.6					
HasInterface	ObjectType	ISupportInfoType		Defined in 5.5.3.	
HasInterface	ObjectType	IDeviceHealthType		Defined in 5.5.3.	
HasComponent	Object	<CPIdentifier>		ConnectionPointType	OptionalPlaceholder
HasProperty	Variable	SerialNumber	String	PropertyType	Mandatory
HasProperty	Variable	RevisionCounter	Int32	PropertyType	Mandatory
HasProperty	Variable	Manufacturer	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	Model	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	DeviceManual	String	PropertyType	Mandatory
HasProperty	Variable	DeviceRevision	String	PropertyType	Mandatory
HasProperty	Variable	SoftwareRevision	String	PropertyType	Mandatory
HasProperty	Variable	HardwareRevision	String	PropertyType	Mandatory
HasProperty	Variable	DeviceClass	String	PropertyType	Optional
HasProperty	Variable	ManufacturerUri	String	PropertyType	Optional
HasProperty	Variable	ProductCode	String	PropertyType	Optional
HasProperty	Variable	ProductInstanceUri	String	PropertyType	Optional
Applied from IDeviceHealthType					
HasComponent	Variable	DeviceHealth	DeviceHealthEnumeration	BaseDataVariableType	Optional
HasComponent	Object	DeviceHealthAlarms		FolderType	Optional
Applied from ISupportInfoType					
HasComponent	Object	DeviceTypeImage		FolderType	Optional
HasComponent	Object	Documentation		FolderType	Optional
HasComponent	Object	ProtocolSupport		FolderType	Optional
HasComponent	Object	ImageSet		FolderType	Optional

DeviceType is a subtype of *ComponentType* which means it inherits all *InstanceDeclarations*.

The *DeviceType ObjectType* is abstract. There will be no instances of a *DeviceType* itself, only of concrete subtypes.

ConnectionPoints (see 6.4) represent the interface (interface card) of a *DeviceType* instance to a *Network*. Multiple *ConnectionPoints* may exist if multiple protocols and/or multiple *Communication Profiles* are supported.

The *Interfaces* and their members are described in 5.5. Some of the *Properties* inherited from the *ComponentType* are declared mandatory for backward compatibility.

Although mandatory, some of the *Properties* may not be supported for certain types of *Devices*. In this case vendors shall provide the following defaults:

• *Properties* with *DataType String*:	**empty string**
• *Properties* with *DataType LocalizedText*:	**empty text field**
• *RevisionCounter Property*:	**- 1**

Clients can ignore the *Properties* when they have these defaults.

When *Properties* are not supported, *Servers* should initialize the corresponding *Property* declaration on the *DeviceType* with the default value. Relevant *Browse Service* requests can then return a *Reference* to this *Property* on the type definition. That way, no extra *Nodes* are needed.

5.8 SoftwareType

This *ObjectType* can be used for software modules of a *Device* or a part of a *Device*. *SoftwareType* is a concrete subtype of *ComponentType* and can be used directly.

Figure 14 Illustrates the *SoftwareType.* It is formally defined in Table 27.

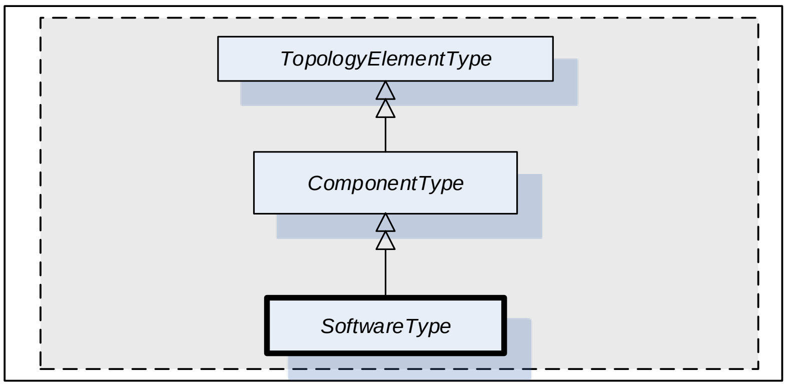

Figure 14 – SoftwareType

Table 27 – SoftwareType definition

Attribute	**Value**				
BrowseName	SoftwareType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *ComponentType* defined in 5.6.					
HasProperty	Variable	Manufacturer	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	Model	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	SoftwareRevision	String	PropertyType	Mandatory

SoftwareType is a subtype of *ComponentType* which means it inherits all *InstanceDeclarations*.

The *Properties Manufacturer*, *Model*, and *SoftwareRevision inherited from ComponentType* are declared mandatory for *SoftwareType* instances.

5.9 DeviceSet entry point

The *DeviceSet Object* is the starting point to locate *Devices*. It shall either directly or indirectly reference all instances of a subtype of *ComponentType* with a *Hierarchical Reference*. For complex *Devices* that are composed of various components that are also *Devices*, only the root instance shall be referenced from the *DeviceSet Object*. The components of such complex *Devices* shall be locatable by following *Hierarchical References* from the root instance. An example is the *Modular Device* defined in 9.4 and also illustrated in Figure 15.

Examples:

- UA *Server* represents a monolithic or modular *Device*: *DeviceSet* only contains one instance
- UA *Server* represents a host system that has access to a number of *Devices* that it manages: *DeviceSet* contains several instances that the host provides access to.
- UA *Server* represents a gateway *Device* that acts as representative for *Devices* that it has access to: *DeviceSet* contains the gateway *Device* instance and instances for the *Devices* that it represents.
- UA *Server* represents a robotic system consisting of mechanics and controls. *DeviceSet* only contains the instance for the root of the robotic system. The mechanics and controls

are represented by *ComponentType* instances which are organised as sub-components of the root instance.

Figure 15 shows the *AddressSpace* organisation with this standard entry point and examples.

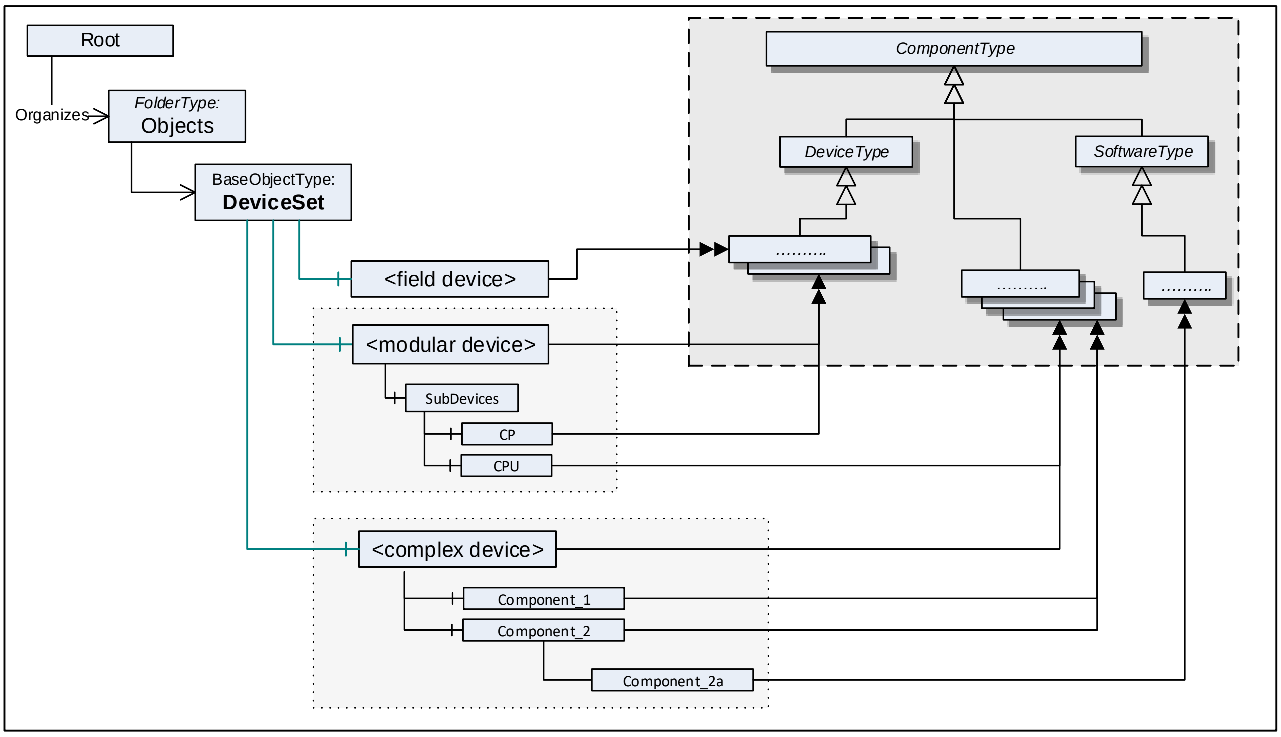

Figure 15 – Standard entry point for Devices

The *DeviceSet Node* is formally defined in Table 28.

Table 28 – DeviceSet definition

Attribute	**Value**		
BrowseName	DeviceSet		
References	**NodeClass**	**BrowseName**	**TypeDefinition**
OrganizedBy by the Objects Folder defined in OPC 10000-5			
HasTypeDefinition	ObjectType	BaseObjectType	

5.10 DeviceFeatures entry point

The *DeviceFeatures Object* can be used to organise other functional entities that are related to the *Devices* referenced by the *DeviceSet*. Companion specifications may standardize such instances and their *BrowseNames*. Figure 16 shows the *AddressSpace* organisation with this standard entry point.

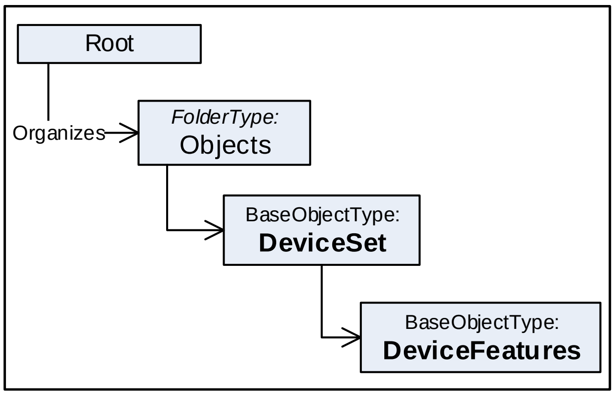

Figure 16 – Standard entry point for DeviceFeatures

The *DeviceFeatures Node* is formally defined in Table 29.

Table 29 – DeviceFeatures definition

Attribute	**Value**		
BrowseName	DeviceFeatures		
References	**NodeClass**	**BrowseName**	**TypeDefinition**
OrganizedBy by the DeviceSet Object defined in 5.9			
HasTypeDefinition	ObjectType	BaseObjectType	

5.11 BlockType

This *ObjectType* defines the structure of a *Block Object*. Figure 17 depicts the *BlockType* hierarchy. It is formally defined in Table 30.

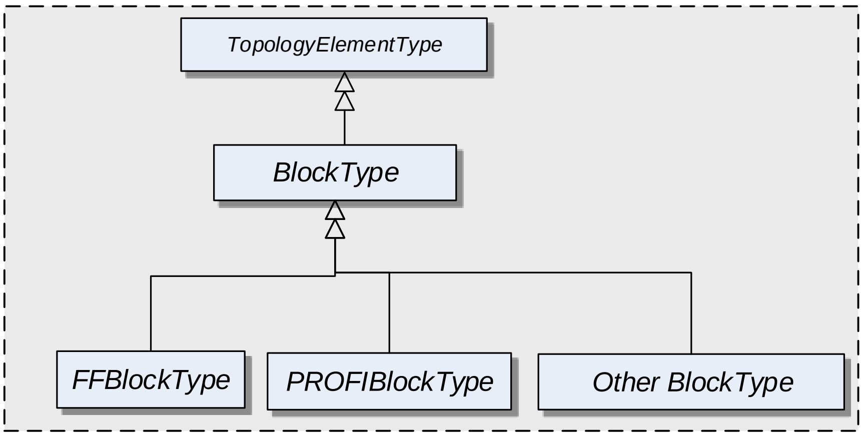

Figure 17 – BlockType hierarchy

FFBlockType and PROFIBlockType are examples. They are not further defined in this specification. It is expected that industry groups will standardize general purpose *BlockTypes*.

Table 30 – BlockType definition

Attribute	Value				
BrowseName	BlockType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *TopologyElementType* defined in 5.2					
HasProperty	Variable	RevisionCounter	Int32	PropertyType	Optional
HasProperty	Variable	ActualMode	LocalizedText	PropertyType	Optional
HasProperty	Variable	PermittedMode	LocalizedText[]	PropertyType	Optional
HasProperty	Variable	NormalMode	LocalizedText[]	PropertyType	Optional
HasProperty	Variable	TargetMode	LocalizedText[]	PropertyType	Optional

BlockType is a subtype of *TopologyElementType* and inherits the elements for *Parameters*, *Methods* and *FunctionalGroups*.

The *BlockType* is abstract. There will be no instances of a *BlockType* itself, but there will be instances of subtypes of this *Type*. In this specification, the term *Block* generically refers to an instance of any subtype of the *BlockType*.

The *RevisionCounter* is an incremental counter indicating the number of times the static data within the *Block* has been modified. A value of -1 indicates that no revision information is available.

The following *Properties* refer to the *Block Mode* (e.g. “Manual”, “Out of Service”).

The *ActualMode Property* reflects the current mode of operation.

The *PermittedMode* defines the modes of operation that are allowed for the *Block* based on application requirements.

The *NormalMode* is the mode the *Block* should be set to during normal operating conditions. Depending on the *Block* configuration, multiple modes may exist.

The *TargetMode* indicates the mode of operation that is desired for the *Block*. Depending on the *Block* configuration, multiple modes may exist.

5.12 DeviceHealth Alarm Types

5.12.1 General

The DeviceHealth Property defined in 5.5.4 provides a basic way to expose the health state of a device based on NAMUR NE 107.

This section defines *AlarmTypes* that can be used to indicate an abnormal device condition together with diagnostic information text as defined by NAMUR NE 107 as well as additional manufacturer specific information.

Figure 18 informally describes the *AlarmTypes* for DeviceHealth.

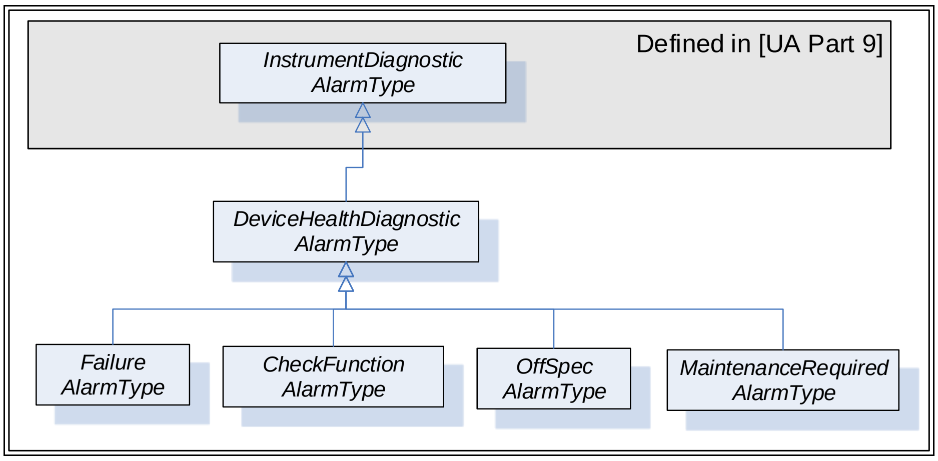
Defined in [UA Part 9]
InstrumentDiagnostic AlarmType
DeviceHealthDiagnostic AlarmType
Failure AlarmType
CheckFunction AlarmType
OffSpec AlarmType
MaintenanceRequired AlarmType

Figure 18 – Device Health Alarm type hierarchy

5.12.2 DeviceHealthDiagnosticAlarmType

The *DeviceHealthDiagnosticAlarmType* is a specialization of the *InstrumentDiagnosticAlarmType* intended to represent abnormal device conditions as defined by NAMUR NE 107. This type can be used in filters for monitored items. Only subtypes of this type will be used in actual implementations. The *Alarm* becomes active when the device condition is abnormal. It is formally defined in Table 31.

Table 31 – DeviceHealthDiagnosticAlarmType definition

Attribute	Value				
BrowseName	DeviceHealthDiagnosticAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the InstrumentDiagnosticAlarmType defined in OPC 10000-9.					
HasSubtype	ObjectType	FailureAlarmType	Defined in clause 5.12.3		
HasSubtype	ObjectType	CheckFunctionAlarmType	Defined in clause 5.12.4		
HasSubtype	ObjectType	OffSpecAlarmType	Defined in clause 5.12.5		
HasSubtype	ObjectType	MaintenanceRequiredAlarmType	Defined in clause 5.12.6		

Conditions of subtypes of *DeviceHealthDiagnosticAlarmType* become active when the device enters the corresponding abnormal state.

The *Message* field in the *Event* notification shall be used for additional information associated with the health status (e.g. the possible cause of the abnormal state and suggested actions to return to normal).

A Device may be in more than one abnormal state at a time in which case multiple *Conditions* will be active.

5.12.3 FailureAlarmType

The *FailureAlarmType* is formally defined in Table 32.

Table 32 – FailureAlarmType definition

Attribute	Value				
BrowseName	FailureAlarmType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DeviceHealthDiagnosticAlarmType defined in 5.12.2.					

5.12.4 CheckFunctionAlarmType

The *CheckFunctionAlarmType* is formally defined in Table 33.

Table 33 – CheckFunctionAlarmType definition

Attribute	**Value**				
BrowseName	CheckFunctionAlarmType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DeviceHealthDiagnosticAlarmType defined in 5.12.2.					

5.12.5 OffSpecAlarmType

The *OffSpecAlarmType* is formally defined in Table 34.

Table 34 – OffSpecAlarmType definition

Attribute	**Value**				
BrowseName	OffSpecAlarmType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DeviceHealthDiagnosticAlarmType defined in 5.12.2.					

5.12.6 MaintenanceRequiredAlarmType

The *MaintenanceRequiredAlarmType* is formally defined in Table 35.

Table 35 – MaintenanceRequiredAlarmType definition

Attribute	**Value**				
BrowseName	MaintenanceRequiredAlarmType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DeviceHealthDiagnosticAlarmType defined in 5.12.2.					

6 Device communication model

6.1 General

Clause 6 introduces *References*, the *ProtocolType*, and basic *TopologyElementTypes* needed to create a communication topology. The types for this model are illustrated in Figure 19.

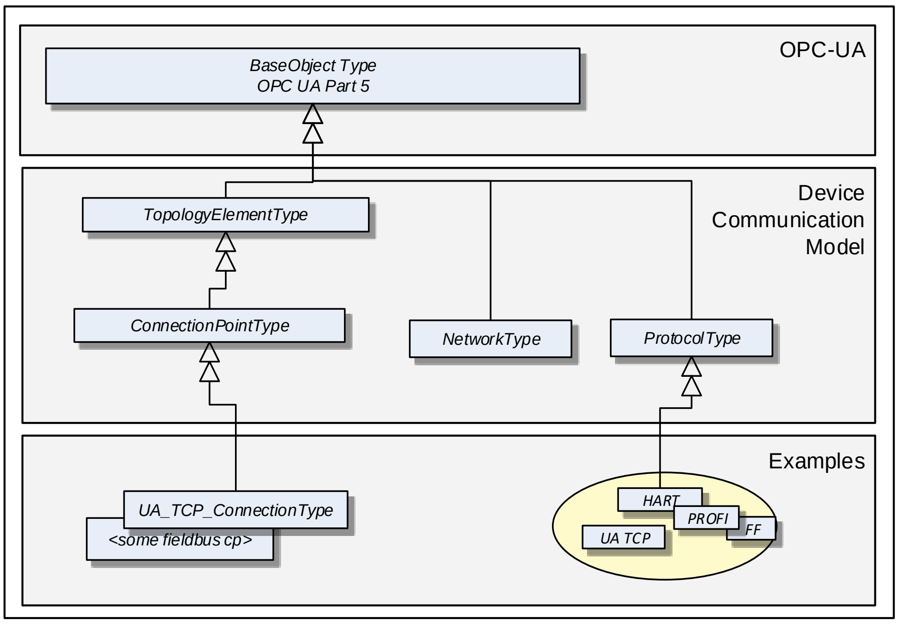

Figure 19 – Device communication model overview

A *ProtocolType ObjectType* represents a specific communication protocol (e.g. *FieldBus*) implemented by a certain *TopologyElement*. Examples are shown in Figure 21.

The *ConnectionPointType* represents the logical interface of a *Device* to a *Network*.

A *Network* is the logical representation of wired and wireless technologies.

Figure 20 provides an overall example.

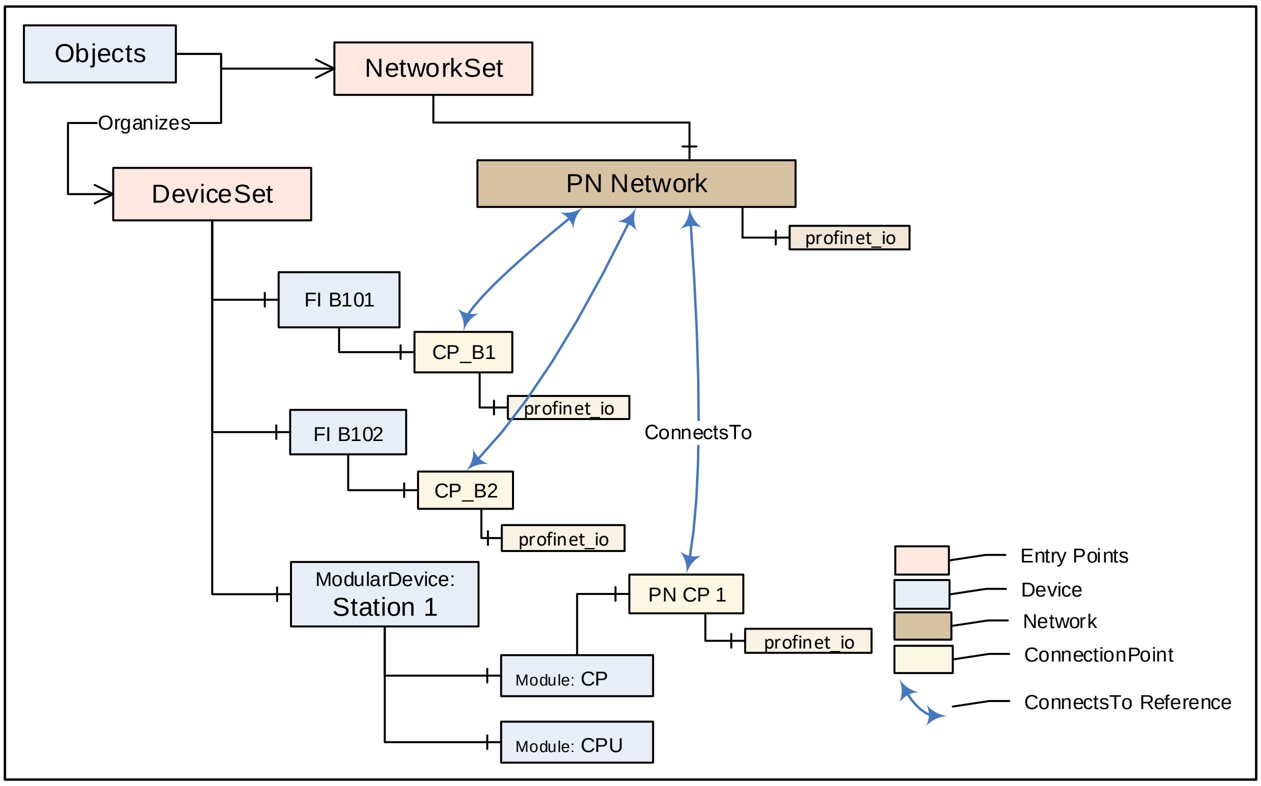

Figure 20 – Example of a communication topology

6.2 ProtocolType

The *ProtocolType ObjectType* and its subtypes are used to specify a specific communication (e.g. *FieldBus*) protocol that is supported by a *Device* (respectively by its *ConnectionPoint*) or *Network*.

The *BrowseName* of each instance of a *ProtocolType* shall define the *Communication Profile* (see Figure 21).

Figure 21 shows the *ProtocolType* including some specific types and instances that represent *Communication Profiles* of that type. It is formally defined in Table 36.

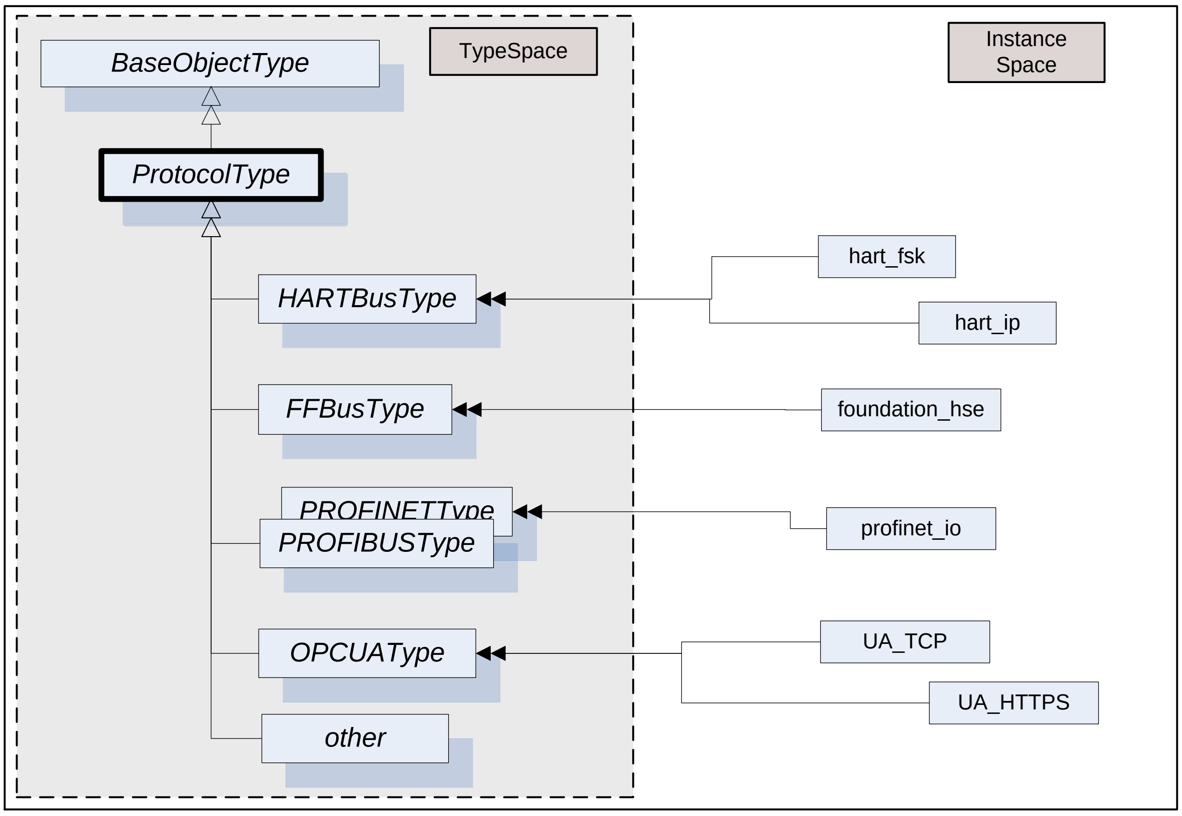

Figure 21 – Example of a ProtocolType hierarchy with instances that represent specific communication profiles

Table 36 – ProtocolType definition

Attribute	**Value**				
BrowseName	ProtocolType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in OPC 10000-5					

6.3 Network

A *Network* is the logical representation of wired and wireless technologies and represents the communication means for *Devices* that are connected to it. A *Network* instance is qualified by its *Communication Profile* components.

Figure 22 shows the type hierarchy and the *NetworkType* components. It is formally defined in Table 37.

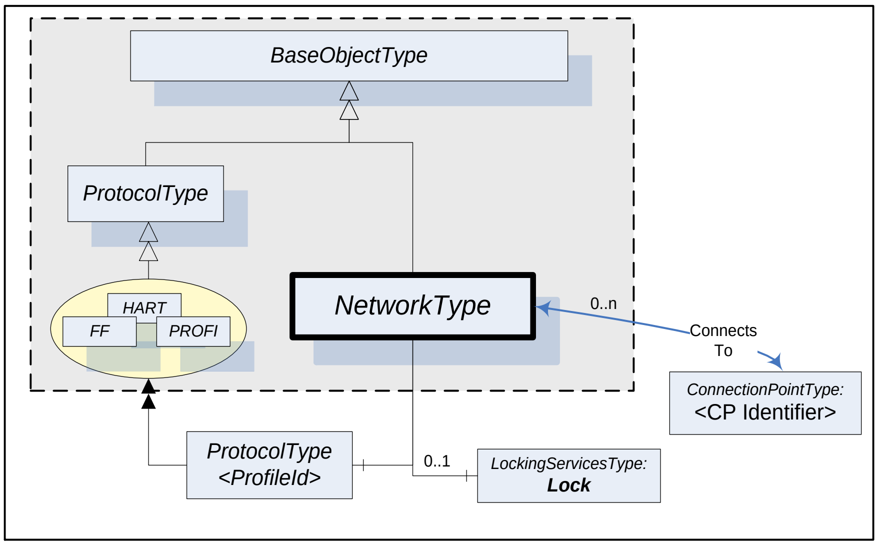

Figure 22 – NetworkType

Table 37 – NetworkType definition

Attribute	**Value**				
BrowseName	NetworkType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseObjectType defined in OPC 10000-5.					
HasComponent	Object	<ProfileIdentifier>		ProtocolType	MandatoryPlaceholder
ConnectsTo	Object	<CPIdentifier>		ConnectionPointType	OptionalPlaceholder
HasComponent	Object	Lock		LockingServicesType	Optional

The <ProfileIdentifier> specifies the *Protocol* and *Communication Profile* that this *Network* is used for.

<CPIdentifier> (referenced by a *ConnectsTo Reference*) references the *ConnectionPoint*(s) that have been configured for this *Network*. All *ConnectionPoint*s shall adhere to the same *Protocol* as the *Network*. See also Figure 25 for a usage example. They represent the protocol-specific access points for the connected *Devices*.

In addition, *Networks* may also support *LockingServices* (defined in 8.3).

6.4 ConnectionPoint

This *ObjectType* represents the logical interface of a *Device* to a *Network*. A specific subtype shall be defined for each protocol. Figure 23 shows the *ConnectionPointType* including some specific types.

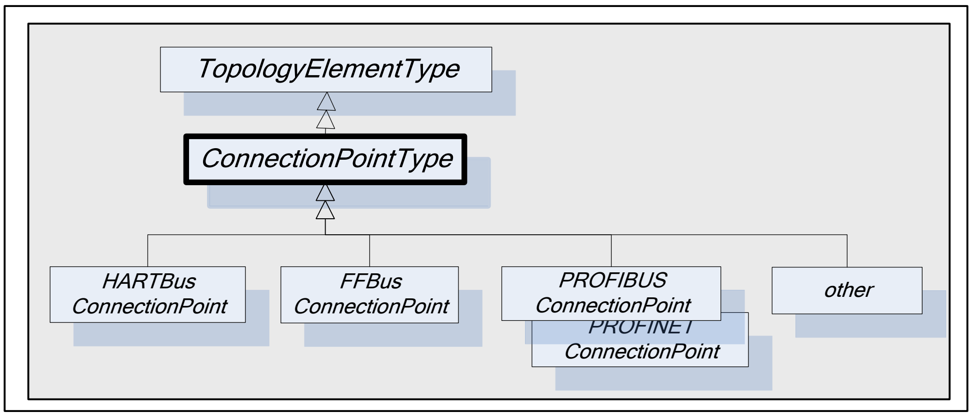

Figure 23 – Example of ConnectionPointType hierarchy

A *Device* can have more than one such interface to the same or to different *Networks*. Different interfaces usually exist for different protocols. Figure 24 shows the *ConnectionPointType* components. It is formally defined in Table 38.

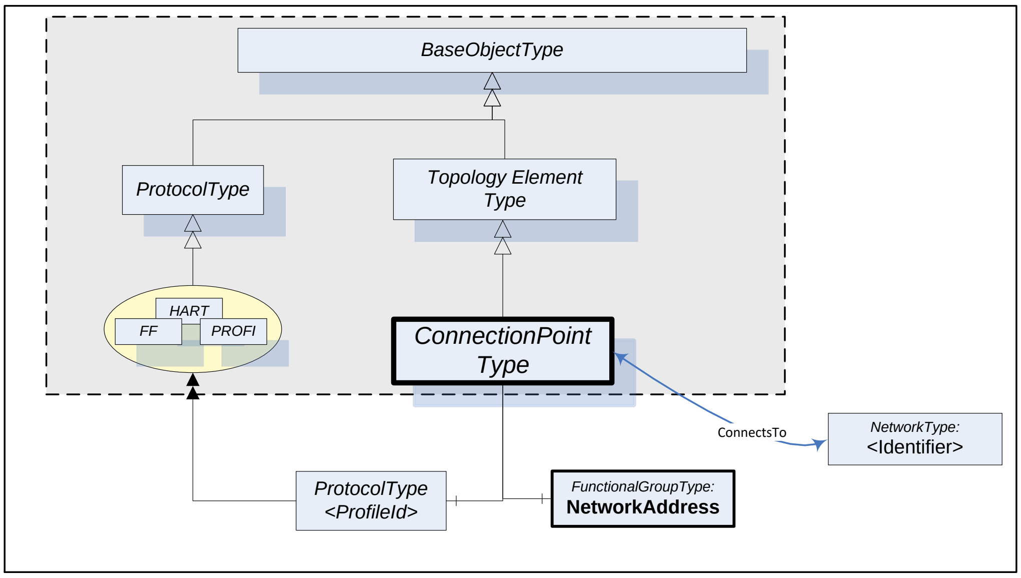

Figure 24 – ConnectionPointType

Table 38 – ConnectionPointType definition

Attribute	**Value**				
BrowseName	ConnectionPointType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the TopologyElementType defined in 5.2.					
HasComponent	Object	NetworkAddress		FunctionalGroupType	Mandatory
HasComponent	Object	<ProfileIdentifier>		ProtocolType	MandatoryPlaceholder
ConnectsTo	Object	<NetworkIdentifier>		NetworkType	OptionalPlaceholder

ConnectionPoints are components of a *Device*, represented by a subtype of *ComponentType*. To allow navigation from a *Network* to the connected *Devices*, the *ConnectionPoints* shall have the inverse *Reference (ComponentOf)* to the *Device*.

ConnectionPoints have *Properties* and other components that they inherit from the *TopologyElementType*.

The *NetworkAddress FunctionalGroup* includes all *Parameters* needed to specify the protocol-specific address information of the connected *Device*. These *Parameters* may be components of the *NetworkAddress FunctionalGroup*, of the *ParameterSet*, or another *Object*.

<ProfileIdentifier> identifies the *Communication Profile* that this *ConnectionPoint* supports. *ProtocolType* and *Communication Profile* are defined in 6.2. It implies that this *ConnectionPoint* can be used to connect *Networks* and *Devices* of the same *Communication Profile*.

ConnectionPoints are between a *Network* and a *Device*. The location in the topology is configured by means of the *ConnectsTo ReferenceType*. Figure 25 illustrates some usage models.

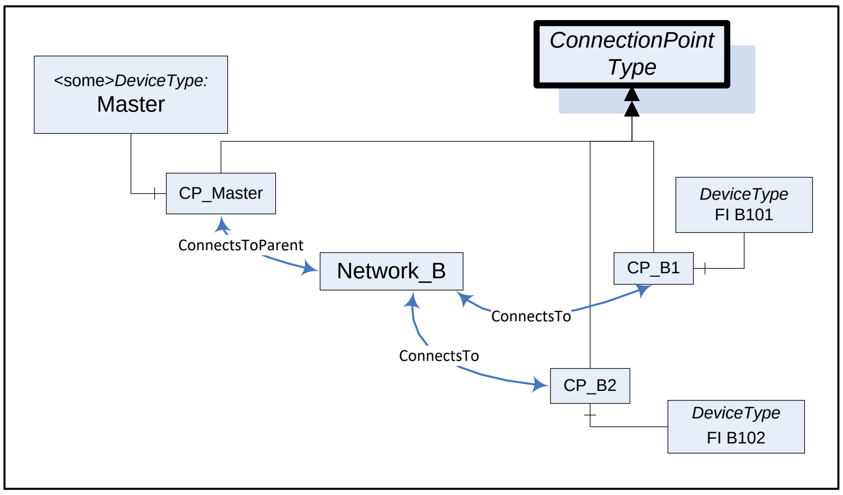

Figure 25 – ConnectionPoint usage

6.5 ConnectsTo and ConnectsToParent ReferenceTypes

The *ConnectsTo ReferenceType* is a concrete *ReferenceType* used to indicate that source and target Node have a topological connection. It is both hierarchical and symmetric, because this is natural for this *Reference*. The *ConnectsTo Reference* exists between a *Network* and the connected *Devices* (or their *ConnectionPoint*, respectively). Browsing a *Network* returns the connected *Devices*; browsing from a *Device*, one can follow the *ConnectsTo Reference* from the *Device's ConnectionPoint* to the *Network*.

The *ConnectsToParent ReferenceType* is a concrete *ReferenceType* used to define the parent (i.e. the communication *Device*) of a *Network*. It is a subtype of The *ConnectsTo ReferenceType*.

The two *ReferenceTypes* are illustrated in Figure 26.

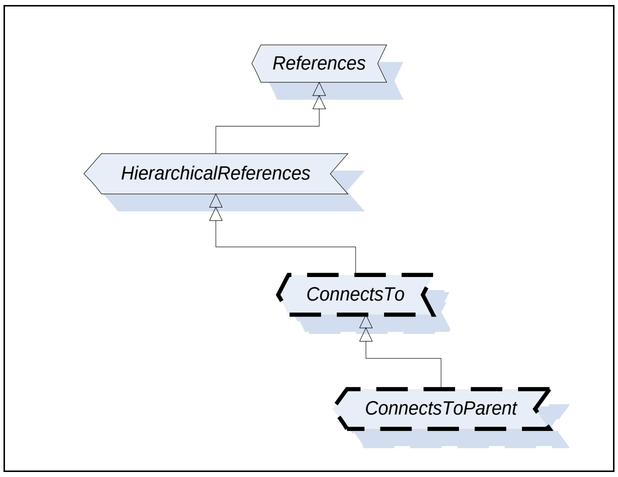

Figure 26 – Type Hierarchy for ConnectsTo and ConnectsToParent References

The representation in the *AddressSpace* is specified in Table 39 and Table 40.

Table 39 – ConnectsTo ReferenceType

Attributes	**Value**		
BrowseName	ConnectsTo		
Symmetric	True		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HierarchicalReferences ReferenceType defined in OPC 10000-5.			

Table 40 – ConnectsToParent ReferenceType

Attributes	**Value**		
BrowseName	ConnectsToParent		
Symmetric	True		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of ConnectsTo ReferenceType			

Figure 27 illustrates how this *Reference* can be used to express topological relationships and parental relationships. In this example two *Devices* are connected; the module DPcomm is the communication *Device* for the *Network*.

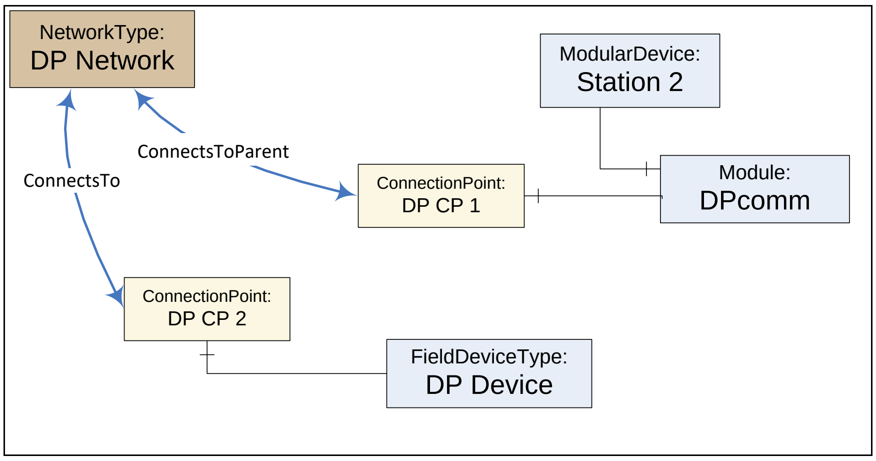

Figure 27 – Example with ConnectsTo and ConnectsToParent References

6.6 NetworkSet Object

All *Networks* shall be components of the **NetworkSet** *Object*.

The **NetworkSet** *Node* is formally defined in Table 41.

Table 41 – NetworkSet definition

Attribute	**Value**		
BrowseName	NetworkSet		
References	**NodeClass**	**BrowseName**	**TypeDefinition**
OrganizedBy by the Objects Folder defined in OPC 10000-5			
HasTypeDefinition	ObjectType	BaseObjectType	

7 Device integration host model

7.1 General

A *Device Integration Host* is a *Server* that manages integration of multiple *Devices* in an automation system and provides *Clients* with access to information about *Devices* regardless of where the information is stored, for example, in the *Device* itself or in a data store. The *Device* communication is internal to the host and may be based on field-specific protocols.

The *Information Model* specifies the entities that can be accessed in a *Device Integration Host*. This standard does not define how these elements are instantiated. The host may use network scanning services, the OPC UA *Node Management Services* or proprietary configuration tools.

One of the main tasks of the *Information Model* is to reflect the topology of the automation system. Therefore it represents the *Devices* of the automation system as well as the connecting communication networks including their properties, relationships, and the operations that can be performed on them.

Figure 28 and Figure 29 illustrate an example configuration and the configured topology as it will appear in the *Server AddressSpace* (details left out).

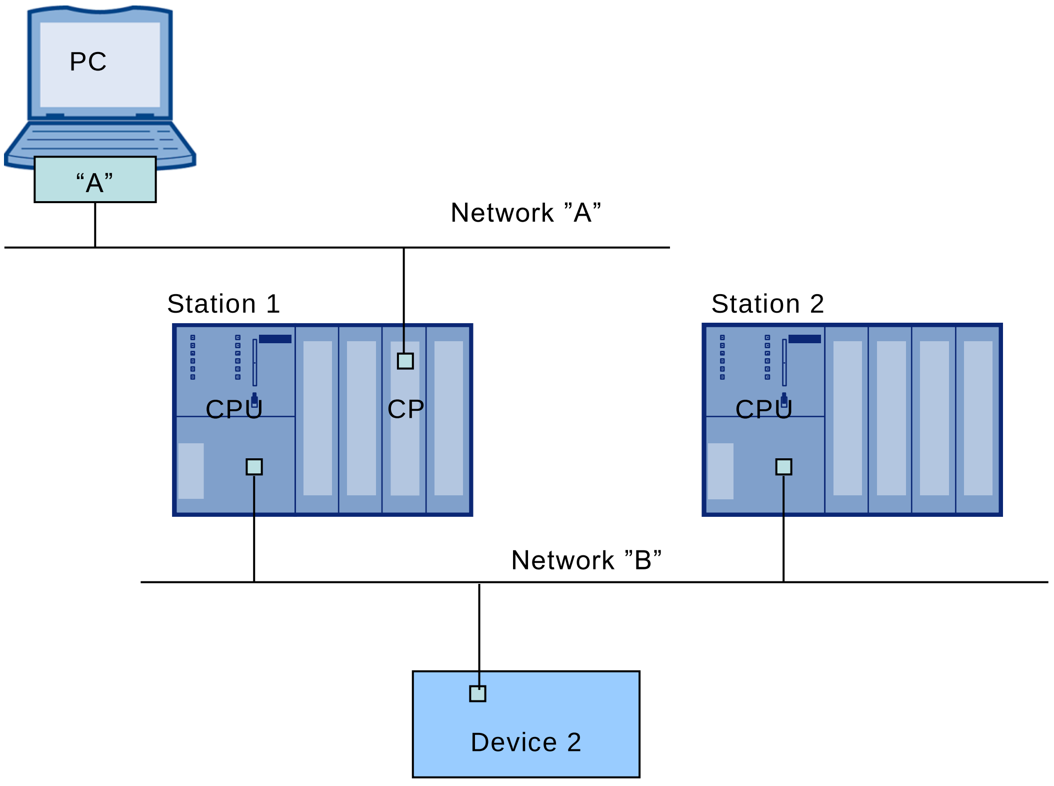

Figure 28 – Example of an automation system

The PC in Figure 28 represents the *Server* (the *Device Integration Host*). The *Server* communicates with *Devices* connected to *Network* "A" via native communication, and it communicates with *Devices* connected to *Network* "B" via nested communication.

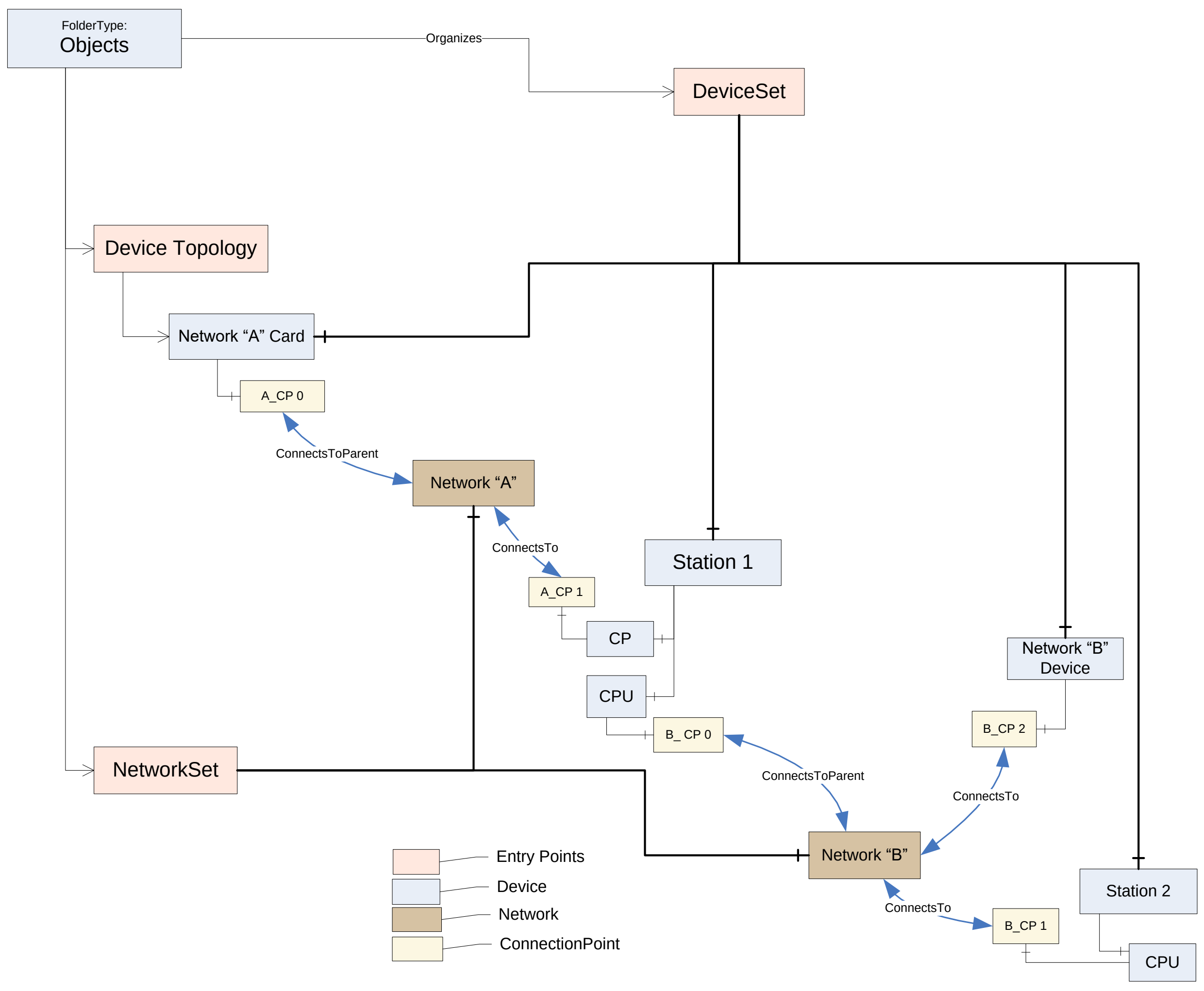

Figure 29 – Example of a Device topology

Coloured boxes are used to recognize the various types of information.

Entry points assure common behaviour across different implementations:

- *DeviceTopology*: Starting node for the topology configuration. See 7.2.
- *DeviceSet*: See 5.9.
- *NetworkSet*: See 6.6.

7.2 DeviceTopology Object

The *Device Topology* reflects the communication topology of the *Devices*. It includes *Devices* and the Networks. The entry point **DeviceTopology** is the starting point within the *AddressSpace* and is used to organise the communication *Devices* for the top level *Networks* that provide access to all instances that constitute the *Device Topology* ((sub-)networks, devices and communication elements).

The *DeviceTopology* node is formally defined in Table 42.

Table 42 – DeviceTopology definition

Attribute	Value			
BrowseName	DeviceTopology			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**
OrganizedBy by the Objects Folder defined in OPC 10000-5				
HasTypeDefinition	ObjectType	BaseObjectType	Defined in OPC 10000-5.	
HasProperty	Variable	OnlineAccess	Boolean	PropertyType

OnlineAccess provides a hint of whether the *Server* is currently able to communicate to *Devices* in the topology. “False” means that no communication is available.

7.3 Online/Offline

7.3.1 General

Management of the *Device Topology* is a configuration task, i.e., the elements in the topology (*Devices*, *Networks*, and *Connection Points*) are usually configured “offline” and – at a later time – will be validated against their physical representative in a real network.

To support explicit access to either the online or the offline information, each element may be represented by two instances that are schematically identical, i.e., there exist component *Objects*, *FunctionalGroups*, and so on. A *Reference* connects online and offline representations and allows to navigate between them.

This is illustrated in Figure 30.

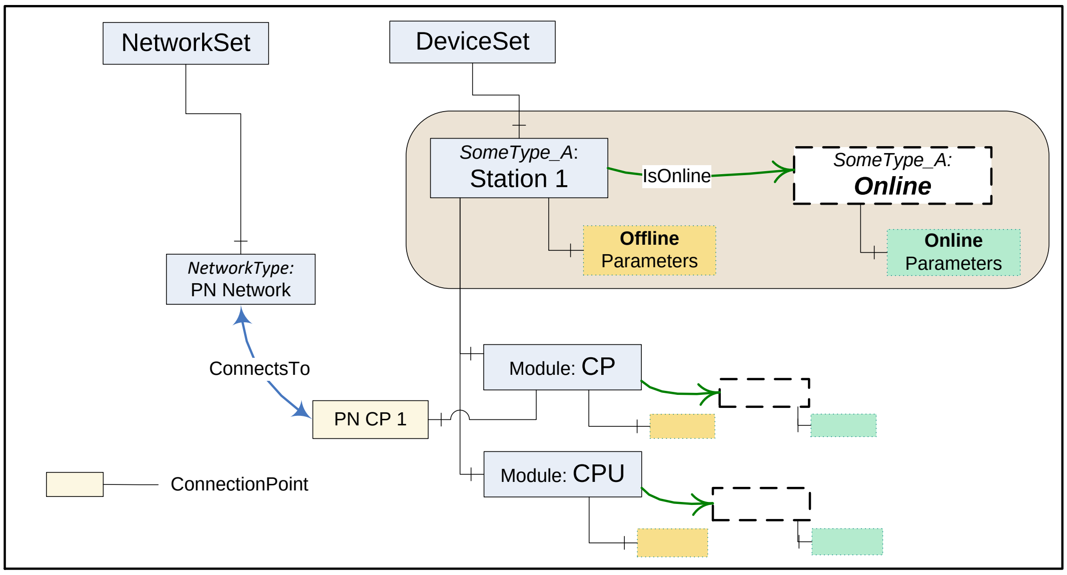

Figure 30 – Online component for access to Device data

If Online/Offline is supported, the main (leading) instance represents the offline information. Its *HasTypeDefinition Reference* points to the concrete configured or identified *ObjectType*. All *Parameters* of this instance represent offline data points and reading or writing them will typically result in configuration database access. *Properties* will also represent offline information.

A *Device* can be engineered through the offline instance without online access.

The online data for a topology element are kept in an associated *Object* with the *BrowseName* **Online** as illustrated in Figure 30. The **Online** *Object* is referenced via an *IsOnline Reference*. It is always of the same *ObjectType* as the offline instance.

The online *Parameter Nodes* reflect values in a physical element (typically a *Device*), i.e., reading or writing to a *Parameter* value will then result in a communication request to this element. When elements are not connected, reading or writing to the online Parameter will return a proper status code (Bad_NotConnected).

The transfer of information (*Parameters*) between offline nodes and the physical device in correct order is supported through *TransferToDevice*, *TransferFromDevice* together with *FetchTransferResultData*. These *Methods* are exposed by means of an *AddIn* instance of *TransferServicesType* described in 8.2.2.

Both offline and online are created and driven by the same *ObjectType*. According to their usability, certain components (*Parameters*, *Methods*, and *FunctionalGroups*) may exist only in either the online or the offline element.

A *Parameter* in the offline *ParameterSet* and its corresponding counterpart in the online *ParameterSet* shall have the same *BrowseName*. Their *NodeIds* need to be different, though, since this is the identifier passed by the *Client* in read/write requests.

The **Identification** *FunctionalGroup* organises *Parameters* that help identify a topology element. *Clients* can compare the values of these *Parameters* in the online and the offline instance to detect mismatches between the configuration data and the currently connected element.

7.3.2 IsOnline ReferenceType

The *IsOnline ReferenceType* is a concrete *ReferenceType* used to bind the offline representation of a *Device* to the online representation. The source and target *Node* of *References* of this type shall be an instance of the same subtype of a *ComponentType*. Each *Device* shall be the source of at most one *Reference* of type *IsOnline*.

The *IsOnline ReferenceType* is illustrated in Figure 31. Its representation in the *AddressSpace* is specified in Table 43.

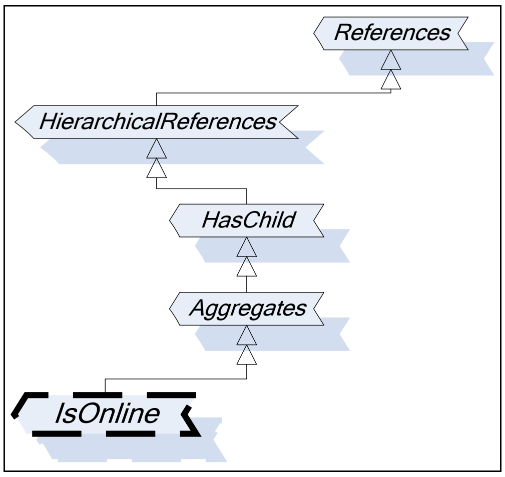

Figure 31 – Type hierarchy for IsOnline Reference

Table 43 – IsOnline ReferenceType

Attributes	**Value**		
BrowseName	IsOnline		
InverseName	OnlineOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of Aggregates ReferenceType defined in OPC 10000-5.			

8 AddIn Capabilities

8.1 Overview

OPC 10001-7 specifies the *AddIn* model as a mechanism to add dedicated features to an *Object* or *ObjectType* using aggregation.

The following features are based on this *AddIn* model.

8.2 Offline-Online data transfer

8.2.1 Definition

The transfer of information (*Parameters*) between offline nodes and the physical device is supported through OPC UA *Methods*. These *Methods* are built on device specific knowledge and functionality.

The transfer is usually terminated if an error occurs for any of the *Parameters*. No automatic retry will be conducted by the *Server*. However, whenever possible after a failure, the *Server* should bring the *Device* back into a functional state. The *Client* has to retry by calling the transfer *Method* again.

The transfer may involve thousands of *Parameters* so that it can take a long time (up to minutes), and with a result that may be too large for a single response. Therefore, the initiation of the transfer and the collection of result data are performed with separate *Methods*.

The *Device* shall have been locked by the *Client* prior to invoking these *Methods* (see 8.3).

8.2.2 TransferServices Type

The *TransferServicesType* provides the *Methods* needed to transfer data to and from the online *Device*. Figure 32 shows the *TransferServicesType* definition. It is formally defined in Table 44.

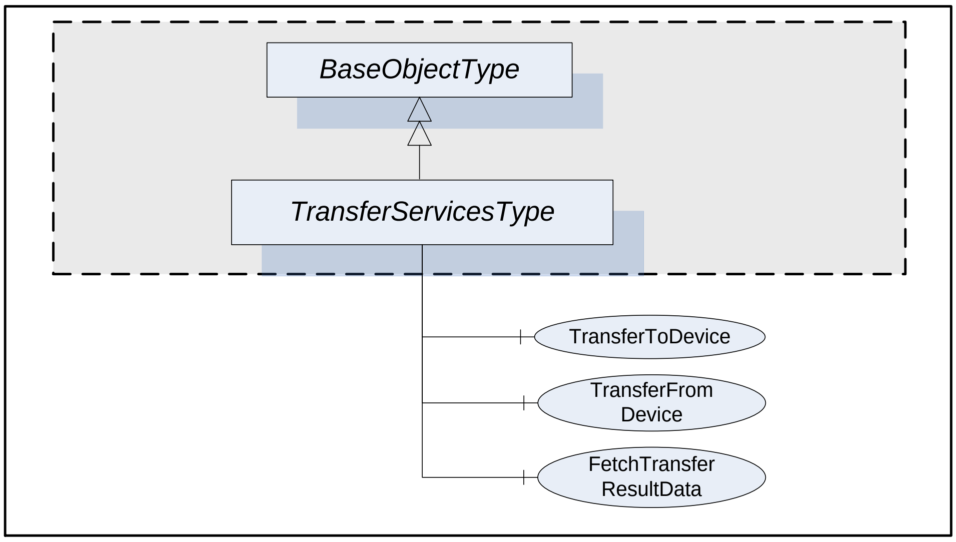

Figure 32 – TransferServicesType

Table 44 – TransferServicesType definition

Attribute	Value				
BrowseName	TransferServicesType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseObjectType defined in OPC 10000-5					
HasComponent	Method	TransferToDevice			Mandatory
HasComponent	Method	TransferFromDevice			Mandatory
HasComponent	Method	FetchTransferResultData			Mandatory

The *StatusCode Bad_MethodInvalid* shall be returned from the Call *Service* for *Objects* where locking is not supported. *Bad_UserAccessDenied* shall be returned if the *Client User* does not have the permission to call the *Methods*.

8.2.3 TransferServices Object

The support of *TransferServices* for an *Object* is declared by aggregating an instance of the *TransferServicesType* as illustrated in Figure 33.

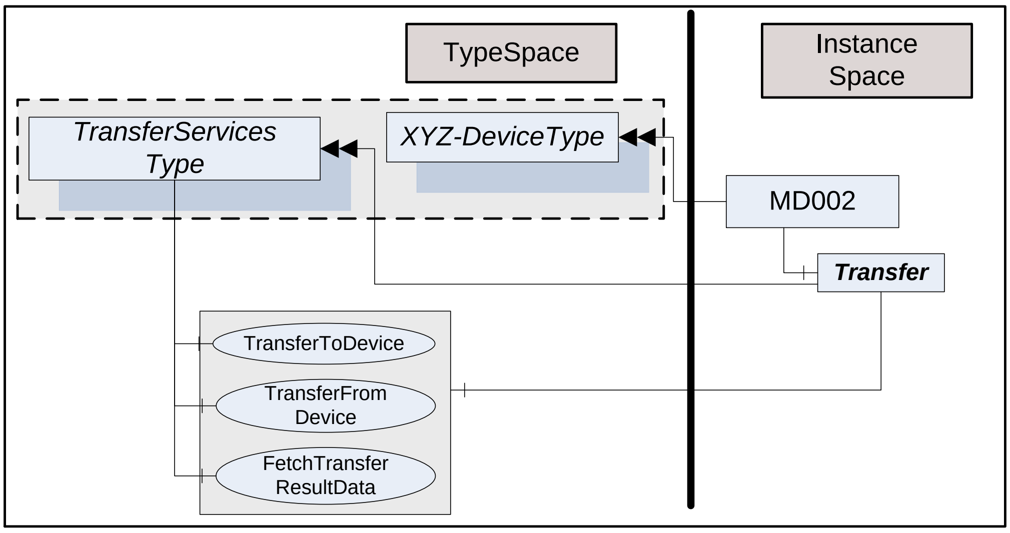

Figure 33 – TransferServices

This *Object* is used as container for the *TransferServices Methods* and shall have the *BrowseName* **Transfer**. *HasComponent* is used to reference from a *Device* to its “TransferServices” *Object*.

The *TransferServiceType* and each instance may share the same *Methods*.

8.2.4 TransferToDevice Method

TransferToDevice initiates the transfer of offline configured data (*Parameters*) to the physical device. This *Method* has no input arguments. Which *Parameters* are transferred is based on *Server*-internal knowledge.

The *Server* shall ensure integrity of the data before starting the transfer. Once the transfer has been started successfully, the *Method* returns immediately with InitTransferStatus = 0. Any status information regarding the transfer itself has to be collected using the *FetchTransferResultData Method*.

The *Server* will reset any cached value for *Nodes* in the online instance representing *Parameters* affected by the transfer. That way the cache will be re-populated from the *Device* next time they are requested.

The signature of this *Method* is specified below. Table 45 and Table 46 specify the arguments and *AddressSpace* representation, respectively.

Signature

```
TransferToDevice(

  [out] Int32               TransferID,
  [out] Int32               InitTransferStatus);
```

Table 45 – TransferToDevice Method arguments

Argument	Description
TransferID	Transfer Identifier. This ID has to be used when calling FetchTransferResultData.
InitTransferStatus	Specifies if the transfer has been initiated. 0 – OK -1 – E_NotLocked – the Device is not locked by the calling *Client* -2 – E_NotOnline – the Device is not online / cannot be accessed

Table 46 – TransferToDevice Method AddressSpace definition

Attribute	Value				
BrowseName	TransferToDevice				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

8.2.5 TransferFromDevice Method

TransferFromDevice initiates the transfer of values from the physical device to corresponding *Parameters* in the offline representation of the *Device*. This *Method* has no input arguments. Which *Parameters* are transferred is based on *Server*-internal knowledge.

Once the transfer has been started successfully, the *Method* returns immediately with InitTransferStatus = 0. Any status information regarding the transfer itself has to be collected using the *FetchTransferResultData Method*.

The signature of this *Method* is specified below. Table 47 and Table 48 specify the arguments and *AddressSpace* representation, respectively.

Signature

```
TransferFromDevice(

  [out] Int32               TransferID,
  [out] Int32               InitTransferStatus);
```

Table 47 – TransferFromDevice Method arguments

Argument	Description
TransferID	Transfer Identifier. This ID has to be used when calling FetchTransferResultData.
InitTransferStatus	Specifies if the transfer has been initiated. 0 – OK -1 – E_NotLocked – the Device is not locked by the calling *Client* -2 – E_NotOnline – the Device is not online / cannot be accessed

Table 48 – TransferFromDevice Method AddressSpace definition

Attribute	**Value**				
BrowseName	TransferFromDevice				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

8.2.6 FetchTransferResultData Method

The *TransferToDevice* and *TransferFromDevice Methods* execute asynchronously after sending a response to the *Client*. Execution status and execution results are collected during execution and can be retrieved using the *FetchTransferResultData Method*. The *TransferID* is used as identifier to retrieve the data.

The *Client* is assumed to fetch the result data in a timely manner. However, because of the asynchronous execution and the possibility of data loss due to transmission errors to the *Client*, the *Server* shall wait some time (some minutes) before deleting data that have not been acknowledged. This should be even beyond *Session* termination, i.e. *Clients* that have to re-establish a *Session* after an error may try to retrieve missing result data.

Result data will be deleted with each new transfer request for the same *Device*.

FetchTransferResultData is used to request the execution status and a set of result data. If called before the transfer is finished it will return only partial data. The amount of data returned may be further limited if it would be too large. “Too large” in this context means that the *Server* is not able to return a larger response or that the number of results to return exceeds the maximum number of results that was specified by the *Client* when calling this *Method*.

Each result returned to the *Client* is assigned a sequence number. The *Client* acknowledges that it received the result by passing the sequence number in the new call to this *Method*. The *Server* can delete the acknowledged result and will return the next result set with a new sequence number.

Clients shall not call the *Method* before the previous one returned. If it returns with an error (e.g. Bad_Timeout), the *Client* can call the *FetchTransferResultData* with a sequence number 0. In this case the *Server* will resend the last result set.

The Server will return Bad_NothingToDo in the *Method*-specific *StatusCode* of the *Call Service* if the transfer is finished and no further result data are available.

The signature of this *Method* is specified below. Table 49 and Table 50 specify the arguments and *AddressSpace* representation, respectively.

Signature

```
FetchTransferResultData(
  [in]  Int32                TransferID,
  [in]  Int32                SequenceNumber,
  [in]  Int32                MaxParameterResultsToReturn,
  [in]  Boolean              OmitGoodResults,
  [out] FetchResultType      FetchResultData);
```

Table 49 –FetchTransferResultData Method arguments

Argument	Description
TransferID	Transfer Identifier returned from *TransferToDevice* or *TransferFromDevice*.
SequenceNumber	The sequence number being acknowledged. The *Server* may delete the result set with this sequence number. “0” is used in the first call after initialising a transfer and also if the previous call of *FetchTransferResultData* failed.
MaxParameterResultsToReturn	The number of *Parameters* in *TransferResult.ParameterDefs* that the *Client* wants the *Server* to return in the response. The *Server* is allowed to further limit the response, but shall not exceed this limit. A value of 0 indicates that the *Client* is imposing no limitation.
OmitGoodResults	If TRUE, the Server will omit data for Parameters which have been correctly transferred. Note that this causes all good results to be released.
FetchResultData	Two subtypes are possible: • TransferResultError Type is returned if the transfer failed completely • TransferResultData Type is returned if the transfer was performed. Status information is returned for each transferred *Parameter*.

Table 50 – FetchTransferResultData Method AddressSpace definition

Attribute	Value				
BrowseName	FetchTransferResultData				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

The *FetchResultDataType* is an abstract type. It is the base *DataType* for concrete result types of the *FetchTransferResultData*. Its elements are defined in Table 51.

Table 51 – FetchResultDataType structure

Attribute	Value		
BrowseName	FetchResultDataType		
IsAbstract	True		
Subtype of Structure defined in OPC 10000-3			
References	**NodeClass**	**BrowseName**	**DataType**
HasSubtype	DataType	TransferResultErrorDataType	Defined in Table 52.
HasSubtype	DataType	TransferResultDataDataType	Defined in Table 53.

The *TransferResultErrorDataType* is a subtype of the *FetchResultDataType* and represents an error result. It is defined in Table 52.

Table 52 – TransferResultError DataType structure

Name	Type	Description
TransferResultError DataType	Structure	This structure is returned in case of errors. No result data are returned. Further calls with the same *TransferID* are not possible.
status	Int32	-1 – Invalid *TransferID*: The Id is unknown. Possible reason: all results have been fetched or the result may have been deleted. -2 – Transfer aborted: The transfer operation was aborted; no results exist. -3 – DeviceError: An error in the device or the communication to the *Device* occurred. “diagnostics” may contain device- or protocol-specific error information. -4 – UnknownFailure: The transfer failed. “diagnostics” may contain *Device*- or *Protocol*-specific error information.
diagnostics	DiagnosticInfo	Diagnostic information. This parameter is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request. The *DiagnosticInfo* type is defined in OPC 10000-4.

The *TransferResultData DataType* is a subtype of the *FetchResultDataType* and includes parameter-results from the transfer operation. It is defined in Table 53.

Table 53 – TransferResultData DataType structure

Name	Type	Description
TransferResultData DataType	Structure	A set of results from the transfer operation.
sequenceNumber	Int32	The sequence number of this result set.
endOfResults	Boolean	TRUE – all result data have been fetched. Additional *FetchTransferResultData* calls with the same *TransferID* will return a FetchTransferError with status=InvalidTransferID. FALSE – further result data shall be expected.
parameterDefs	structure[]	Specific value for each *Parameter* that has been transferred. If OmitGoodResults is TRUE, parameterDefs will only contain *Parameters* which have not been transferred correctly.
NodePath	QualifiedName[]	List of BrowseNames that represent the relative path from the *Device Object* to the *Parameter* following hierarchical references. The *Client* may use these names for *TranslateBrowsePathsToNodeIds* to retrieve the *Parameter NodeId* for the online or the offline representation.
statusCode	StatusCode	OPC UA *StatusCode* as defined in OPC 10000-4 and in OPC 10000-8.
diagnostics	DiagnosticInfo	Diagnostic information. This parameter is empty if diagnostics information was not requested in the request header or if no diagnostic information was encountered in processing of the request. The *DiagnosticInfo* type is defined in OPC 10000-4.

8.3 Locking

8.3.1 Overview

Locking is the means to avoid concurrent modifications to a *TopologyElement* or *Network* and their components. *Clients* shall use the locking services if they need to make a set of changes (for example, several *Write* operations and *Method* invocations) and where a consistent state is available only after all of these changes have been performed. The main purpose of locking a *TopologyElement* is avoiding concurrent modifications. The main purpose of locking a *Network* is avoiding concurrent topology changes.

A lock from one *Client* usually allows other *Clients* to view (navigate/read) the locked element. *Servers* may choose to implement an exclusive locking where other *Clients* have no access at all (e.g. in cases where even read operations require certain settings in a *TopologyElement*).

When locking a *TopologyElement*, the lock applies to the complete *TopologyElement* (including all components such as blocks or modules).

Servers may allow independent locking of component *TopologyElements*, if no lock is applied to the top-level *TopologyElement*.

If the Online/Offline model is supported (see 7.3), the lock always applies to both the online and the offline version.

When locking a *Network*, the lock applies to the *Network* and all connected *TopologyElements*. If any of the connected *TopologyElements* provides access to a sub-ordinate *Network* (like a gateway), the sub-ordinate *Network* and its connected *TopologyElements* are locked as well.

8.3.2 LockingServices Type

The *LockingServicesType* provides the *Methods* needed to lock or unlock. Figure 34 shows the *LockingServicesType* definition. It is formally defined in Table 54.

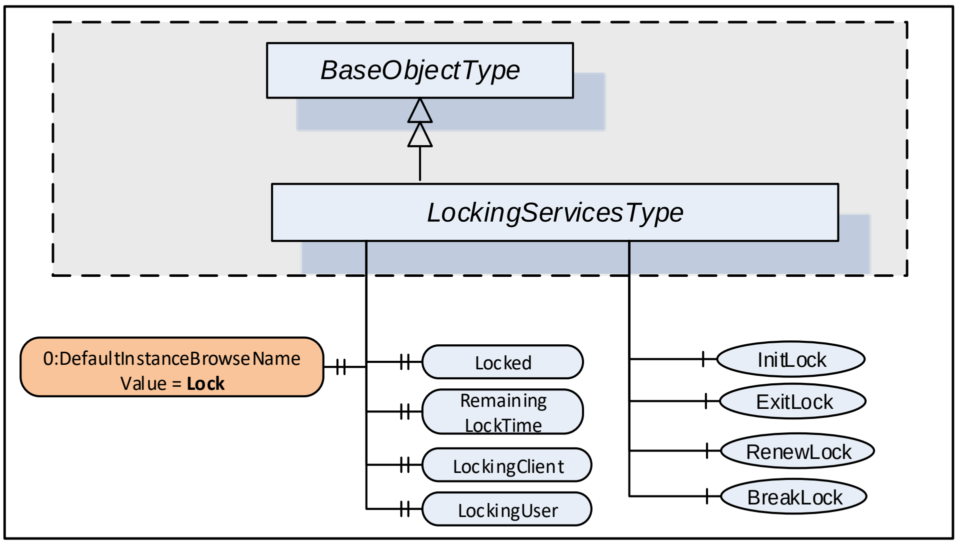

Figure 34 – LockingServicesType

Table 54 – LockingServicesType definition

Attribute	**Value**				
BrowseName	LockingServicesType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseObjectType defined in OPC 10000-5.					
HasComponent	Method	InitLock			Mandatory
HasComponent	Method	RenewLock			Mandatory
HasComponent	Method	ExitLock			Mandatory
HasComponent	Method	BreakLock			Mandatory
HasProperty	Variable	0:DefaultInstanceBrowseName “Lock”	String	PropertyType	
HasProperty	Variable	Locked	Boolean	PropertyType	Mandatory
HasProperty	Variable	LockingClient	String	PropertyType	Mandatory
HasProperty	Variable	LockingUser	String	PropertyType	Mandatory
HasProperty	Variable	RemainingLockTime	Duration	PropertyType	Mandatory

The *StatusCode Bad_MethodInvalid* shall be returned from the Call *Service* for *Objects* where locking is not supported. *Bad_UserAccessDenied* shall be returned if the *Client User* does not have the permission to call the *Methods*.

The *DefaultInstanceBrowseName Property* – defined in OPC 10000-3 – is used to specify the recommended *BrowseName* for instances of the *LockingServicesType*.

The following *LockingServices Properties* offer lock-status information.

Locked when True indicates that this element has been locked by some *Client* and that no or just limited access is available for other *Clients*.

LockingClient contains the ApplicationUri of the *Client* as provided in the CreateSession *Service* call (see OPC 10000-4).

LockingUser contains the identity of the user. It is obtained directly or indirectly from the UserIdentityToken passed by the *Client* in the ActivateSession *Service* call (see OPC 10000-4).

RemainingLockTime denotes the remaining time in milliseconds after which the lock will automatically be timed out by the *Server*. This time is based upon *MaxInactiveLockTime* (see 8.3.4).

8.3.3 LockingServices Object

The support of *LockingServices* for an *Object* is declared by aggregating an instance of the *LockingServicesType* as illustrated in Figure 35.

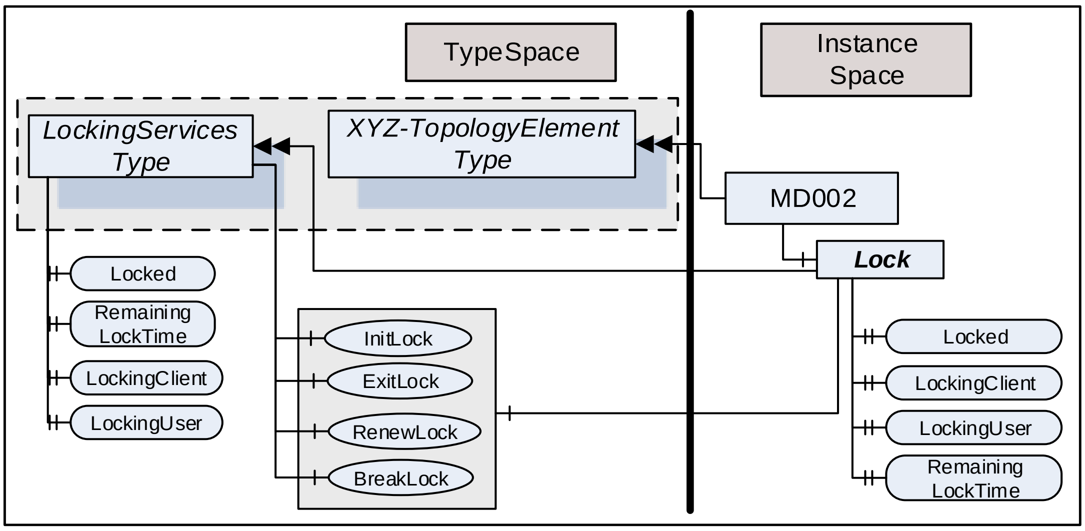

Figure 35 – LockingServices

This *Object* is used as container for the *LockingServices Methods* and *Properties* and should have the *BrowseName* **Lock**. *HasComponent* or *HasAddIn* are used to reference from a *TopologyElement* (for example, a *Device*) to its “LockingServices” *Object*.

The *LockingServiceType* and each instance may share the same *Methods*. All *Properties* are distinct.

8.3.4 MaxInactiveLockTime Property

The *MaxInactiveLockTime Property* shall be added to the *ServerCapabilities Object* (see OPC 10000-5).

It contains a *Server*-specific period of time in milliseconds until which the *Server* will revoke the lock. The *Server* will initiate a timer based on this time as part of processing the *InitLock* request. Calling the *RenewLock Method* as well as other *Service* calls from the *Client* for the locked element shall reset the timer. The lock will never be disabled during execution of a *Service* that requires a lock.

Inactivity for *MaxInactiveLockTime* will trigger a timeout. As a result the *Server* will release the lock.

A timeout shall not cancel any already executing *Services* like Write.

The *MaxInactiveLockTime Property* is formally defined in Table 55.

Table 55 – MaxInactiveLockTime Property definition

References	NodeClass	BrowseName	DataType	TypeDefinition	ModellingRule
HasProperty	Variable	MaxInactiveLockTime	Duration	PropertyType	Mandatory

8.3.5 InitLock Method

InitLock restricts access for other *Clients*.

A call of this *Method* for an element that is already locked will be rejected. This may also be due to an implicit lock created by the *Server*. If *InitLock* is requested for a *Network*, it will be rejected if any of the *Devices* connected to this *Network* or any sub-ordinate *Network* including their connected *Devices* is already locked.

While locked, requests from other *Clients* to modify the locked element (e.g., writing to *Parameters*, modifying the topology, or invoking *Methods*) will be rejected. However, requests to read or navigate will typically work. *Servers* may choose to implement an exclusive locking where other *Clients* have no access at all (e.g. in cases where even read operations require certain settings in a *TopologyElement*).

The lock is removed when *ExitLock* is called. It is automatically removed when the *Session* ends. This is typically the case when the connection to the *Client* breaks and the *Session* times out. *Servers* shall also maintain an automatic unlock if *Clients* do not access the locked element for a certain time (see 8.3.4).

The signature of this *Method* is specified below. Table 56 and Table 57 specify the arguments and *AddressSpace* representation, respectively.

Signature

```
InitLock(
  [in]  String       Context,
  [out] Int32        InitLockStatus);
```

Table 56 – InitLock Method Arguments

Argument	Description
Context	A string used to provide context information about the current activity going on in the *Client*.
InitLockStatus	0 – OK -1 – E_AlreadyLocked – the element is already locked -2 – E_Invalid – the element cannot be locked

Table 57 – InitLock Method AddressSpace definition

Attribute	Value				
BrowseName	InitLock				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Mandatory
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

8.3.6 ExitLock Method

ExitLock removes the lock. This *Method* may only be called from the same *Session* from which *InitLock* had been called.

The signature of this *Method* is specified below. Table 58 and Table 59 specify the arguments and *AddressSpace* representation, respectively.

Signature

```
ExitLock(
  [out] Int32       ExitLockStatus);
```

Table 58 – ExitLock Method Arguments

Argument	Description
ExitLockStatus	0 – OK -1 – E_NotLocked – the Object is not locked

Table 59 – ExitLock Method AddressSpace definition

Attribute	Value				
BrowseName	ExitLock				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

8.3.7 RenewLock Method

The lock timer is automatically renewed whenever the *Client* initiates a request for the locked element or while *Nodes* of the locked element are subscribed to. *RenewLock* is used to reset the lock timer to the value of the *MaxInactiveLockTime Property* and prevent the *Server* from automatically aborting the lock. This *Method* may only be called from the same *Session* from which *InitLock* had been called.

The signature of this Method is specified below. Table 60 and Table 61 specify the arguments and *AddressSpace* representation, respectively.

Signature

```
RenewLock(
  [out] Int32      RenewLockStatus);
```

Table 60 – RenewLock Method Arguments

Argument	Description
RenewLockStatus	0 – OK -1 – E_NotLocked – the Object is not locked

Table 61 – RenewLock Method AddressSpace definition

Attribute	Value				
BrowseName	RenewLock				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

8.3.8 BreakLock Method

BreakLock allows a *Client* (with sufficiently high user rights) to break the lock held by another *Client*. This *Method* will typically be available only to users with administrator privileges. *BreakLock* should be used with care as the locked element may be in an inconsistent state.

The signature of this *Method* is specified below. Table 62 and Table 63 specify the arguments and *AddressSpace* representation, respectively.

Signature

```
BreakLock(
  [out] Int32      BreakLockStatus);
```

Table 62 – BreakLock Method Arguments

Argument	Description
BreakLockStatus	0 – OK -1 – E_NotLocked – the Object is not locked

Table 63 – BreakLock Method AddressSpace definition

Attribute	**Value**				
BrowseName	BreakLock				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Mandatory

9 Specialized topology elements

9.1 General

This section defines specialized types that are commonly used for Field *Devices*. It makes use of the *ConfigurableObjectType* as a way to add functionality using composition.

9.2 Configurable components

9.2.1 General pattern

Subclause 9.2 defines a generic pattern to expose and configure components. It defines the following principles:

- A configurable *Object* shall contain a folder called *SupportedTypes* that references the list of *Types* available for configuring components using *Organizes References*. Sub-folders can be used for further structuring of the set. The names of these sub-folders are vendor specific.
- The configured instances shall be components of the configurable *Object*.

Figure 36 illustrates these principles.

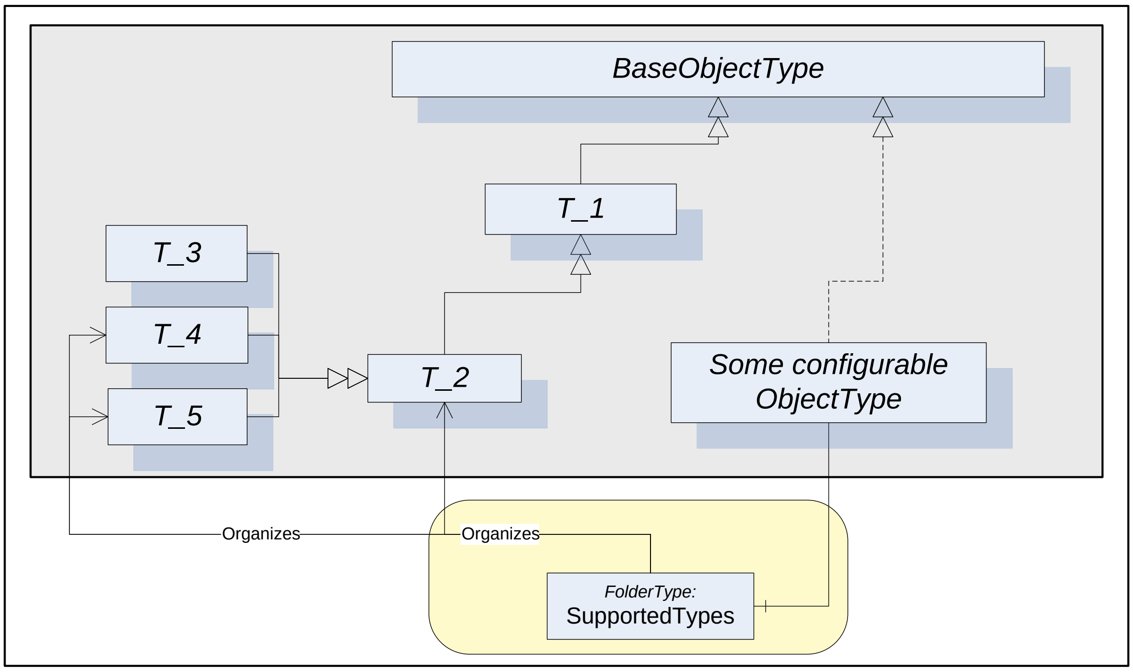

Figure 36 – Configurable component pattern

In some cases the *SupportedTypes* folder on the instance may be different to the one on the *Type* and may contain only a subset. It may be for example that only one instance of each *Type* can be configured. In this case the list of supported *Types* will shrink with each configured component. If the list of supported *Types* is allowed to shrink on an instance, the *TypeDefinition* shall use an appropriate *ModellingRule* like “*Optional*”.

9.2.2 ConfigurableObjectType

This *ObjectType* implements the configurable component pattern and is used when an *Object* or an instance declaration needs nothing but configuration capability. Figure 37 illustrates the

ConfigurableObjectType. It is formally defined in Table 64. Concrete examples are in Clauses 9.3 and 9.4.

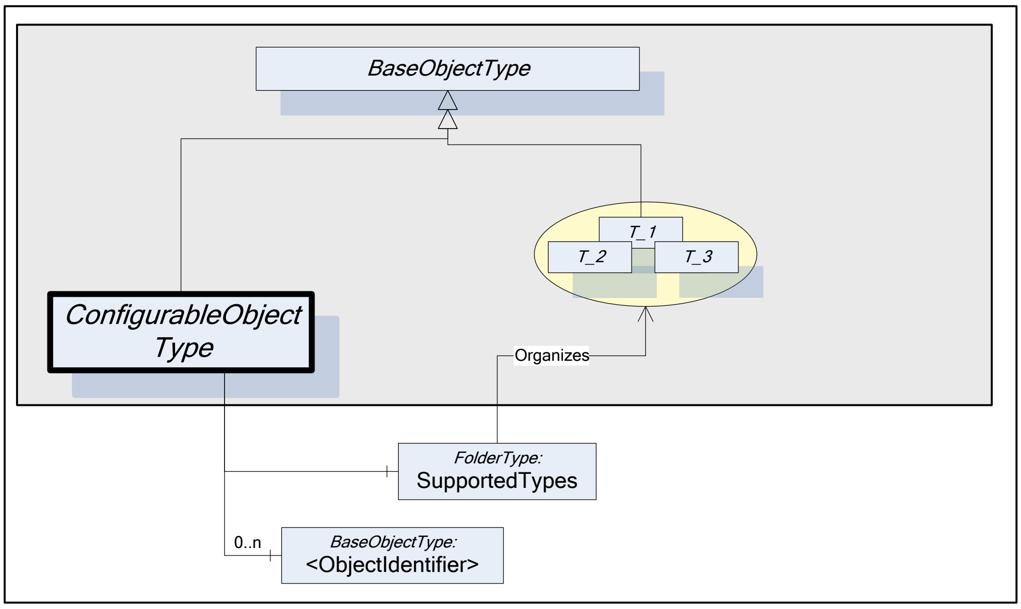

Figure 37 – ConfigurableObjectType

Table 64 – ConfigurableObjectType definition

Attribute	**Value**				
BrowseName	ConfigurableObjectType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in OPC 10000-5					
HasComponent	Object	SupportedTypes		FolderType	Mandatory
HasComponent	Object	<ObjectIdentifier>		BaseObjectType	OptionalPlaceholder

The *SupportedTypes* folder is used to maintain the set of (subtypes of) *BaseObjectTypes* that can be instantiated in this configurable *Object* (the course of action to instantiate components is outside the scope of this specification).

The configured instances shall be components of the *ConfigurableObject*.

9.3 Block Devices

A block-oriented *Device* can be composed using the modelling elements defined in this specification. A block-oriented *Device* includes a configurable set of *Blocks*. Figure 38 shows the general structure of block-oriented *Devices*.

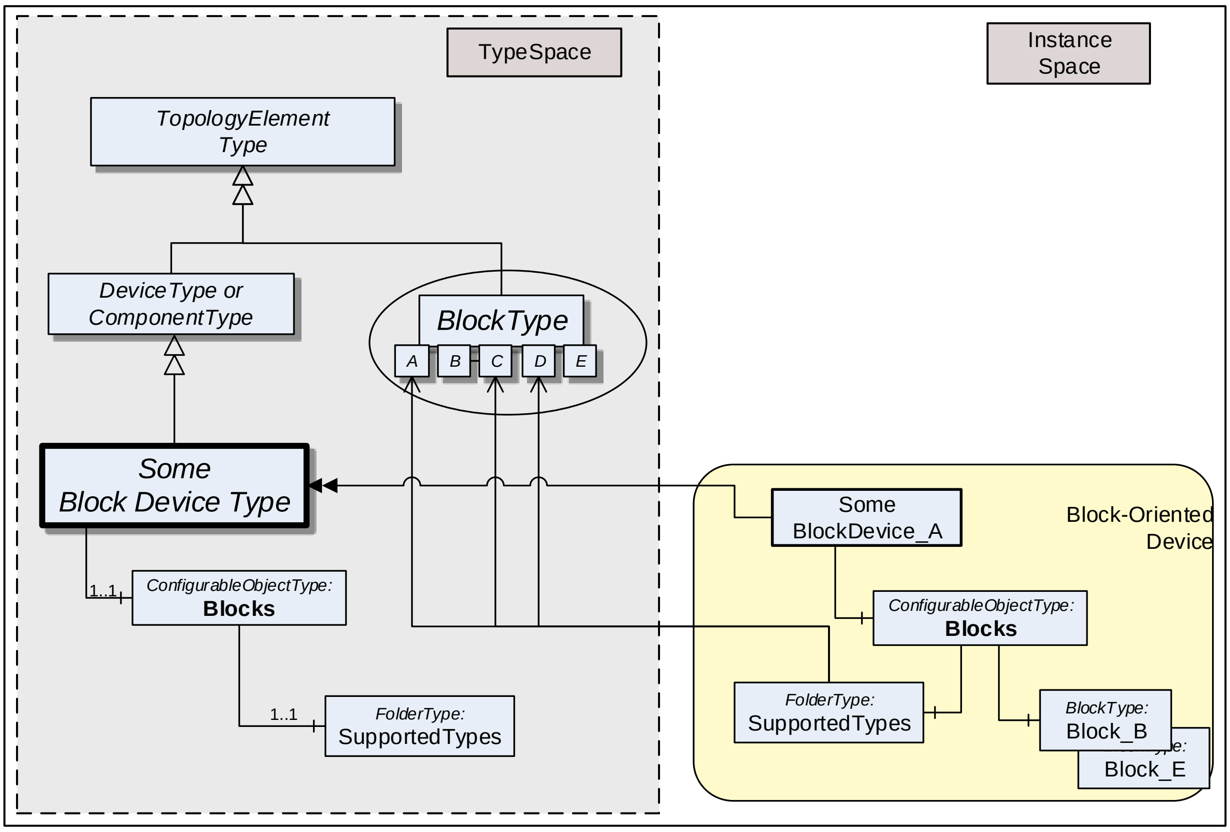

Figure 38 – Block-oriented Device structure example

An *Object* called ***Blocks*** is used as a container for the actual *BlockType* instances. It is of the *ConfigurableObjectType* which includes the *SupportedTypes* folder. The *SupportedTypes* folder for ***Blocks*** is used to maintain the set of (subtypes of) *BlockTypes* that can be instantiated. The supported *Blocks* may be restricted by the block-oriented *Device*. In Figure 38 the *BlockTypes* B and E have already been instantiated. In this example, only one instance of these types is allowed and the *SupportedTypes* folder therefore does not reference these types anymore. See 9.2.1 for the complete definition of the *ConfigurableObjectType*.

9.4 Modular Devices

A *Modular Device* is represented by a (subtype of) *ComponentType* that is composed of a top-*Device* and a set of subdevices (modules). The top-*Device* often is the head module with the program logic but a large part of the functionality depends on the used subdevices. The supported subdevices may be restricted by the *Modular Device*. Figure 39 shows the general structure of *Modular Devices*.

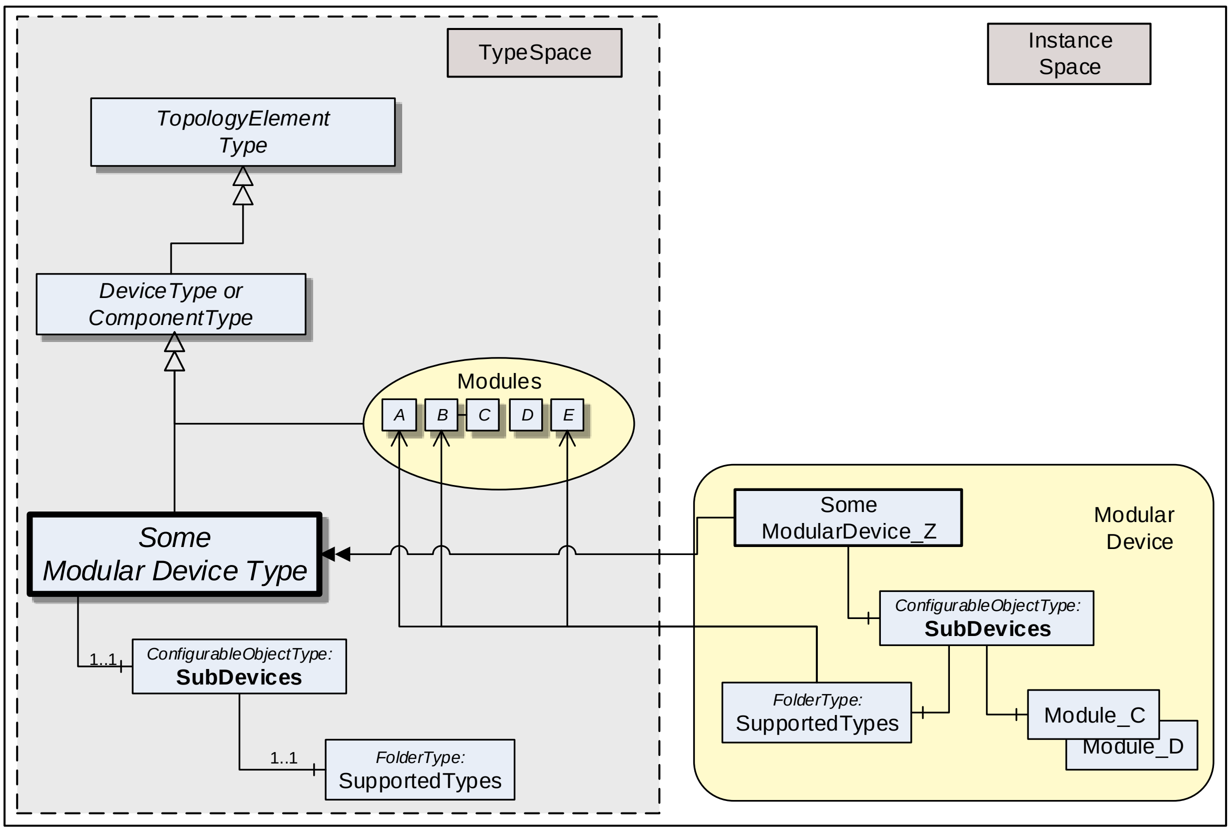
TypeSpace
Instance Space
TopologyElement Type
DeviceType or ComponentType
Modules
A
B
C
D
E
Some Modular Device Type
1..1
ConfigurableObjectType: SubDevices
1..1
FolderType: SupportedTypes
Some ModularDevice_Z
Modular Device
ConfigurableObjectType: SubDevices
FolderType: SupportedTypes
Module_C
Module_D

Figure 39 – Modular Device structure example

The modules (subdevices) of *Modular Devices* are aggregated in the **SubDevices** *Object*. It is of the *ConfigurableObjectType*, which includes the *SupportedTypes* folder. The *SupportedTypes* folder for **SubDevices** is used to maintain the set modules that can be added to the *Modular Device*. Modules are not in the DeviceSet *Object*.

Depending on the actual configuration, *Modular Device* instances might already have a set of pre-configured subdevices. Furthermore, the *SupportedTypes* folder might only refer to a subset of all possible subdevices for the *Modular Device*. In Figure 39 the modules C and D have already been instantiated. In this example, only one instance of these types is allowed and the *SupportedTypes* folder therefore does not reference these types anymore. See clause 9.2.1 for the complete definition of the *ConfigurableObjectType*.

Subdevices may themselves be *Modular Devices*.

10 Profiles

10.1 General

Profiles are named groupings of *ConformanceUnits* as defined in OPC 10000-7. The term *Facet* in the title of a *Profile* indicates that this *Profile* is expected to be part of another larger *Profile* or concerns a specific aspect of OPC UA. *Profiles* with the term Facet in their title are expected to be combined with other *Profiles* to define the complete functionality of an OPC UA *Server* or *Client.*

This specification defines *Facets* for *Servers* or *Clients* when they plan to support OPC UA for Devices. They are described in 10.2 and 10.3.

10.2 Device Server Facets

The following tables specify the *Facets* available for *Servers* that implement the *Devices* companion standard. Table 65 describes *Conformance Units* included in the minimum needed *Facet*. It includes the organisation of instantiated *Devices* in the *Server AddressSpace*.

Table 65 – BaseDevice_Server_Facet definition

Conformance Unit	**Description**	**Optional/ Mandatory**
DI Information Model	Supports *Objects* that conform to the types specified in the chapter *Device* model of this companion standard. This includes in particular *Objects* of (subtypes of) *ComponentType* and *FunctionalGroups*.	M
DI DeviceSet	Supports the **DeviceSet** object to aggregate *Device* instances.	M
DI Nameplate	Supports *Properties* of the *VendorNameplate Interface* defined in 5.5.2.	O
DI Software Component	Supports *Objects* of *SoftwareType* or a subtype.	O
DI DeviceHealth	Supports the DeviceHealth Property defined in 5.5.3.	O
DI DeviceSupportInfo	*Server* provides additional data for its *Devices* as defined in 5.5.5.	O

Table 66 defines a *Facet* for the identification *FunctionalGroup* of *Devices*. This includes the option of identifying the *Protocol*(s).

Table 66 – DeviceIdentification_Server_Facet definition

Conformance Unit	**Description**	**Optional/ Mandatory**
DI Identification	Supports the **Identification** *FunctionalGroup* for *Devices*.	M
DI Protocol	Supports the *ProtocolType* and instances of it to identify the used communication profiles for specific instances.	O

Table 67 defines extensions specifically needed for *BlockDevices*.

Table 67 – BlockDevice_Server_Facet definition

Conformance Unit	**Description**	**Optional/ Mandatory**
DI Blocks	Supports the *BlockType* (or subtypes respectively) and the *Blocks Object* in some of the instantiated *Devices*.	M

Table 68 defines a *Facet* for the Locking *AddIn Capability*. This includes the option of breaking a lock.

Table 68 – Locking_Server_Facet definition

Conformance Unit	**Description**	**Optional/ Mandatory**
DI Locking	Supports the *LockingService* for certain *TopologyElements*.	M
DI BreakLocking	Supports the *BreakLock Method* to break the lock held by another *Client*.	O

Table 69 defines a *Facet* for the support of the Device Communication model.

Table 69 – DeviceCommunication_Server_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI Network	Supports the *NetworkType* to instantiate *Network* instances.	M
DI ConnectionPoint	Supports subtypes of the *ConnectionPointType*.	M
DI NetworkSet	Supports the *NetworkSet Object* to aggregate all *Network* instances.	M
DI ConnectsTo	Supports the *ConnectsTo Reference* to associate *Devices* with a *Network*.	M

Table 70 defines a *Facet* for the support of the Device Integration Host model.

Table 70 – DeviceIntegrationHost_Server_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI DeviceTopology	Supports the *DeviceTopology Object* as starting *Node* for the communication topology of the *Devices* to integrate.	M
DI Offline	Supports offline and online representations of *Devices* including the *Methods* to transfer data from or to the *Device*.	M

10.3 Device Client Facets

The following tables specify the *Facets* available for *Clients* that implement the *Devices* companion standard. Table 71 describes *Conformance Units* included in the minimum needed *Facet*.

Table 71 – BaseDevice_Client_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI Client Information Model	Consumes *Objects* that conform to the types specified in the chapter *Device* model of this companion standard. This includes in particular *Objects* of (subtypes of) *ComponentType* and *FunctionalGroups*.	M
DI Client DeviceSet	Uses the **DeviceSet** *Object* to detect available *Devices*.	M
DI Client Nameplate	Consumes *Properties* of the *VendorNameplate Interface* defined in 5.5.2.	O
DI Client Software Component	Consumes *Objects* of *SoftwareType* or a subtype.	O
DI Client DeviceHealth	Uses the *DeviceHealth Property* defined in 5.5.3.	O
DI Client DeviceSupportInfo	Uses available additional data for *Devices* as defined in 5.5.5.	O

Table 72 defines a *Facet* for the **identification** *FunctionalGroup* of *Devices*. This includes the option of identifying the *Protocol*(s).

Table 72 – *DeviceIdentification_Client_Facet* definition

Conformance Unit	Description	Optional/ Mandatory
DI Client Identification	Consumes the **Identification** *FunctionalGroup* for *Devices* including the (optional) reference to supported protocol(s).	M

Table 73 defines extensions specifically needed for *BlockDevices*.

Table 73 – BlockDevice_Client_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI Client Blocks	Understands and uses *BlockDevices* and their *Blocks* including *FunctionalGroups* on both *Device* and *Block* level.	M

Table 74 defines a *Facet* for the Locking *AddIn Capability*. This includes the option of breaking a lock.

Table 74 – Locking_Client_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI Locking	Uses the *LockingService* where available.	M
DI BreakLocking	Support use of the *BreakLock Method* to break the lock held by another *Client*.	O

Table 75 defines a *Facet* for the support of the Device Communication model.

Table 75 – DeviceCommunication_Client_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI Network	Uses the *NetworkType* to instantiate *Network* instances.	M
DI ConnectionPoint	Uses subtypes of the *ConnectionPointType*.	M
DI NetworkSet	Uses the *NetworkSet Object* to store or find *Network* instances.	M
DI ConnectsTo	Uses the *ConnectsTo Reference* to associate *Devices* with a *Network*.	M

Table 76 defines a *Facet* for the support of the Device Integration Host model.

Table 76 – DeviceIntegrationHost_Client_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI DeviceTopology	Uses the *DeviceTopology Object* as starting *Node* for the communication topology of the *Devices* to integrate.	M
DI Offline	Uses offline and online representations of *Devices* including the *Methods* to transfer data from or to the *Device*.	M

11 Namespaces

11.1 Namespace Metadata

Editor note: Update the table on release date

Table 77 defines the namespace metadata for this specification. The *Object* is used to provide version information for the namespace and an indication about static *Nodes*. Static *Nodes* are identical for all *Attributes* in all *Servers*, including the *Value Attribute*. See OPC 10000-5 for more details.

The information is provided as *Object* of type *NamespaceMetadataType*. This *Object* is a component of the *Namespaces Object* that is part of the *Server Object*. The *NamespaceMetadataType ObjectType* and its *Properties* are defined in OPC 10000-5.

The version information is also provided as part of the ModelTableEntry in the UANodeSet XML file. The UANodeSet XML schema is defined in OPC 10000-6.

Table 77 – NamespaceMetadata Object for this Specification

Attribute	**Value**		
BrowseName	http://opcfoundation.org/UA/DI/		
References	**BrowseName**	**DataType**	**Value**
HasProperty	NamespaceUri	String	http://opcfoundation.org/UA/DI/
HasProperty	NamespaceVersion	String	1.02
HasProperty	NamespacePublicationDate	DateTime	2019-xx-xx
HasProperty	IsNamespaceSubset	Boolean	Vendor-specific
HasProperty	StaticNodeIdTypes	IdType[]	Null
HasProperty	StaticNumericNodeIdRange	NumericRange[]	{0:9999}
HasProperty	StaticStringNodeIdPattern	String	Null

11.2 Handling of OPC UA namespaces

Namespaces are used by OPC UA to create unique identifiers across different naming authorities. The *Attributes NodeId* and *BrowseName* are identifiers. A *Node* in the UA *Address Space* is unambiguously identified using a *NodeId*. Unlike *NodeIds*, the *BrowseName* cannot be used to unambiguously identify a *Node*. Different *Nodes* may have the same *BrowseName*. They are used to build a browse path between two nodes or to define a standard *Property*.

Servers may often choose to use the same namespace for the *NodeId* and the *BrowseName*. However, if they want to provide a standard *Property*, its *BrowseName* shall have the namespace of the standards body although the namespace of the *NodeId* reflects something else, for example the *EngineeringUnits Property*. All *NodeIds* of *Nodes* not defined in this specification shall not use the standard namespaces.

Table 78 provides a list of mandatory and optional namespaces used in a DI OPC UA *Server*.

Table 78 – Namespaces used in an OPC UA for Devices Server

NamespaceURI	**Description**	**Use**
http://opcfoundation.org/UA/	Namespace for *NodeIds* and *BrowseNames* defined in the OPC UA specification. This namespace shall have namespace index 0.	Mandatory
Local Server URI	Namespace for *Nodes* defined in the local *Server*. This may include types and instances used in a *Device* represented by the *Server*. This namespace shall have namespace index 1.	Mandatory
http://opcfoundation.org/UA/DI/	Namespace for *NodeIds* and *BrowseNames* defined in this specification. The namespace index is *Server* specific.	Mandatory
Vendor specific types and instances	A *Server* may provide vendor specific types like types derived from *TopologyElementType* or *NetworkType* or vendor-specific instances of those types in a vendor specific namespace.	Optional

Table 79 provides a list of namespaces and their index used for *BrowseNames* in this specification. The default namespace of this specification is not listed since all *BrowseNames* without prefix use this default namespace.

Table 79 – Namespaces used in this specification

NamespaceURI	**Namespace Index**	**Example**
http://opcfoundation.org/UA/	0	0:EngineeringUnits

Annex A
(normative)

Namespace and mappings

This Annex defines the numeric identifiers for all of the numeric *NodeIds* defined in this standard. The identifiers are specified in a CSV file with the following syntax:

```
<SymbolName>, <Identifier>, <NodeClass>
```

where the *SymbolName* is either the *BrowseName* of a *Type Node* or the *BrowsePath* for an *Instance Node* that appears in the specification and the *Identifier* is the numeric value for the *NodeId*.

The *BrowsePath* for an instance *Node* is constructed by appending the *BrowseName* of the instance *Node* to the *BrowseName* for the containing instance or type. An underscore character is used to separate each *BrowseName* in the path. Let's take for example, the *DeviceType ObjectType Node* which has the *SerialNumber Property*. The *SymbolName* for the *SerialNumber InstanceDeclaration* within the *DeviceType* declaration is: *DeviceType_SerialNumber*.

The *NamespaceUri* for all *NodeIds* defined here is http://opcfoundation.org/UA/DI/

The CSV released with this version of the standard can be found at:
http://www.opcfoundation.org/UADevices/1.2/NodeIds.csv

NOTE 1 The latest CSV that is compatible with this version of the standard can be found at:
http://www.opcfoundation.org/UADevices/NodeIds.csv

A computer processible version of the complete Information Model defined in this standard is also provided. It follows the XML Information Model schema syntax defined in OPC 10000-6.

The Information Model Schema released with this version of the standard can be found at:
http://www.opcfoundation.org/UADevices/1.2/Opc.Ua.Di.NodeSet2.xml

NOTE 2 The latest Information Model schema that is compatible with this version of the standard can be found at:
http://www.opcfoundation.org/UADevices/Opc.Ua.Di.NodeSet2.xml

Annex B
(informative)

Examples

This Annex includes examples referenced in the normative sections.

B.1 Functional Group Usages

The examples in Figure B.1 and Figure B.2 illustrate the use of *FunctionalGroups:*

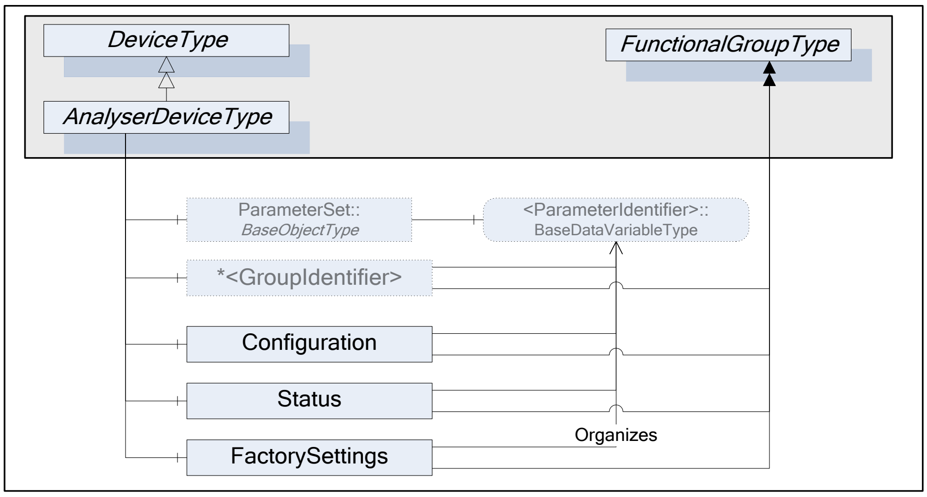

Figure B.1 – Analyser Device use for FunctionalGroups

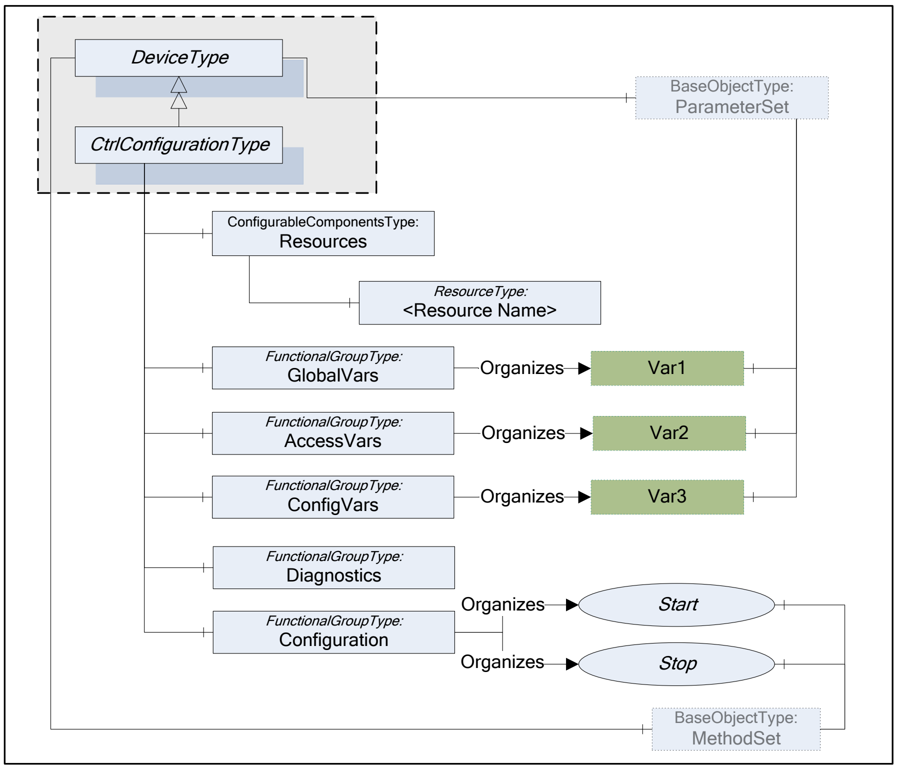

Figure B.2 – PLCopen use for FunctionalGroups

B.2 Identification Functional Group

The *Properties* of a *TopologyElement*, like Manufacturer, SerialNumber, will usually be sufficient as identification. If other *Parameters* or even *Methods* are required, all elements needed shall be organised in a *FunctionalGroup* called **Identification**. Figure B.3 illustrates the **Identification** *FunctionalGroup* with an example.

Note that companion standards are expected to define the Identification contents for their model.

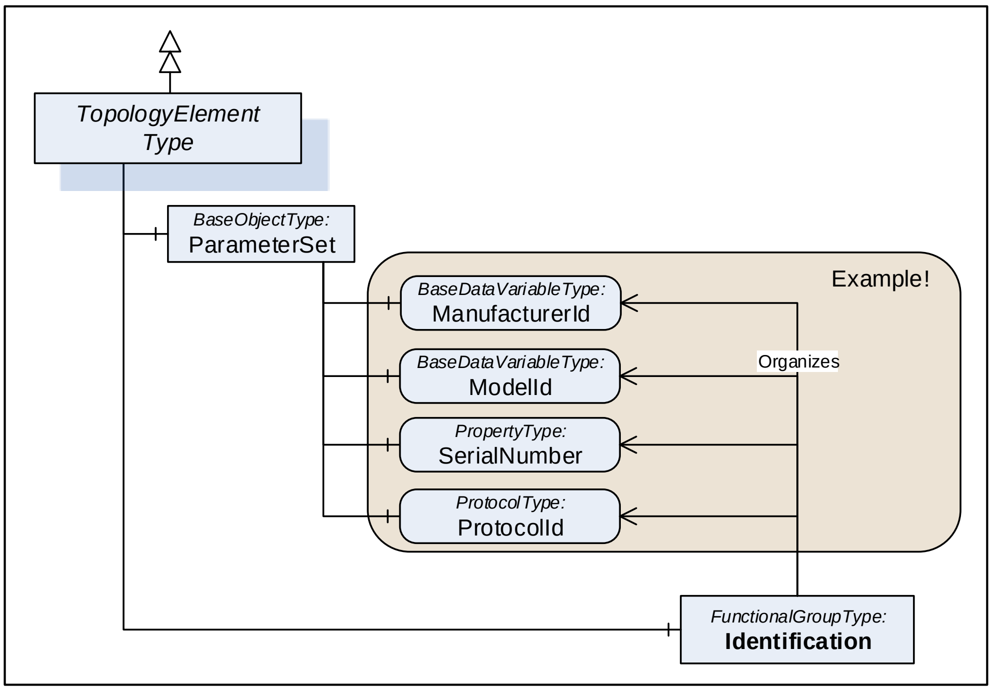

Figure B.3 – Example of an Identification FunctionalGroup

Annex C
(informative)

Guidelines for the usage of OPC UA for Devices as base for Companion Specifications

This informative Annex describes guidelines for the usage of this specification as base for creating companion specifications as well as guidelines on how to combine different companion specifications based on this specification describing different aspects of the same device in one OPC UA application.

C.1 Overview

This specification is used as base for many other companion specifications like

- OPC UA for IEC61131-3
- OPC UA Information Model for FDT Technology
- AutoId
- OPC UA for IO-Link.

Those companion specifications define different aspects of devices, for example

- some specific functionality (like the scan operation of a RFID reader in the AutoId spec),
- the view of the device accessed by a specific protocol (like IO-Link),
- or the configuration capabilities of a device as defined in a vendor-specific device package (like FDI or FDT).

When an OPC UA application wants to combine those different aspects of one device in its address space, there are potential problems as shown in Figure C.1. The example shows the application of the AutoId specification as well as the FDT specification for the same device. For simplicity, only the base ObjectTypes are shown. In reality, there has to be a subtype of the abstract FdtDeviceType and there would be very likely a vendor-specific subtype of the RfidReaderDeviceType.

As shown in the figure, there are actually two Objects of different ObjectTypes representing different aspects of the same device in the real world.

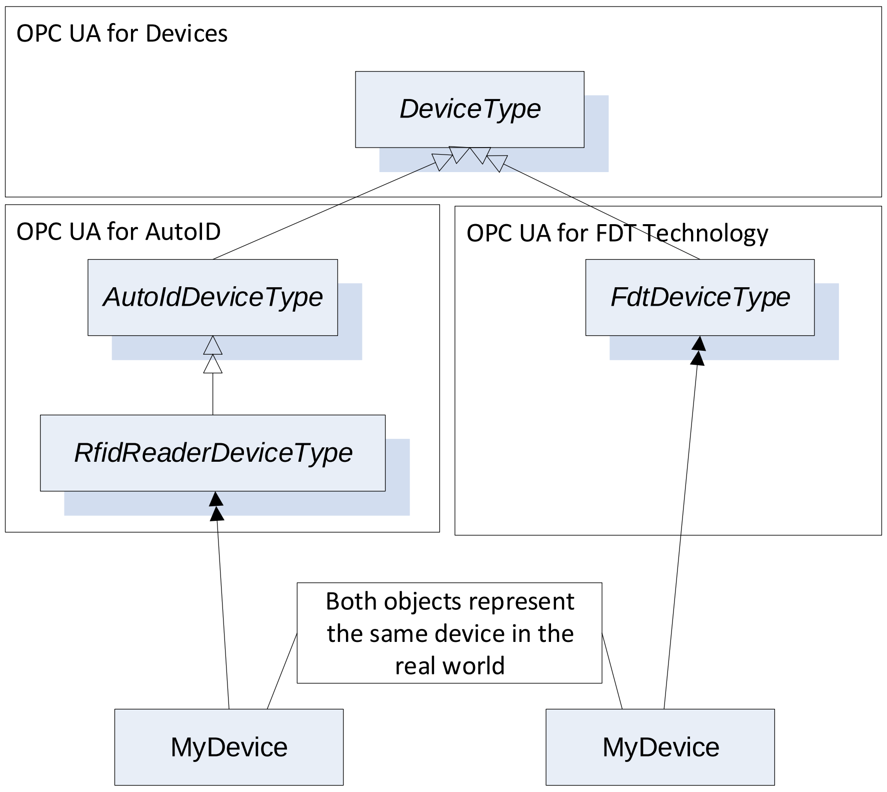

Figure C.1 – Example of applying two companion specifications based on OPC UA for Devices

In order to avoid multiple-inheritance, which is not further defined in OPC UA, it is not possible to directly combine both ObjectTypes into one ObjectType containing all aspects of the device. And an Object cannot be defined by two ObjectTypes. Therefore, in order to expose the information, that both Objects actually represent different aspects of the same device, composition should be used as shown in Figure C.2.

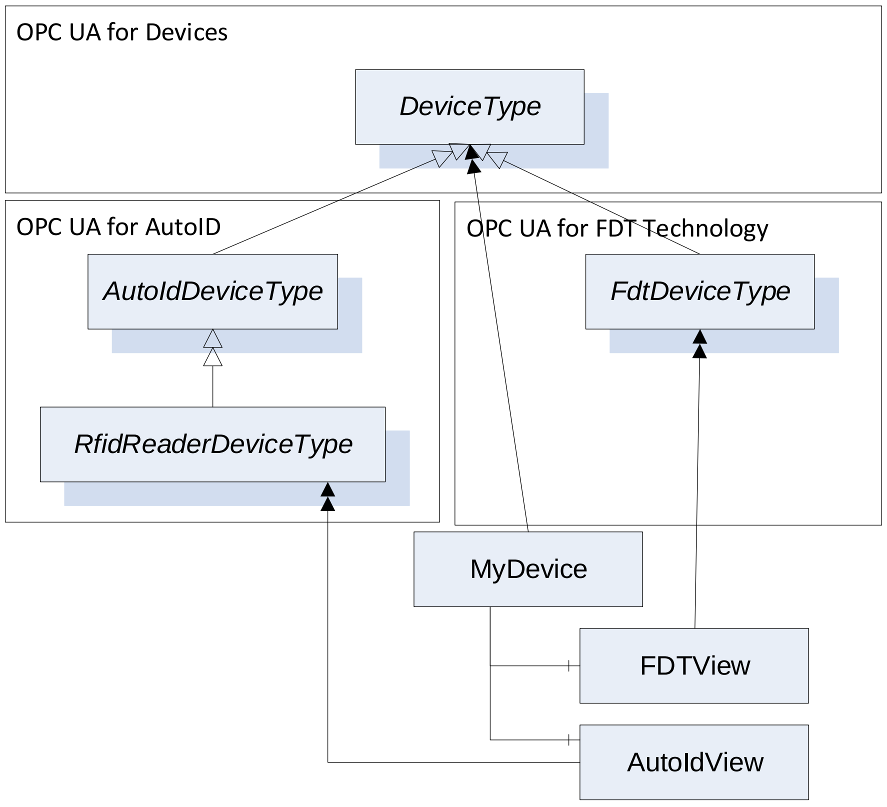

Figure C.2 – Using composition to compose one device representation defined by two companion specifications

In this case, the device is represented by an Object "MyDevice" where the vendor of the OPC UA Application can provide its specific knowledge of the device. In addition, the Object has two components called FDTView and AutoIdView in the figure, containing the information as defined in the corresponding companion specifications.

C.2 Guidelines to define Companion Specifications based on OPC UA for Devices

As shown in the previous section, composition can be used to combine the ObjectTypes defined by various specifications describing aspects of a device in order to combine the information in one OPC UA application. This can lead, as shown in the example in Figure C.2, to the usage of several instances of the DeviceType to represent one device. In order to avoid this, it is recommended that companion specifications do not directly derive from the DeviceType but instead derive from the TopologyElementType or other subtypes of the TopologyElementType (but not the DeviceType). This allows an OPC UA application to represent the device by one instance of the DeviceType and compose potentially several other aspects without the need to use the DeviceType again.

The DeviceType defines several Properties identifying the device as mandatory. By the above described approach, the Properties do not need to be repeated several times as needed in the example in Figure C.2. Here, the mandatory SerialNumber is a Property of MyDevice, FDTView, and AutoIdView. However, companion specification can still define some of those Properties on their ObjectTypes, either optional in order to allow the usage of their ObjectTypes without an additional Object (for example if only one companion specification is supported by the OPC UA application) or mandatory, if a specific access-path to the information shall be exposed. For example, the SerialNumber accessed by a specific protocol might be different than the SerialNumber managed directly by the DeviceVendor. Whereas Profibus or IO-Link represent the SerialNumber as a String, the HART protocol uses three Bytes. So, if a companion specification should expose the SerialNumber accessed via HART, it can add it as mandatory Property to its ObjectType. To conclude, it is recommended that companion specification provide the Properties of the DeviceType by implementing the IVendorNameplateType, which adds all the Properties optionally to the ObjectType.

If desired, they can make some of those Properties mandatory to force that a specific access path is used (e.g. via a specific protocol).

In order to easily identify the components representing different views on the device, it is recommended to use the AddIn concept to define a standardized BrowseName for the Object (DefaultInstanceBrowseName Property). In the example in Figure C.2 that would mean that FdtDeviceType would have defined a DefaultInstanceBrowseName “FDTView”, and thus OPC UA Clients can easily find the FDT specific data of the device by looking for an Instance called “FDTView”, for example by using the TranslateBrowsePathsToNodeIds Service.

C.3 Guidelines on how to combine different companion specifications based on OPC UA for Devices in one OPC UA application

When supporting several companion specifications in one OPC UA application it is recommended to use the composition approach as described in section C.1. To expose the possibilities further, the example is extended as shown in Figure C.3. Again, subtypes for the concrete type of device are not considered for simplicity. The IOLinkDeviceType is already not derived from DeviceType but TopologyElementType. As the FDT and AutoID specifications derive from DeviceType, the device is represented by several instances of the DeviceType.

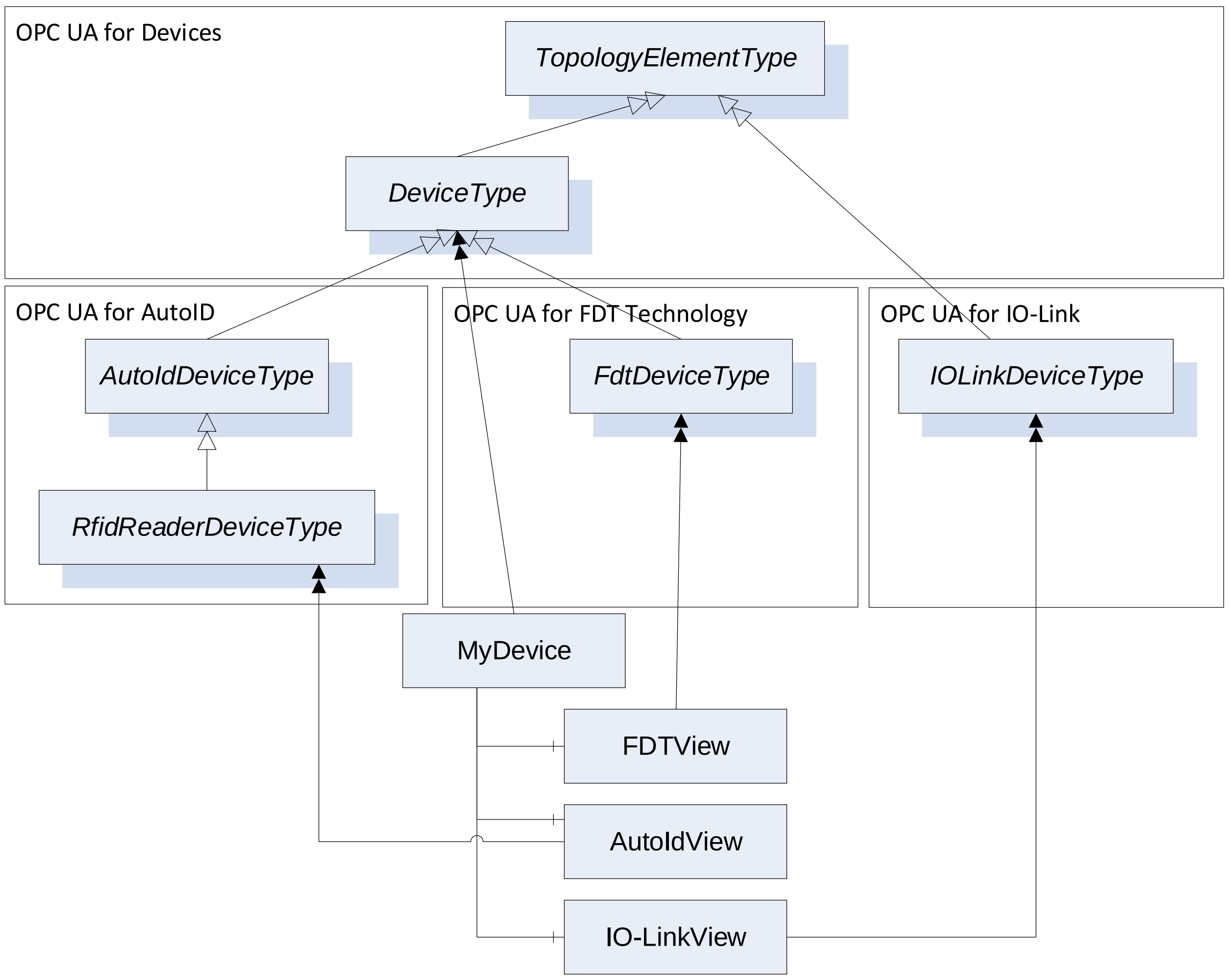

Figure C.3 – Example of applying several companion specifications (I)

In order to limit the usage of DeviceType instances, an alternative approach is shown in Figure C.4. Here, the RfidReaderDeviceType is used as main Object to represent the device, and the objects defined by the other companion specifications are composed.

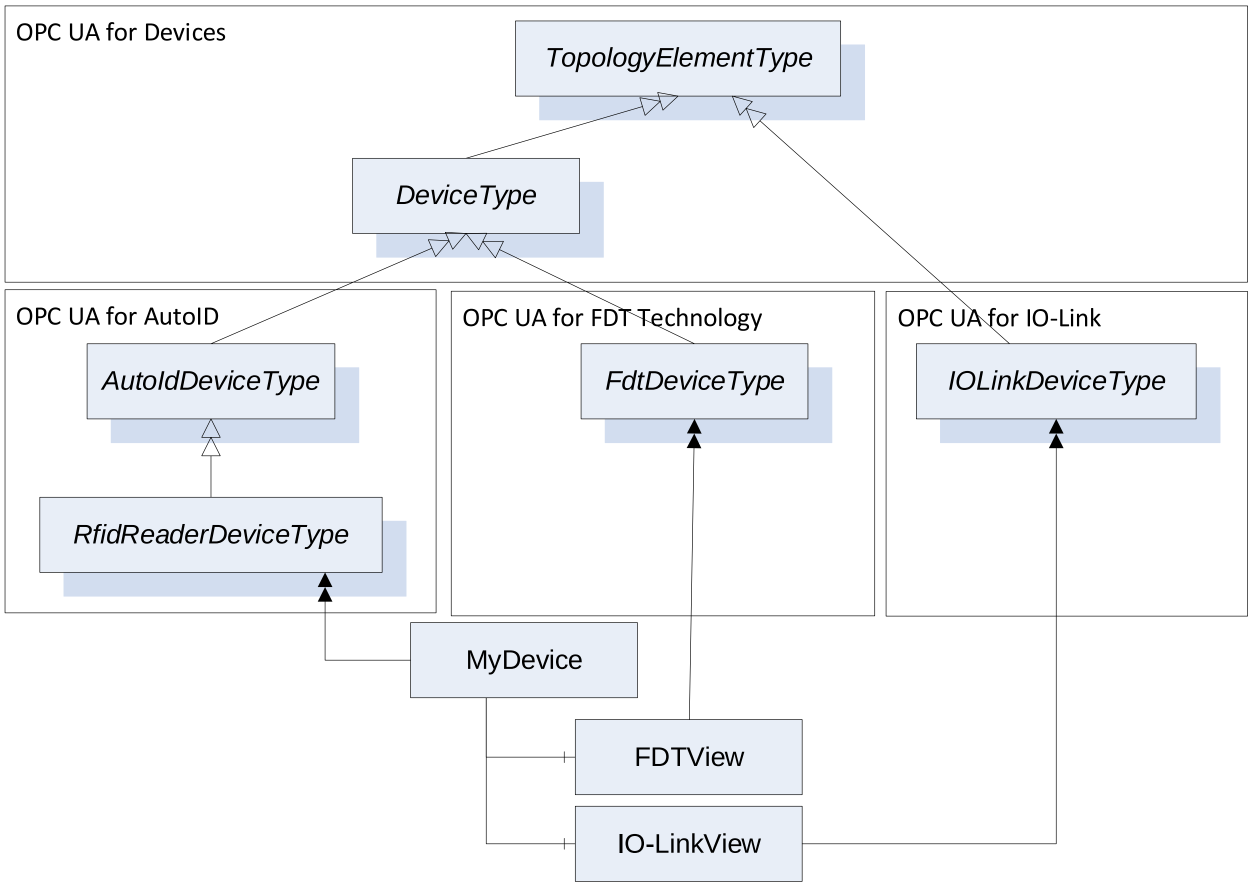

Figure C.4 – Example of applying several companion specifications (II)

It is recommended to use one of the two approaches described above.

Bibliography

IEC 61784: *Industrial Communication Networks - Profiles*

IEC 61499-1 ed2.0: *Function Blocks –Part 1: Architecture*

IEC 62591: *Industrial communication networks - Wireless communication network and communication profiles - WirelessHART™*

[IEC 61131], *IEC standard for Programmable Logic Controllers (PLCs)*