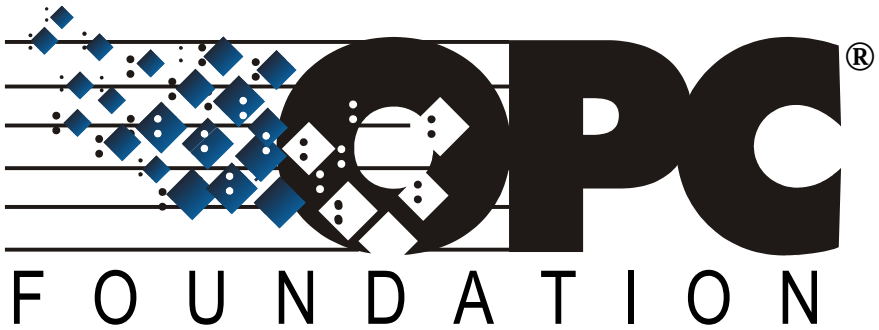

OPC 10000-100

OPC Unified Architecture

Part 100: Devices

Release 1.03.1

2021-12-07

OPC UA Specification

Specification Type	Industry Standard Specification	Comments:	
Document Number	**OPC 10000-100**		
Title:	OPC Unified Architecture Devices	Date:	2021-12-07
Version:	Release 1.03.1	Software Source:	MS-Word OPC 10000-100 - UA Specification Part 100 - Devices v1.03.1.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.

AGREEMENT OF USE

only if the software satisfactorily meets the certification requirements set by the OPC Foundation. Products that do not meet these requirements may claim only that the product was based on this specification and must not claim compliance or conformance with this specification.

TRADEMARKS

Most computer and software brand names have trademarks or registered trademarks. The individual trademarks have not been listed here.

GENERAL PROVISIONS

Should any provision of this Agreement be held to be void, invalid, unenforceable or illegal by a court, the validity and enforceability of the other provisions shall not be affected thereby.

This Agreement shall be governed by and construed under the laws of the State of Minnesota, excluding its choice or law rules.

This Agreement embodies the entire understanding between the parties with respect to, and supersedes any prior understanding or agreement (oral or written) relating to, this specification.

ISSUE REPORTING

The OPC Foundation strives to maintain the highest quality standards for its published specifications; hence they undergo constant review and refinement. Readers are encouraged to report any issues and view any existing errata here: http://www.opcfoundation.org/errata

Revision 1.3.x Highlights

This revision contains extensions to Version 1.3.

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

Mantis ID	Summary	Resolution
Issues resolved with revision 1.03.0		
6195	LockingService should be usable outside DI	Moved type-specific semantic to TopologyElementType and NetworkType. Made Locking a separate chapter.
6196	Relation of types to conformance units missing	Added the relevant CU name to the type tables.
6230	Abstract needs to be reversed between DeviceHealthDiagnosticAlarmType and its subtypes	Fixed as suggested.
6282	Feature for Software Update needed	Added new Software Update Feature (Firmware Update) as a new chapter.
6514	Mandatory Placeholder in MethodSet	Removed the instance declaration and specified the expected behavior in text.
Issues resolved with revision 1.03.1		
6927	Add PatchIdentifiers and SoftwareReleaseDate Properties to IVendorNameplateType	Added as suggested
6928	Standardized FunctionalGroup Name for OperationCounters	Added as suggested
6926	Add capability to provide a semantic version for SoftwareRevision	Added as suggested

OPC UNIFIED ARCHITECTURE –

Part 100: Devices

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.

This document also defines AddIns that can be used for the models in this document but also for models in other specifications. They are:

- Locking model – a generic AddIn to control concurrent access,
- Software update model – an AddIn to manage software in a *Device*.

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, *OPC UA Companion Specification for Analyser Devices*

OPC 30000, *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, abbreviated terms, and conventions

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in 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

3.1.11
Direct-Loading
an update method where the original software is overwritten during the transfer

3.1.12
Cached-Loading
an update method where the new software is stored in a separate area

Note 1 to entry: Installation is performed later as an extra step.

3.1.13
File System based Loading
an update method based on an accessible directory structure and a separate install method

3.1.14
Software Package
a single file that contains the data for the software update in a device specific format

3.1.15
Software Update Client
an update client that can be used for devices of several vendors

Note 1 to entry: There can be different Software Update Clients for different domains (e.g. process industry or manufacturing).

3.1.16
Current Version
version information of the software that is currently installed

3.1.17
Pending Version
version information for a *Software Package* that was transferred before and is ready to be installed

3.1.18
Fallback Version
version information about an alternatively installable software that is located on the *Server*

Note 1 to entry: Examples: factory default version or the version before the latest update

3.2 Abbreviated terms

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 Node descriptions

3.3.1.1 Node definitions

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
0:Int32	0:Int32	-1	omitted or null	A scalar Int32.
0:Int32[]	0:Int32	1	omitted or {0}	Single-dimensional array of Int32 with an unknown size.
0:Int32[][]	0:Int32	2	omitted or {0,0}	Two-dimensional array of Int32 with unknown sizes for both dimensions.
0:Int32[3][]	0: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.
0:Int32[5][3]	0: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.
0:Int32{Any}	0:Int32	-2	omitted or null	An Int32 where it is unknown if it is scalar or array with any number of dimensions.
0:Int32{ScalarOrOneDimension}	0: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.

Note that if a symbolic name of a different namespace is used, it is prefixed by the *NamespaceIndex* (see 3.3.2.2).

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 Other 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 "--" is used.				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Other**
ReferenceType name	*NodeClass* of the target *Node*.	*BrowseName* of the target *Node*.	*DataType* of the referenced *Node*, only applicable for *Variables*.	*TypeDefinition* of the referenced *Node*, only applicable for *Variables* and *Objects*.	Additional characteristics of the *TargetNode* such as the *ModellingRule* or *AccessLevel*.
NOTE Notes referencing footnotes of the table content.					

Components of *Nodes* can be complex that is containing components by themselves. The *TypeDefinition*, *NodeClass* and *DataType* can be derived from the type definitions, and the symbolic name can be created as defined in 3.3.3.1. Therefore, those containing components are not explicitly specified; they are implicitly specified by the type definitions.

The Other column defines additional characteristics of the Node. Examples of characteristics that can appear in this column are show in Table 3.

Table 3 – Examples of Other Characteristics

Name	Short Name	Description
0:Mandatory	M	The *Node* has the *Mandatory ModellingRule*.
0:Optional	O	The *Node* has the *Optional ModellingRule.*
0:MandatoryPlaceholder	MP	The *Node* has the *MandatoryPlaceholder ModellingRule*.
0:OptionalPlaceholder	OP	The *Node* has the *OptionalPlaceholder ModellingRule.*
ReadOnly	RO	The *Node AccessLevel* has the *CurrentRead* bit set but not the *CurrentWrite* bit.
ReadWrite	RW	The *Node AccessLevel* has the *CurrentRead* and *CurrentWrite* bits set.
WriteOnly	WO	The Node AccessLevel has the *CurrentWrite* bit set but not the *CurrentRead* bit.

If multiple characteristics are defined they are separated by commas. The name or the short name may be used.

Each *Node* defined in this specification has *ConformanceUnits* defined in 10.1 that require the *Node* to be in the *AddressSpace*. If a *Server* supports a *ConformanceUnit*, it shall expose the *Nodes* related to the *ConformanceUnit* in its *AddressSpace*. If two *Nodes* are exposed, all *References* between the *Nodes* defined in this specification shall be exposed as well.

The relations between *Nodes* and *ConformanceUnits* are defined at the end of the tables defining *Nodes*, one row per *ConformanceUnit*. The *ConformanceUnit* is reflected with a Category element in the *UANodeSet* file (see OPC 10000-6).

The *Nodes* defined in a table are not only the *Node* defined on top level, for example an *ObjectType*, but also the *Nodes* that are referenced, as long as they are not defined in other tables. For example, the *ObjectType TopologyElementType* defines its *InstanceDeclarations* in the same table, so the *InstanceDeclarations* are also bound to the *ConformanceUnits* defined for the table. The table even indirectly defines additional *InstanceDeclarations* as components of the top-level *InstanceDeclarations*, that are not directly visible in the table. The *TypeDefinitions* and *DataTypes* used in the *InstanceDeclarations*, and the *ReferenceTypes* are defined in their individual tables and not in the table itself, therefore they are not bound to the *ConformanceUnits* of the table.

3.3.1.2 Additional References

To provide information about additional *References*, the format as shown in Table 4 is used.

Table 4 – <some>Type Additional References

SourceBrowsePath	Reference Type	Is Forward	TargetBrowsePath
SourceBrowsePath is always relative to the *TypeDefinition*. Multiple elements are defined as separate rows of a nested table.	*ReferenceType* name	True = forward *Reference*.	TargetBrowsePath points to another *Node*, which can be a well-known instance or a *TypeDefinition*. You can use *BrowsePaths* here as well, which is either relative to the *TypeDefinition* or absolute. If absolute, the first entry needs to refer to a type or well-known instance, uniquely identified within a namespace by the *BrowseName*.

References can be to any other *Node*.

3.3.1.3 Additional sub-components

To provide information about sub-components, the format as shown in Table 5 is used.

Table 5 – <some>Type Additional Subcomponents

BrowsePath	References	NodeClass	BrowseName	DataType	TypeDefinition	Others
BrowsePath is always relative to the *TypeDefinition*. Multiple elements are defined as separate rows of a nested table	NOTE Same as for Table 2					

3.3.1.4 Additional Attribute values

The type definition table provides columns to specify the values for required *Node Attributes* for *InstanceDeclarations*. To provide information about additional *Attributes*, the format as shown in Table 6 is used.

Table 6 – <some>Type Attribute values for child Nodes

BrowsePath	<Attribute name> Attribute
BrowsePath is always relative to the *TypeDefinition*. Multiple elements are defined as separate rows of a nested table	The values of attributes are converted to text by adapting the reversible JSON encoding rules defined in OPC 10000-6. If the JSON encoding of a value is a JSON string or a JSON number then that value is entered in the value field. Double quotes are not included. If the DataType includes a NamespaceIndex (QualifiedNames, NodeIds or ExpandedNodeIds) then the notation used for BrowseNames is used. If the value is an Enumeration the name of the enumeration value is entered. If the value is a Structure then a sequence of name and value pairs is entered. Each pair is followed by a newline. The name is followed by a colon. The names are the names of the fields in the DataTypeDefinition. If the value is an array of non-structures then a sequence of values is entered where each value is followed by a newline. If the value is an array of Structures or a Structure with fields that are arrays or with nested Structures then the complete JSON array or JSON object is entered. Double quotes are not included.

There can be multiple columns to define more than one *Attribute*.

3.3.2 NodeIds and BrowseNames

3.3.2.1 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 document 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 document, the symbolic name is unique.

The *NamespaceUri* for all *NodeIds* defined in this document 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 document 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 *Server*-specific, including the namespace. But the *NamespaceIndex* of those *Nodes* cannot be the *NamespaceIndex* used for the *Nodes* defined in this document, because they are not defined by this document but generated by the *Server*.

3.3.2.2 BrowseNames

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

For *InstanceDeclarations* of *NodeClass Object* and *Variable* that are placeholders (*OptionalPlaceholder* and *MandatoryPlaceholder ModellingRule*), the *BrowseName* and the *DisplayName* are enclosed in angle brackets (<>) as recommended in OPC 10000-3. If the *BrowseName* is not defined by this document, a namespace index prefix is added to the *BrowseName* (e.g., prefix '0' leading to '0:EngineeringUnits' or prefix '2' leading to '2:DeviceRevision'). This is typically necessary if a *Property* of another specification is overwritten or used in the OPC UA types defined in this document. Clause 11.2 provides a list of namespaces and their indexes as used in this document.

3.3.3 Common Attributes

3.3.3.1 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 document or if it is server-specific.

For all *Nodes* specified in this document, the *Attributes* named in Table 7 shall be set as specified in the table.

Table 7 – 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* are server-specific.
Description	Optionally a server-specific description is provided.
NodeClass	Shall reflect the *NodeClass* of the *Node.*
NodeId	The *NodeId* is described by *BrowseNames* as defined in 3.3.2.1.
WriteMask	Optionally the *WriteMask Attribute* can be provided. If the *WriteMask Attribute* is provided, it shall set all non-server-specific *Attributes* to not writable. For example, the *Description Attribute* may be set to writable since a *Server* may provide a server-specific description for the *Node*. The *NodeId* shall not be writable, because it is defined for each *Node* in this document.
UserWriteMask	Optionally the *UserWriteMask Attribute* can be provided. The same rules as for the *WriteMask Attribute* apply.
RolePermissions	Optionally server-specific role permissions can be provided.
UserRolePermissions	Optionally the role permissions of the current Session can be provided. The value is server-specific and depends on the *RolePermissions Attribute* (if provided) and the current *Session*.
AccessRestrictions	Optionally server-specific access restrictions can be provided.

3.3.3.2 Objects

For all *Objects* specified in this document, the *Attributes* named in Table 8 shall be set as specified in the Table 8. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 8 – Common Object Attributes

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

3.3.3.3 Variables

For all *Variables* specified in this document, the *Attributes* named in Table 9 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 9 – Common Variable Attributes

Attribute	Value
MinimumSamplingInterval	Optionally, a server-specific minimum sampling interval is provided.
AccessLevel	The access level for *Variables* used for type definitions is server-specific, for all other *Variables* defined in this document, the access level shall allow reading; other settings are server-specific.
UserAccessLevel	The value for the *UserAccessLevel Attribute* is server-specific. It is assumed that all *Variables* can be accessed by at least one user.
Value	For *Variables* used as *InstanceDeclarations,* the value is server-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 behavior is server-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 server-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.

3.3.3.4 VariableTypes

For all *VariableTypes* specified in this document, the *Attributes* named in Table 10 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 10 – Common VariableType Attributes

Attributes	Value
Value	Optionally a server-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 behavior is server-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*.

3.3.3.5 Methods

For all *Methods* specified in this document, the *Attributes* named in Table 11 shall be set as specified in the table. The definitions for the *Attributes* can be found in OPC 10000-3.

Table 11 – Common Method Attributes

Attributes	Value
Executable	All *Methods* defined in this document 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 server-specific. It is assumed that all *Methods* can be executed by at least one user.

4 Device model

4.1 General

Figure 1 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.

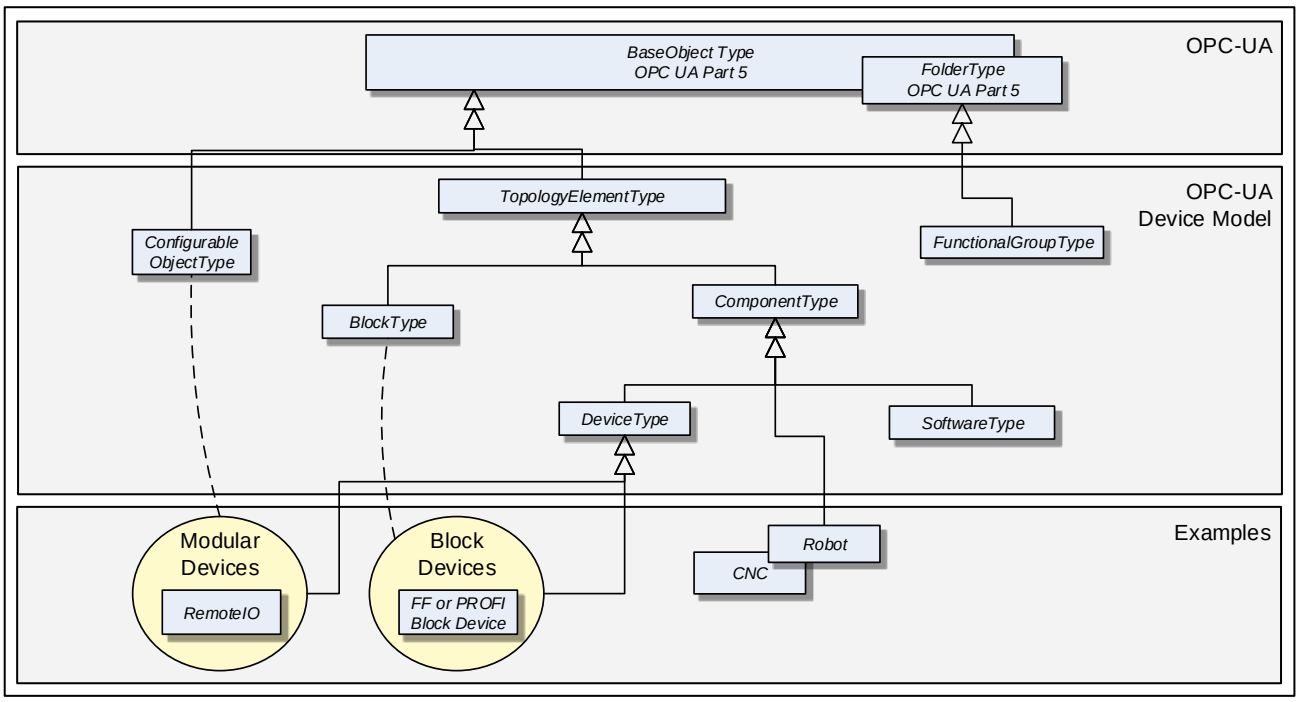

Figure 1 – Device model overview

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 organize 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 organize their modules. Block-oriented *Device*s use it to expose and organize their *Blocks*.

4.2 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.

4.3 TopologyElementType

This *ObjectType* defines a generic model for elements in a device or component topology. Among others, it introduces *FunctionalGroups*, *ParameterSet*, and *MethodSet*. Figure 2 shows the *TopologyElementType*. It is formally defined in Table 12.

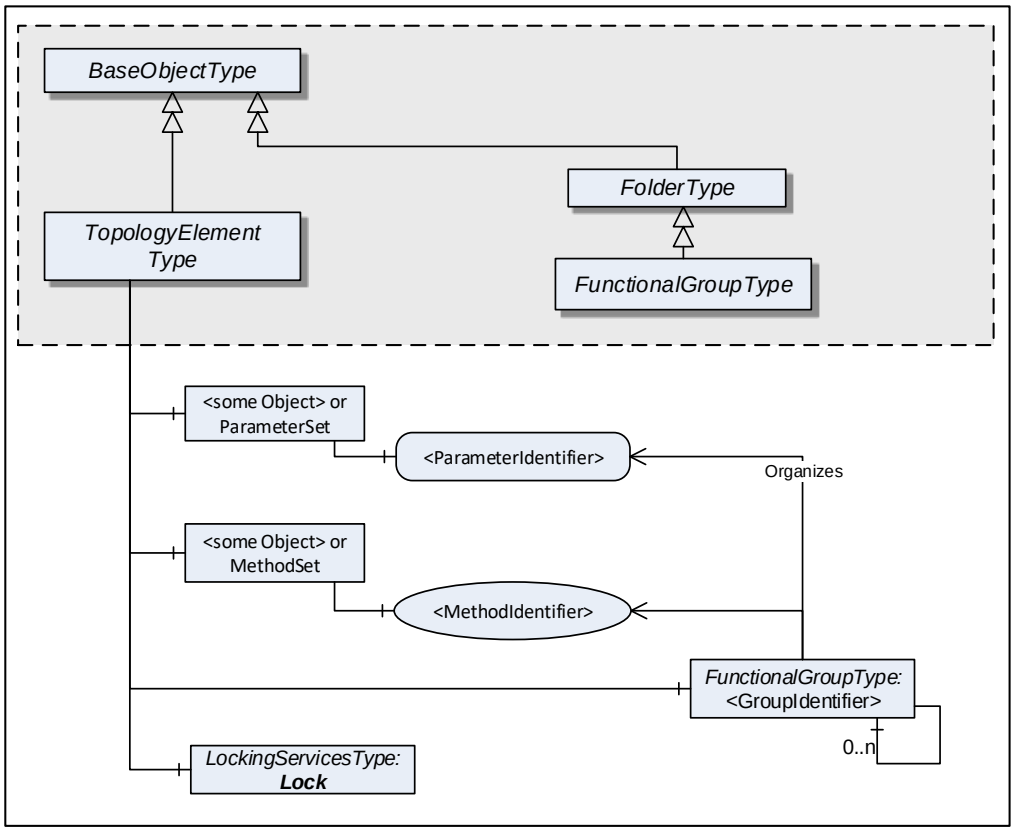

Figure 2 – Components of the TopologyElementType

Table 12 – 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 4.6		
HasSubtype	ObjectType	BlockType	Defined in 4.11		
HasSubtype	ObjectType	ConnectionPointType	Defined in 5.4		
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
Conformance Units					
DI Information Model					

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 4.4.

A *FunctionalGroup* called **Identification** can be used to organize identification information of this *TopologyElement* (see 4.4.2). 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 7).

Clients shall use the *LockingServices* 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 lock applies to the complete *TopologyElement* (including all components such as blocks or modules). *Servers* may expose a Lock Object on a component *TopologyElement* to allow independent locking of components, if no lock is applied to the top-level *TopologyElement*.

If the Online/Offline model is supported (see 6.3), the lock always applies to both the online and the offline version.

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 *MethodSet* is only available if it includes at least one *Method*.

The components of the *TopologyElementType* have additional references as defined in Table 13.

Table 13 – TopologyElementType Additional Subcomponents

Source Path	References	NodeClass	BrowseName	DataType	TypeDefinition	Others
ParameterSet	HasComponent	Variable	<ParameterIdentifier>	BaseDataType	BaseDataVariableType	MandatoryPlaceholder

4.4 FunctionalGroupType

4.4.1 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 4.4.2.

Figure 3 shows the *FunctionalGroupType* components. It is formally defined in Table 14.

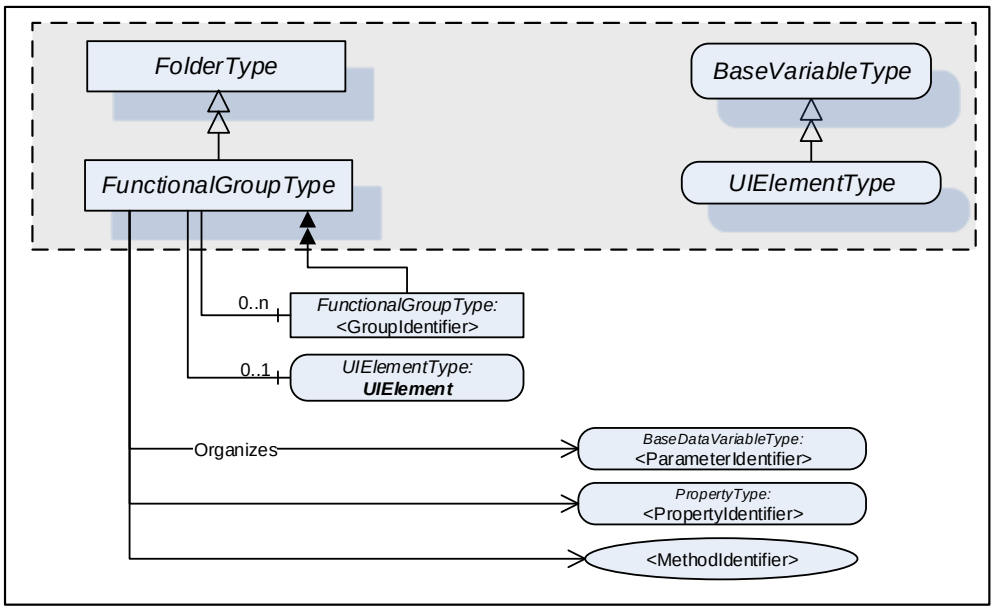

Figure 3 – FunctionalGroupType

Table 14 – 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
Conformance Units					
DI Information Model					

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 4.4.3 for the definition of *UIElements*.

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

4.4.2 Recommended FunctionalGroup BrowseNames

Table 15 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 15 – 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.
OperationCounters	*Parameters* representing numbers of interest when managing a *TopologyElement* while it is operated. Examples are the hours of operation, hours in standby, etc. Those are often the base to calculate KPIs (key performance indicators) like the OEE (overall equipment efficiency). The *Parameters* are domain specific and not defined in this specification. Companion specifications or vendors will define them. This *FunctionalGroup* might be organized into other *FunctionalGroups*, so *Clients* shall expect that they need to browse several hops to get to all *OperationCounters*.
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 organized in a *FunctionalGroup* called **Identification**. See Annex B.1 for an example.

4.4.3 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 16.

Table 16 – 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.					
Conformance Units					
DI Information Model					

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

4.5 Interfaces

4.5.1 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 4 shows the *Interfaces* described in this specification.

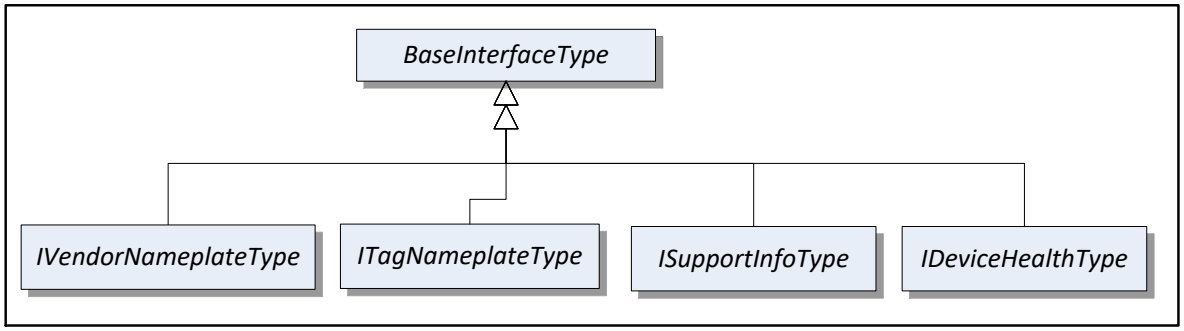

Figure 4 – Overview of Interfaces for Devices and Device components

4.5.2 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 5 and formally defined in Table 17.

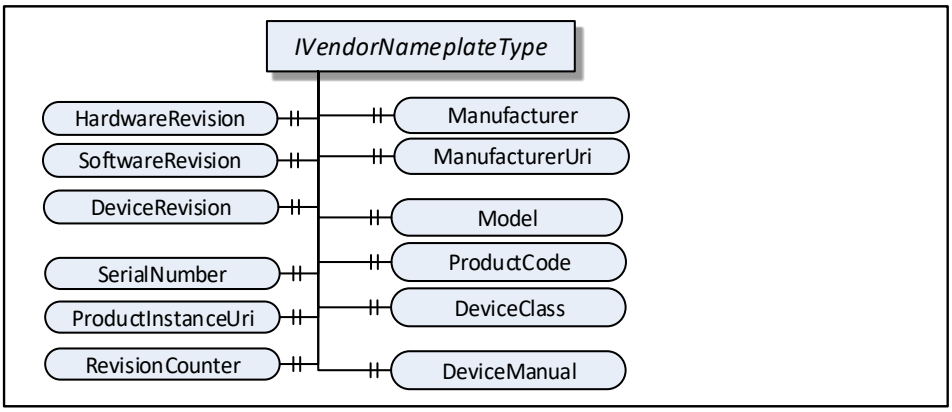

Figure 5 – VendorNameplate Interface

Table 17 – 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
HasProperty	Variable	SoftwareReleaseDate	DateTime	PropertyType	Optional
HasProperty	Variable	PatchIdentifiers	String[]	PropertyType	Optional
Conformance Units					
DI Nameplate					

Product type specific *Properties*:

Manufacturer provides the name of the company that manufactured the item this *Interface* is applied to. *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. *SemanticVersionString* (a sub-type of *String* defined in OPC 10000-5) may be used when using the Semantic Versioning format.

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*. *SemanticVersionString* (a sub-type of *String* defined in OPC 10000-5) may be used when using the Semantic Versioning format.

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*. *SemanticVersionString* (a sub-type of *String* defined in OPC 10000-5) may be used when using the Semantic Versioning format.

DeviceManual allows specifying an address of the user manual. 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 item for which the *Interface* is applied 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, or OPC 10020).

Product instance specific *Properties*:

SerialNumber is a unique production number provided by the manufacturer. 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 provided by the manufacturer. 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 recommended syntax of the *ProductInstanceUri* is: <ManufacturerUri>/<any string> where <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 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.*

SoftwareReleaseDate defines the date when the software is released. If the version information is about patches, this should be the date of the latest patch. It is additional information for the user.

PatchIdentifiers identify the list of patches that are applied to a software version. The format and semantics of the strings are vendor-specific. The order of the strings shall not be relevant.

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

Table 18 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 18 – VendorNameplate Mapping to IRDIs

Property	IRDI
Manufacturer	0112/2///61987#ABA565#007
ManufacturerUri	0112/2///61987#ABN591#001
Model	0112/2///61987#ABA567#007
SerialNumber	0112/2///61987#ABA951#007
HardwareRevision	0112/2///61987#ABA926#006
SoftwareRevision	0112/2///61987#ABA601#006
DeviceRevision	-
RevisionCounter	0112/2///61987#ABN603#001
ProductCode	0112/2///61987#ABA300#006
ProductInstanceUri	0112/2///61987#ABN590#001
DeviceManual	-
DeviceClass	0112/2///61987#ABA566 - type of product

4.5.3 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 6 and formally defined in Table 19.

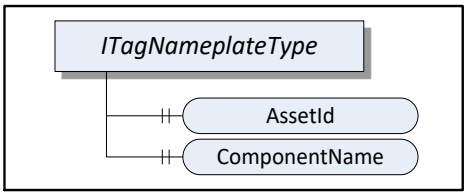

Figure 6 – TagNameplate Interface

Table 19 – 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
Conformance Units					
DI TagNameplate					

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

Table 20 – TagNameplate Mapping to IRDIs

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

4.5.4 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* 7 and formally defined in Table 21.

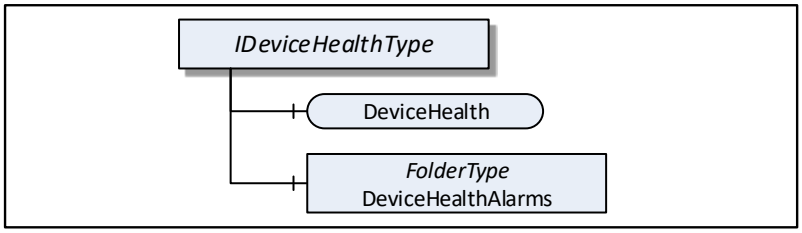

Figure 7 – DeviceHealth Interface

Table 21 – 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
Conformance Units					
DI DeviceHealth					

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 22.

Table 22 – DeviceHealthEnumeration values

Name	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, and 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 4.12.

DeviceHealthAlarms may also be used for other *Alarm* instances that relate to the health condition of the *Device*.

4.5.5 SupportInfo Interface

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 organized into different folders. Each information element is represented by a read-only *Variable*. The information can be retrieved by reading the *Variable* value.

Figure 8 Illustrates the *SupportInfo Interface.* It is formally defined in Table 23.

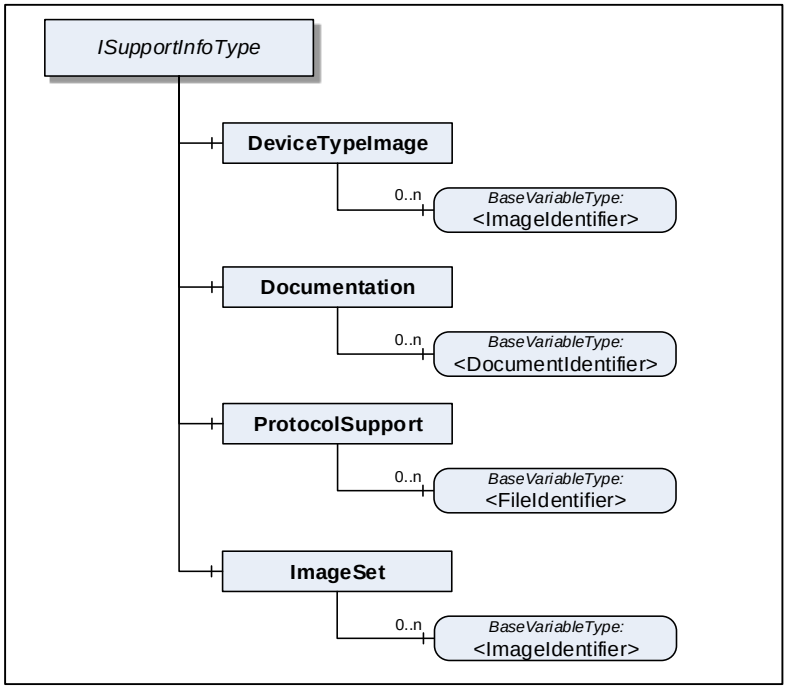

Figure 8 – Support information Interface

Table 23 – 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
Conformance Units					
DI DeviceSupportInfo					

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 components of the *ISupportInfoType* have additional references as defined in Table 24.

Table 24 – ISupportInfoType Additional Subcomponents

Source Path	**References**	**Node Class**	**BrowseName**	**Data Type**	**TypeDefinition**	**Others**
DeviceTypeImage	HasComponent	Variable	<ImageIdentifier>	Image	BaseDataVariableType	MP
Documentation	HasComponent	Variable	<DocumentIdentifier>	ByteString	BaseDataVariableType	MP
ProtocolSupport	HasComponent	Variable	<ProtocolSupportIdentifier>	ByteString	BaseDataVariableType	MP
ImageSet	HasComponent	Variable	<ImageIdentifier>	Image	BaseDataVariableType	MP

Pictures can be exposed as *Variables* organized in the *DeviceTypeImage* folder. There may be multiple images of different resolutions. Each image is a separate *Variable*.

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).

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.

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".

Protocol support files are exposed as *Variables* organized in the *ProtocolSupport* folder. They may represent various types of information as defined by a protocol. Examples are a GSD or a CFF file.

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.

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

4.6 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 *VendorNameplate* and the *TagNameplate Interface*. Figure 9 Illustrates the *ComponentType.* It is formally defined in Table 25.

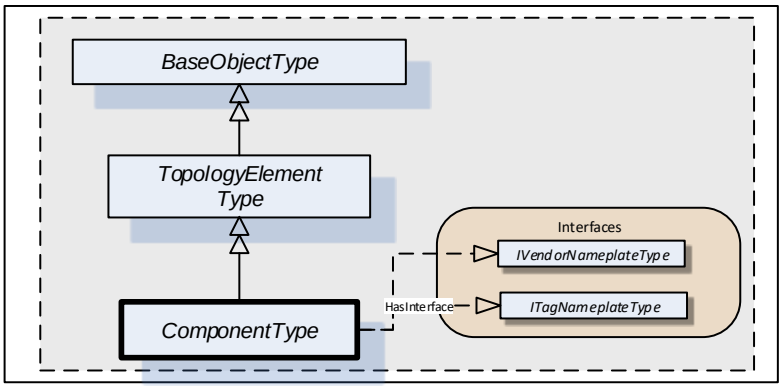

Figure 9 – ComponentType

Table 25 – ComponentType definition

Attribute	Value				
BrowseName	ComponentType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *TopologyElementType* defined in 4.3.					
HasSubtype	ObjectType	DeviceType	Defined in 4.7.		
HasSubtype	ObjectType	SoftwareType	Defined in 4.8.		
HasInterface	ObjectType	IVendorNameplateType	Defined in 4.5.2.		
HasInterface	ObjectType	ITagNameplateType	Defined in 4.5.3.		
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
Conformance Units					
DI Information Model					

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 4.5.2.

ITagNameplateType and its members are described in 4.5.3.

4.7 DeviceType

This *ObjectType* can be used to define the structure of a *Device*. Figure 10 shows the *DeviceType*. It is formally defined in Table 26.

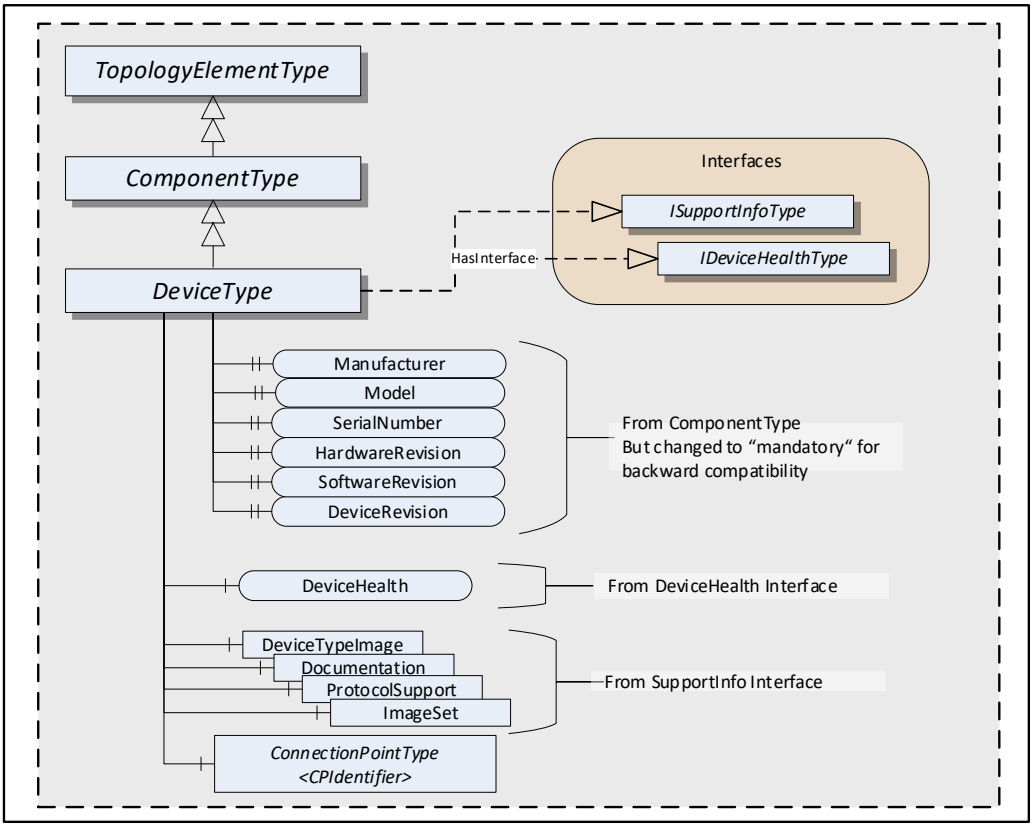

Figure 10 – DeviceType

Table 26 – DeviceType definition

Attribute	**Value**				
BrowseName	DeviceType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *ComponentType* defined in 4.6					
HasInterface	ObjectType	ISupportInfoType		Defined in 4.5.3.	
HasInterface	ObjectType	IDeviceHealthType		Defined in 4.5.3.	
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
Conformance Units					
DI DeviceType					

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 5.4) 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 4.5. 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.

4.8 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 11 Illustrates the *SoftwareType.* It is formally defined in Table 27.

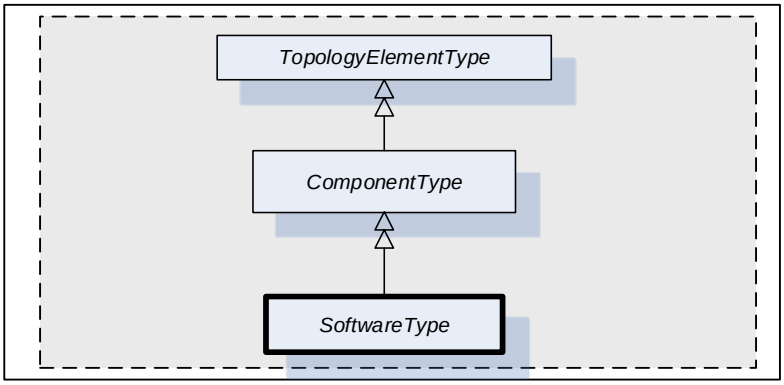

Figure 11 – SoftwareType

Table 27 – SoftwareType definition

Attribute	**Value**				
BrowseName	SoftwareType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *ComponentType* defined in 4.6.					
HasProperty	Variable	Manufacturer	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	Model	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	SoftwareRevision	String	PropertyType	Mandatory
Conformance Units					
DI Software Component					

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.

4.9 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 12.

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 organized as sub-components of the root instance.

Figure 12 shows the *AddressSpace* organisation with this standard entry point and examples.

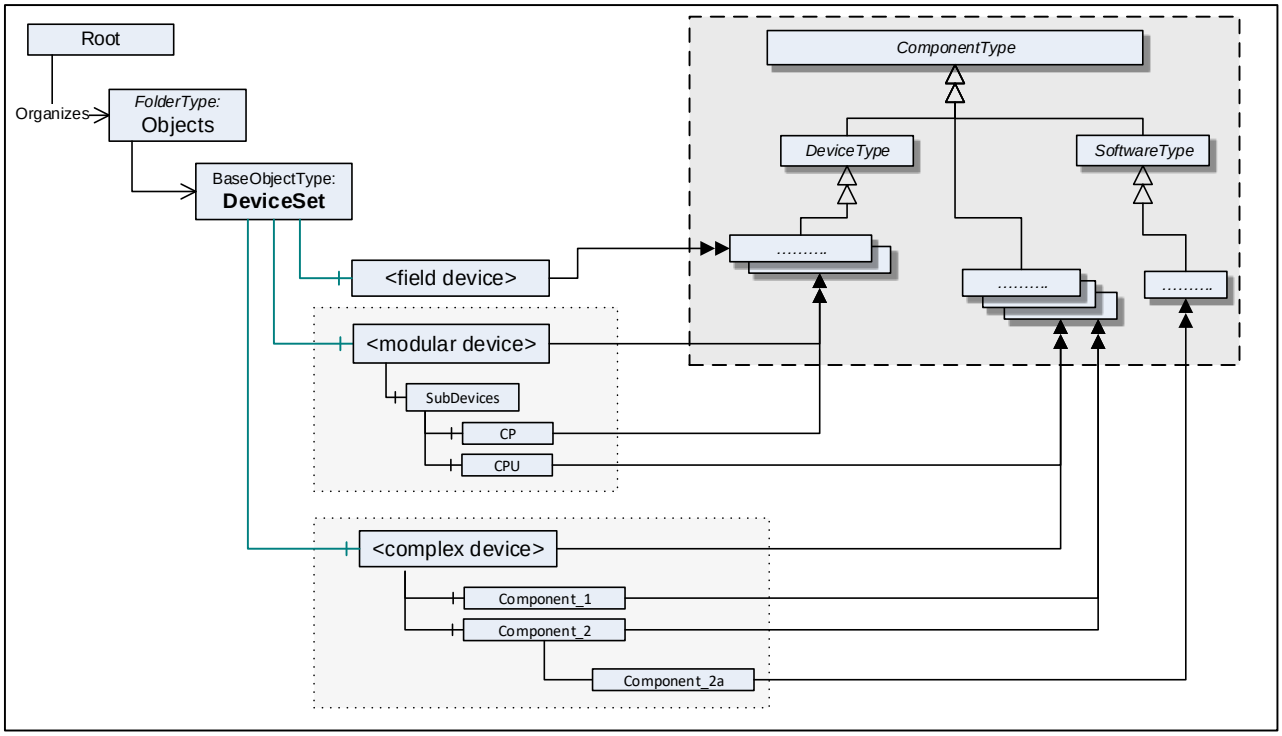

Figure 12 – Standard entry point for Devices

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	
Conformance Units			
DI DeviceSet			

4.10 DeviceFeatures entry point

The *DeviceFeatures Object* can be used to organize other functional entities that are related to the *Devices* referenced by the *DeviceSet*. Companion specifications may standardize such instances and their *BrowseNames*. Figure 13 shows the *AddressSpace* organisation with this standard entry point.

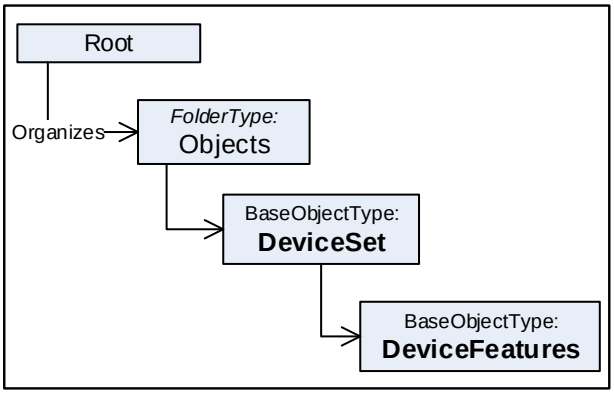

Figure 13 – Standard entry point for DeviceFeatures

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 4.9			
HasTypeDefinition	ObjectType	BaseObjectType	
Conformance Units			
DI DeviceSet			

4.11 BlockType

This *ObjectType* defines the structure of a *Block Object*. Figure 14 depicts the *BlockType* hierarchy. It is formally defined in Table 30.

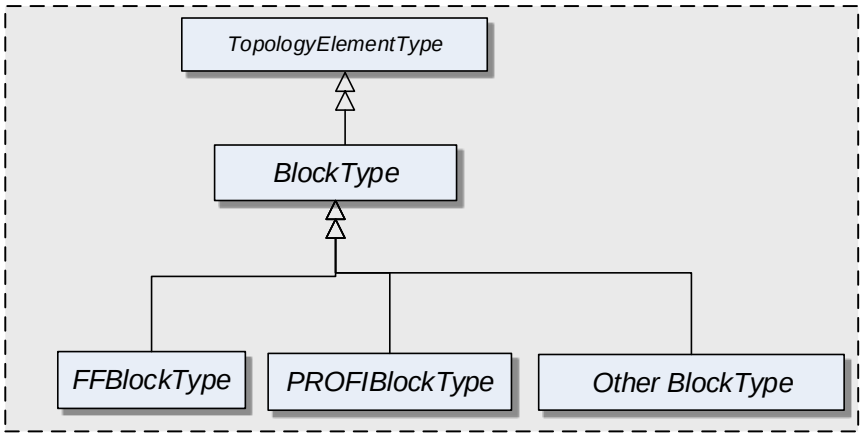

Figure 14 – BlockType hierarchy

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 4.2					
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
Conformance Units					
DI Blocks					

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.

4.12 DeviceHealth Alarm Types

4.12.1 General

The DeviceHealth Property defined in 4.5.4 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 15 informally describes the *AlarmTypes* for DeviceHealth.

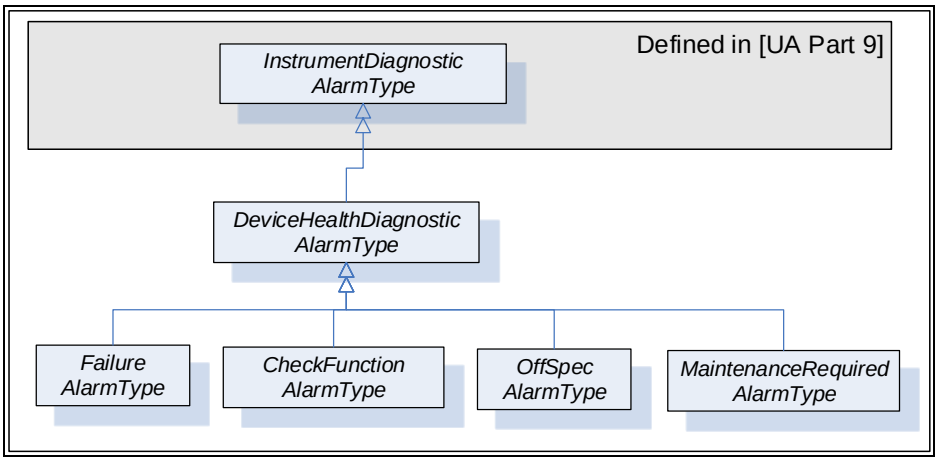

Figure 15 – Device Health Alarm type hierarchy

4.12.2 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	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the InstrumentDiagnosticAlarmType defined in OPC 10000-9.					
HasSubtype	ObjectType	FailureAlarmType	Defined in clause 4.12.3		
HasSubtype	ObjectType	CheckFunctionAlarmType	Defined in clause 4.12.4		
HasSubtype	ObjectType	OffSpecAlarmType	Defined in clause 4.12.5		
HasSubtype	ObjectType	MaintenanceRequiredAlarmType	Defined in clause 4.12.6		
Conformance Units					
DI HealthDiagnosticsAlarm					

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.

4.12.3 FailureAlarmType

The *FailureAlarmType* is formally defined in Table 32. For description of the FAILURE state see Table 22.

Table 32 – FailureAlarmType definition

Attribute	Value				
BrowseName	FailureAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DeviceHealthDiagnosticAlarmType defined in 4.12.2.					
Conformance Units					
DI HealthDiagnosticsAlarm					

4.12.4 CheckFunctionAlarmType

The *CheckFunctionAlarmType* is formally defined in Table 33. For description of the CHECK_FUNCTION state see Table 22.

Table 33 – CheckFunctionAlarmType definition

Attribute	Value				
BrowseName	CheckFunctionAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DeviceHealthDiagnosticAlarmType defined in 4.12.2.					
Conformance Units					
DI HealthDiagnosticsAlarm					

4.12.5 OffSpecAlarmType

The *OffSpecAlarmType* is formally defined in Table 34. For description of the OFF_SPEC state see Table 22.

Table 34 – OffSpecAlarmType definition

Attribute	Value				
BrowseName	OffSpecAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DeviceHealthDiagnosticAlarmType defined in 4.12.2.					
Conformance Units					
DI HealthDiagnosticsAlarm					

4.12.6 MaintenanceRequiredAlarmType

The *MaintenanceRequiredAlarmType* is formally defined in Table 35. For description of the MAINTENANCE_REQUIRED state see Table 22.

Table 35 – MaintenanceRequiredAlarmType definition

Attribute	Value				
BrowseName	MaintenanceRequiredAlarmType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the DeviceHealthDiagnosticAlarmType defined in 4.12.2.					
Conformance Units					
DI HealthDiagnosticsAlarm					

5 Device communication model

5.1 General

Clause 5 introduces *References*, the *ProtocolType*, and basic *TopologyElementTypes* needed to create a communication topology. The types for this model are illustrated in Figure 16.

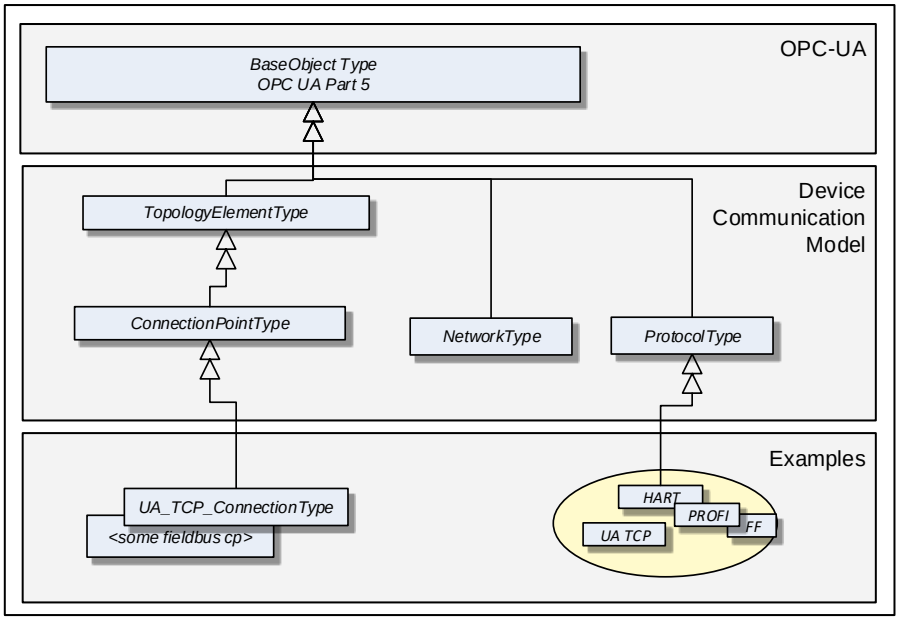

Figure 16 – Device communication model overview

A *ProtocolType ObjectType* represents a specific communication protocol (e.g. *FieldBus*) implemented by a certain *TopologyElement*. Examples are shown in Figure 18.

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

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

Figure 17 provides an overall example.

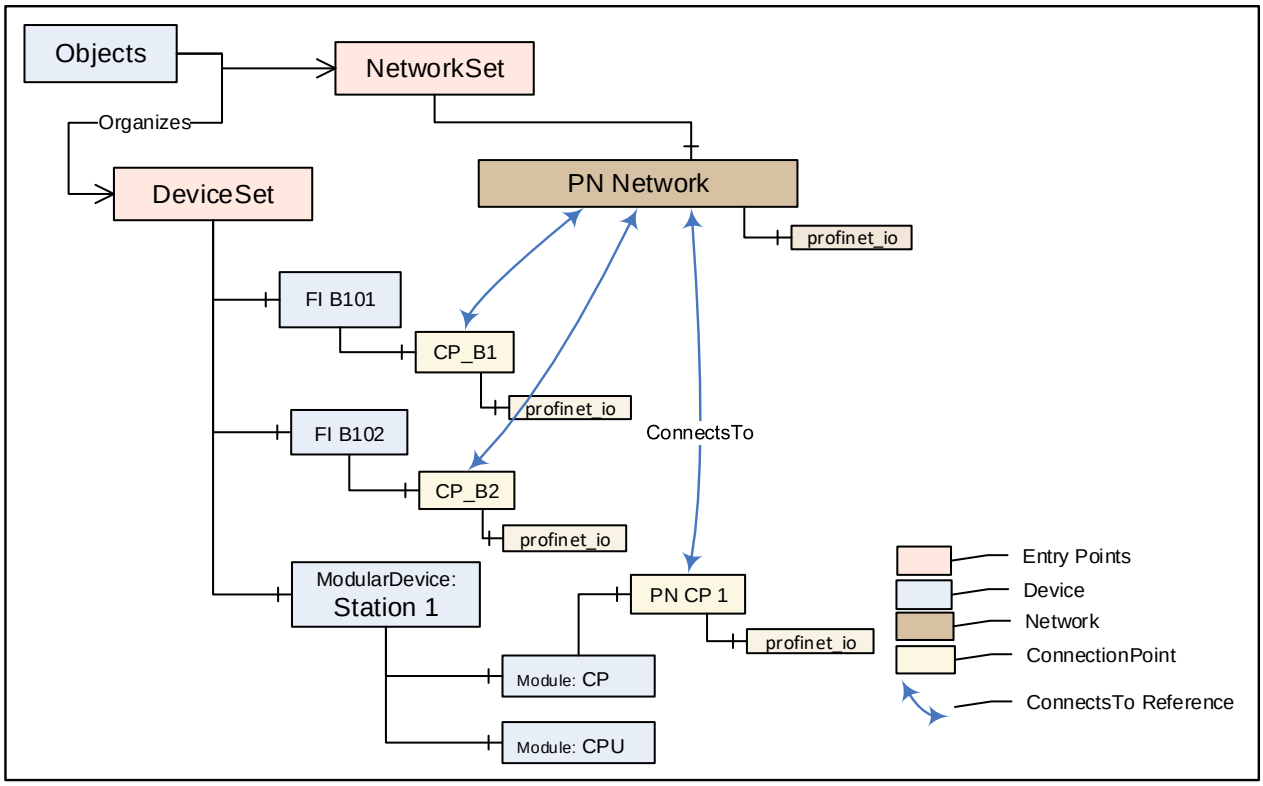

Figure 17 – Example of a communication topology

5.2 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 18).

Figure 18 shows the *ProtocolType* including some specific types and instances that represent *Communication Profiles* of that type. It is formally defined in Table 36.

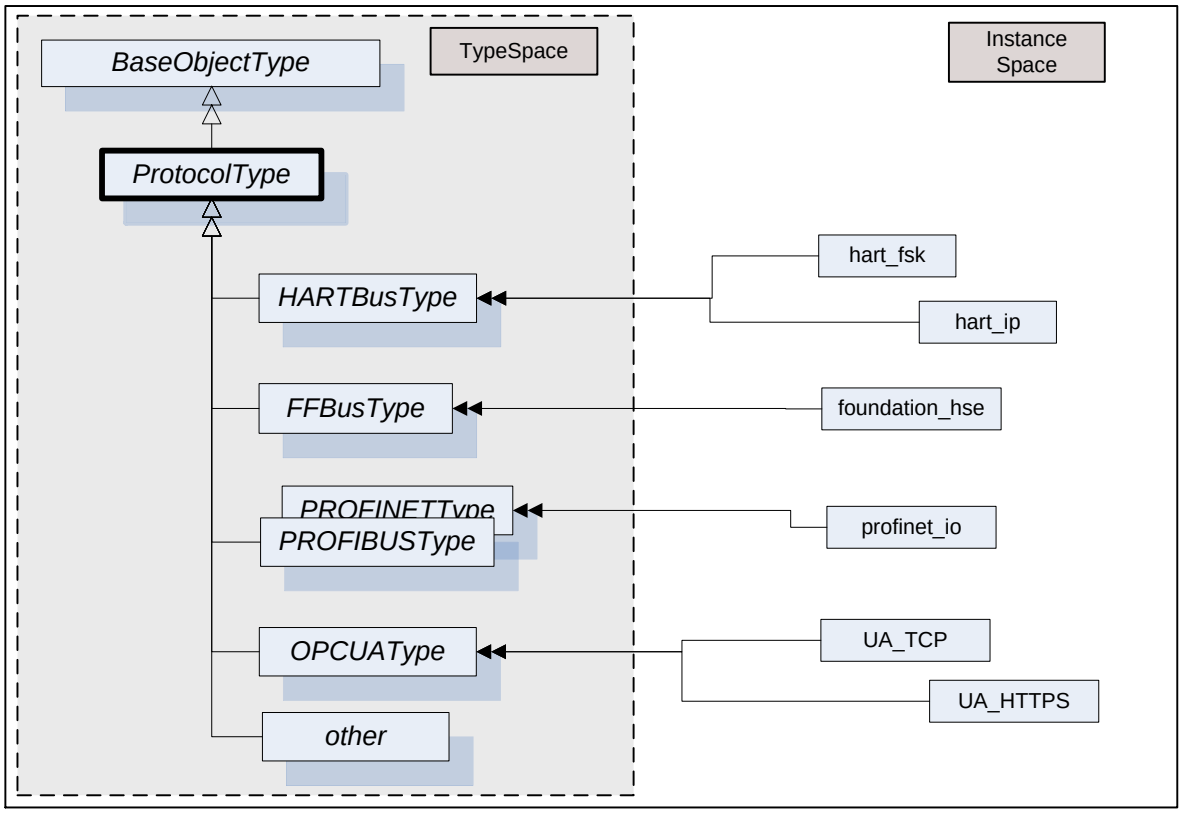

Figure 18 – Example of a ProtocolType hierarchy with instances that represent specific communication profiles

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					
Conformance Units					
DI Network					
DI Protocol					

5.3 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 19 shows the type hierarchy and the *NetworkType* components. It is formally defined in Table 37.

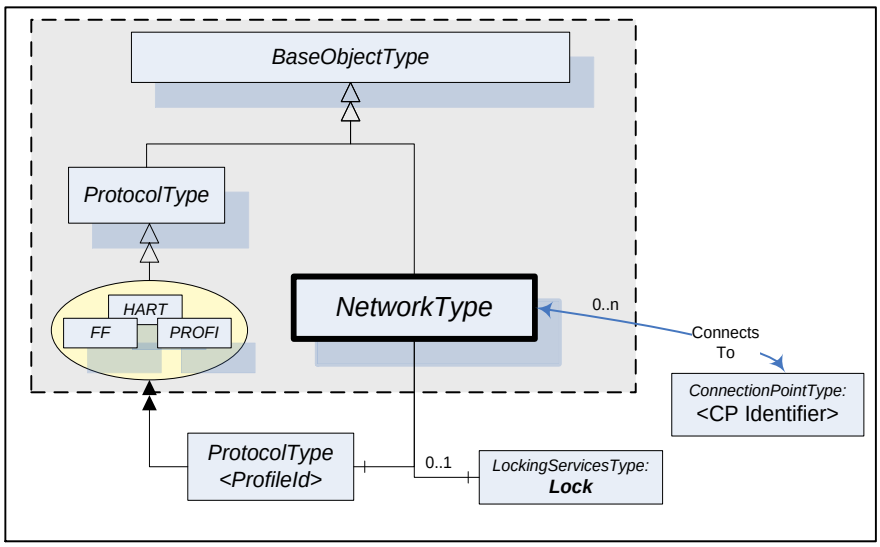

Figure 19 – NetworkType

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
Conformance Units					
DI Network					

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 22 for a usage example. They represent the protocol-specific access points for the connected *Devices*.

In addition, *Networks* may also support *LockingServices* (defined in 7).

Clients shall use the *LockingServices* 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 *Network* is avoiding concurrent topology changes.

The lock on a *Network* applies to the *Network*, all connected *TopologyElements* and their components. 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.

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.

If the Online/Offline model is supported (see 6.3), the lock always applies to both the online and the offline version.

5.4 ConnectionPoint

This *ObjectType* represents the logical interface of a *Device* to a *Network*. A specific subtype shall be defined for each protocol. Figure 20 shows the *ConnectionPointType* including some specific types.

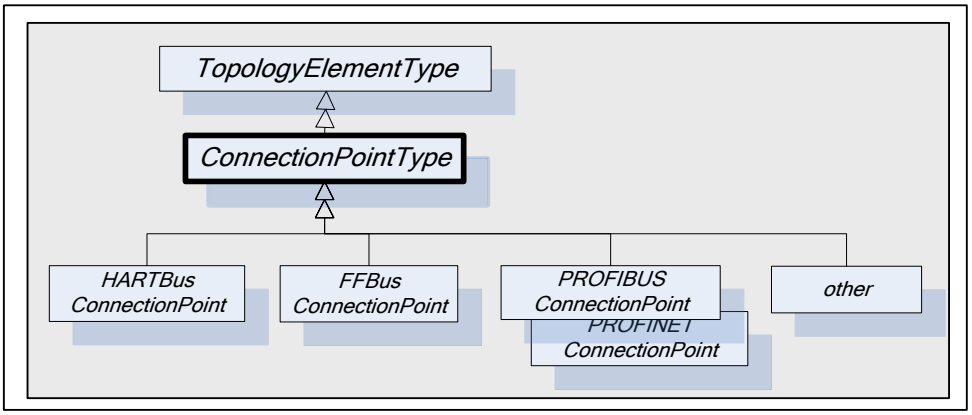

Figure 20 – Example of ConnectionPointType hierarchy

A *Device* can have more than one such interface to the same or to different *Networks*. Different interfaces usually exist for different protocols. Figure 21 shows the *ConnectionPointType* components. It is formally defined in Table 38.

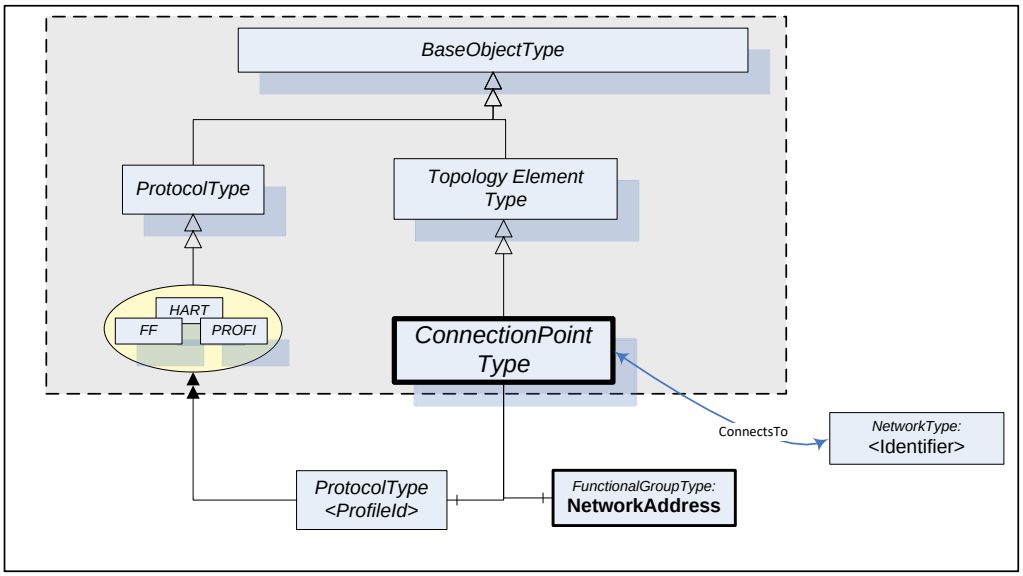

Figure 21 – ConnectionPointType

Table 38 – ConnectionPointType definition

Attribute	Value				
BrowseName	ConnectionPointType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the TopologyElementType defined in 4.2.					
HasComponent	Object	NetworkAddress		FunctionalGroupType	Mandatory
HasComponent	Object	<ProfileIdentifier>		ProtocolType	MandatoryPlaceholder
ConnectsTo	Object	<NetworkIdentifier>		NetworkType	OptionalPlaceholder
Conformance Units					
DI ConnectionPoint					

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 5.2. 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 22 illustrates some usage models.

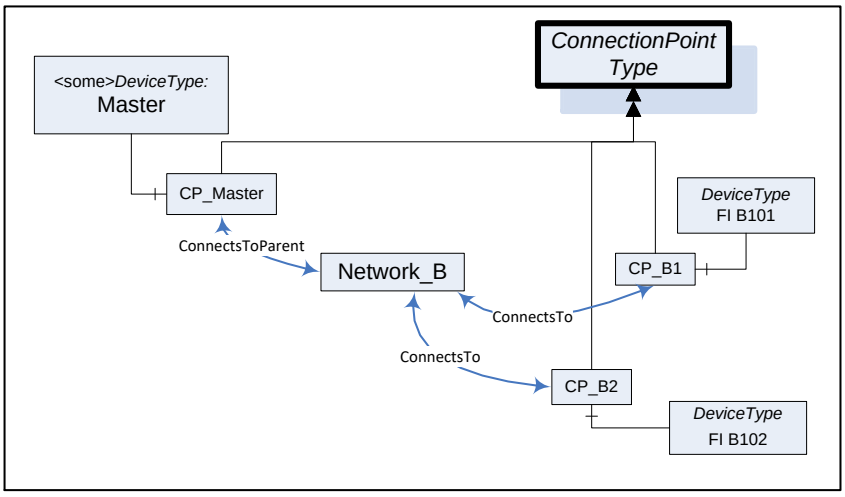

Figure 22 – ConnectionPoint usage

5.5 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 23.

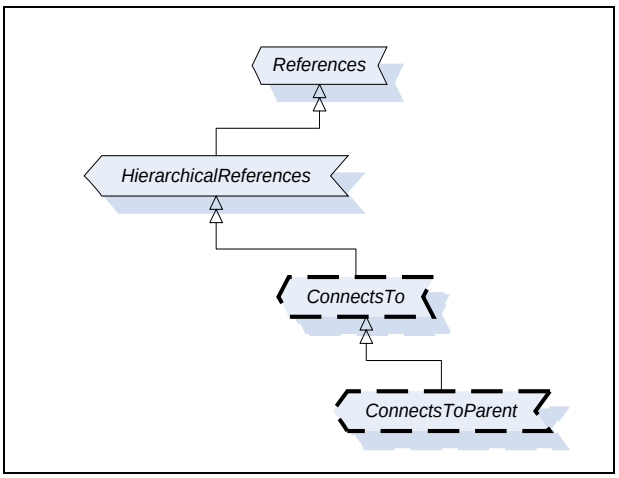

Figure 23 – Type Hierarchy for ConnectsTo and ConnectsToParent References

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.			
Conformance Units			
DI ConnectsTo			

Table 40 – ConnectsToParent ReferenceType

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

Figure 24 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*.

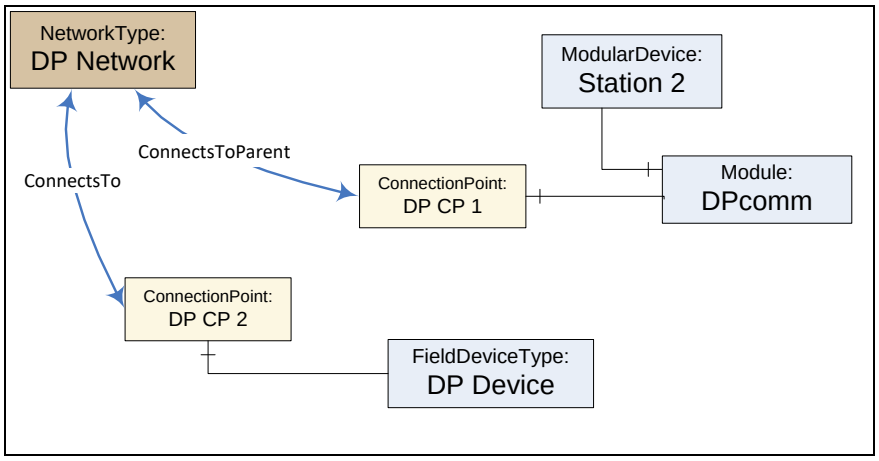

Figure 24 – Example with ConnectsTo and ConnectsToParent References

5.6 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	
Conformance Units			
DI NetworkSet			

6 Device integration host model

6.1 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 25 and Figure 26 illustrate an example configuration and the configured topology as it will appear in the *Server AddressSpace* (details left out).

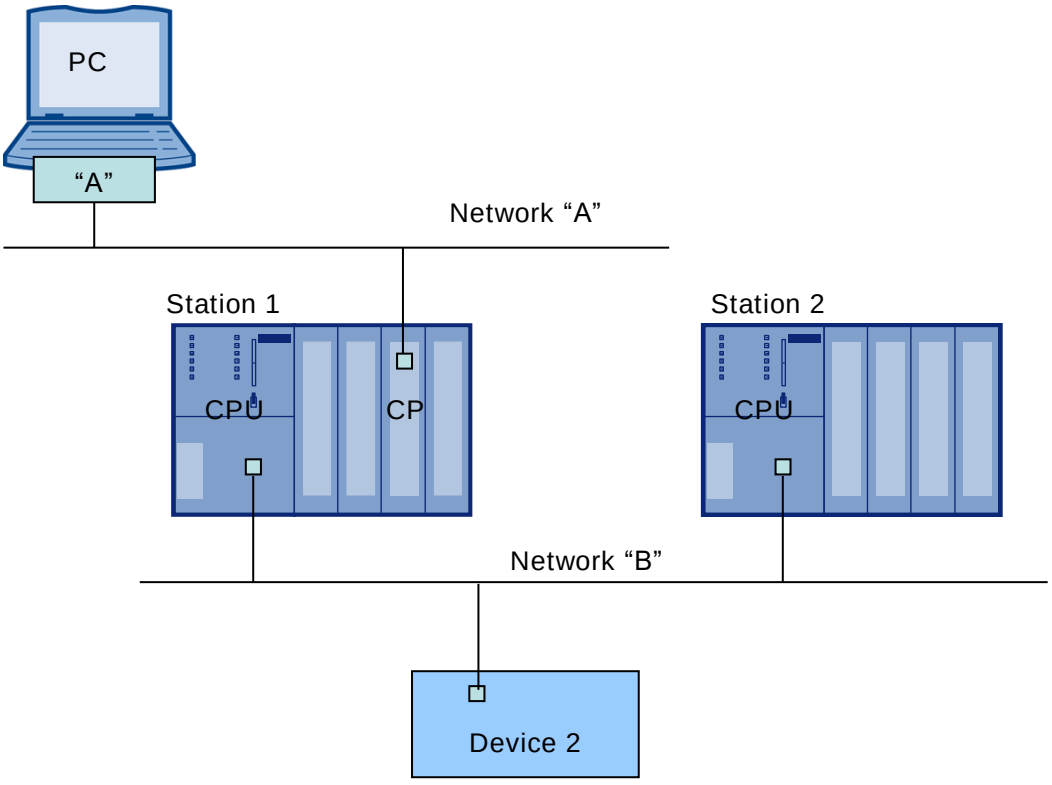

Figure 25 – Example of an automation system

The PC in Figure 25 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.

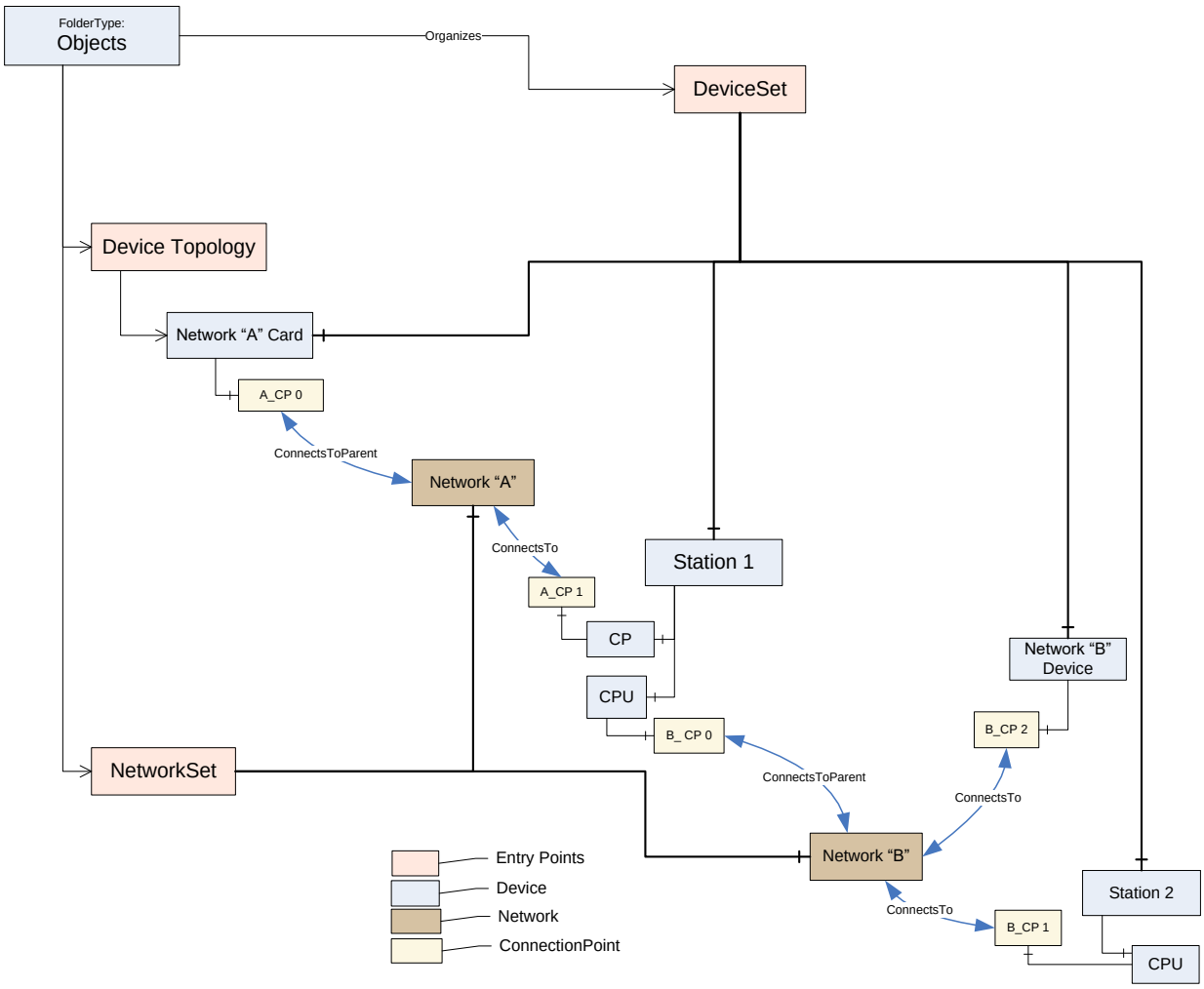

Figure 26 – Example of a Device topology

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

Entry points assure common behavior across different implementations:

- *DeviceTopology*: Starting node for the topology configuration. See 6.2.
- *DeviceSet*: See 4.9.
- *NetworkSet*: See 5.6.

6.2 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 organize 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
Conformance Units				
DI DeviceTopology				

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.

6.3 Online/Offline

6.3.1 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 27.

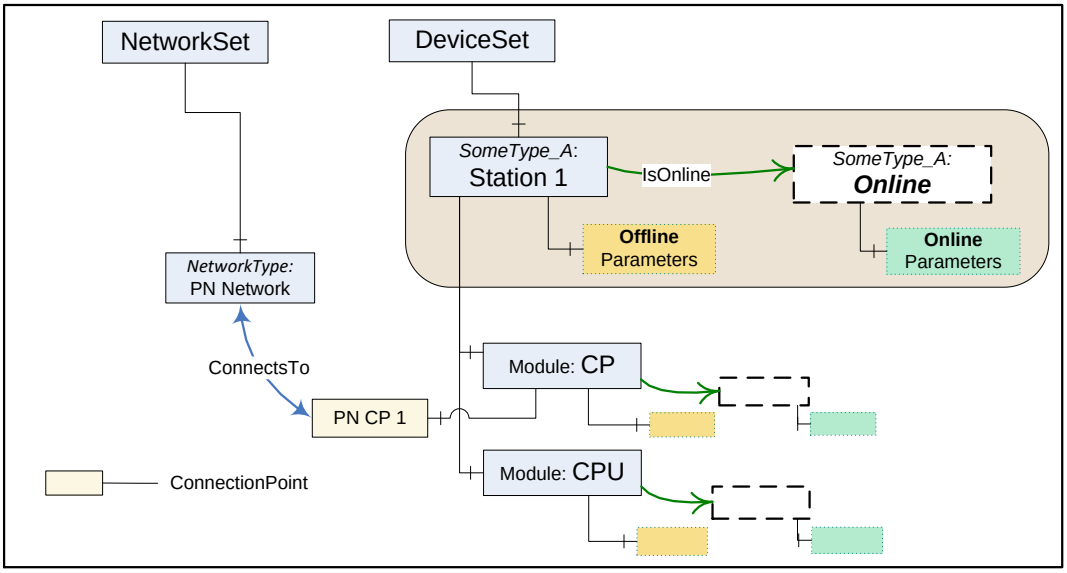

Figure 27 – Online component for access to Device data

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 27. 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 6.4.2.

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* organizes *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.

6.3.2 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 28. Its representation in the *AddressSpace* is specified in Table 43.

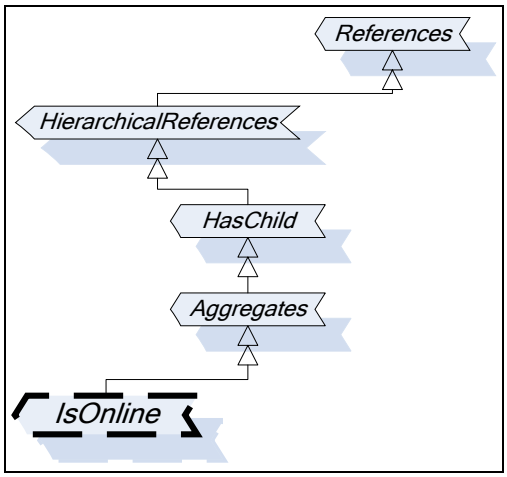

Figure 28 – Type hierarchy for IsOnline Reference

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.			
Conformance Units			
DI Offline			

6.4 Offline-Online data transfer

6.4.1 Definition

The "Online-offline data transfer" is based on the *AddIn* model specified in OPC 10001-7.

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 7).

6.4.2 TransferServices Type

The *TransferServicesType* provides the *Methods* needed to transfer data to and from the online *Device*. Figure 29 shows the *TransferServicesType* definition. It is formally defined in Table 44.

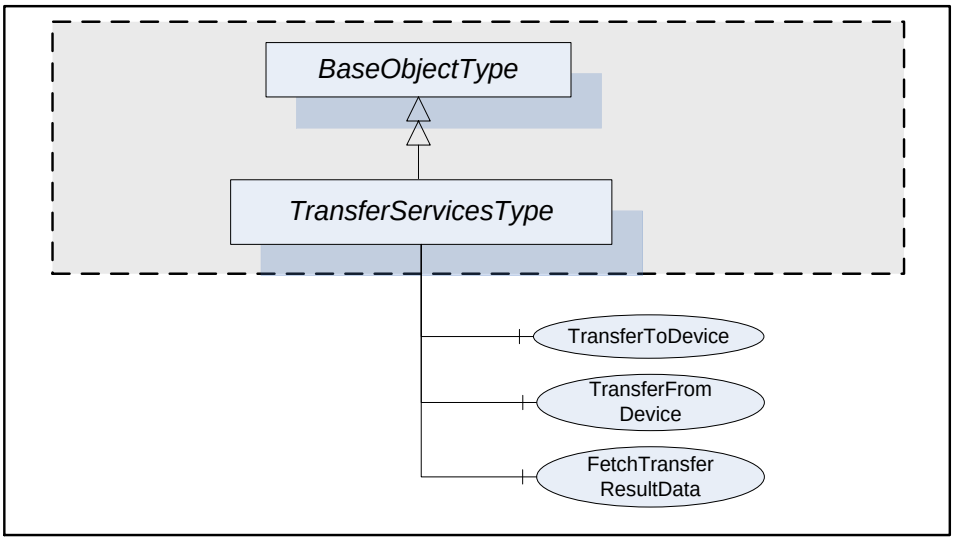

Figure 29 – TransferServicesType

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
Conformance Units					
DI Offline					

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*.

6.4.3 TransferServices Object

The support of *TransferServices* for an *Object* is declared by aggregating an instance of the *TransferServicesType* as illustrated in Figure 30.

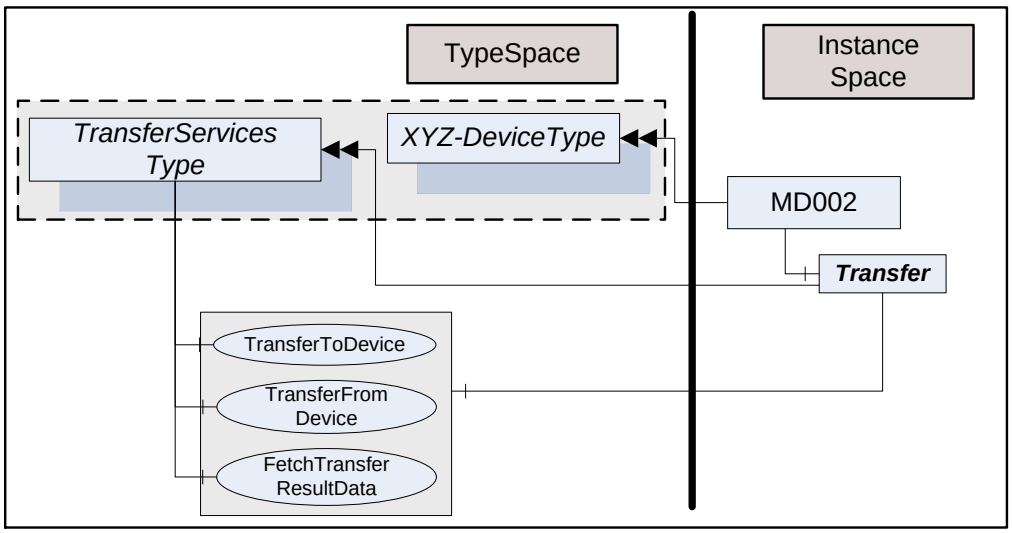

Figure 30 – TransferServices

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*.

6.4.4 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

6.4.5 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

6.4.6 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	ParameterResult DataType []	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.

7 Locking model

7.1 Overview

The following Locking feature is based on the *AddIn* model specified in OPC 10001-7.

Locking is the means to avoid concurrent modifications to an *Object* by restricting access to the entity (often a *Client* but could also be an internal process) that initiated the lock. *LockingServices* are typically used 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 context of the lock is specific to the *ObjectType* where it is applied to (subsequently named "lock-owner"). These specifics need to be described as part of this lock-owner *ObjectType*. See for example the section on lock in the *TopologyElement* (clause 4.3) and the *Network* (clause 5.3).

By default, a lock allows other *Applications* to view (navigate/read) the locked element. However, *Servers* may choose to implement an exclusive locking where other *Applications* have no access at all (e.g. in cases where even read operations require certain settings to Variables).

7.2 LockingServices Type

The *LockingServicesType* provides *Methods* to manage the lock and *Properties* with status information. This section describes the common semantic. The lock-owner *ObjectTypes* will often extend these semantics.

Figure 31 shows the *LockingServicesType* definition. It is formally defined in Table 54.

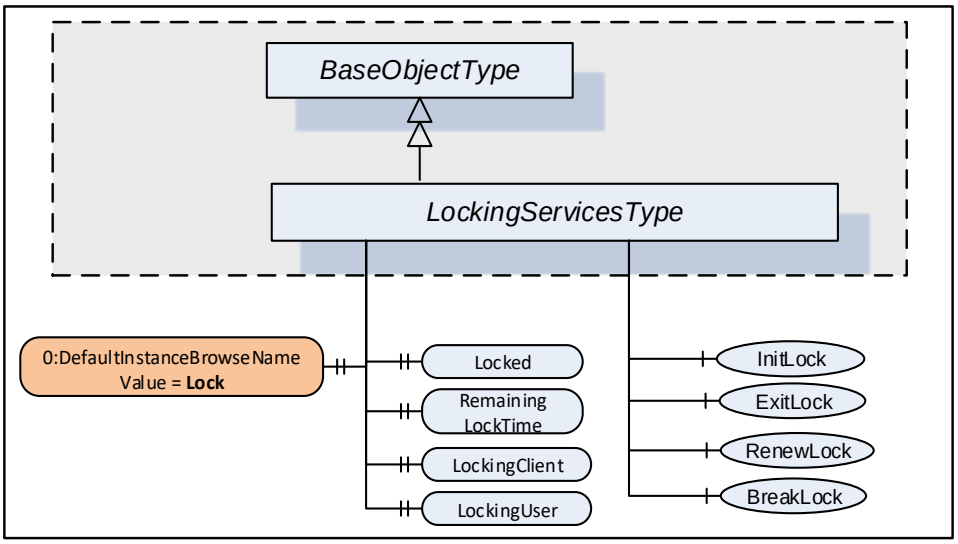

Figure 31 – LockingServicesType

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	Defined in 7.5		Mandatory
HasComponent	Method	RenewLock	Defined in 7.7		Mandatory
HasComponent	Method	ExitLock	Defined in 7.6		Mandatory
HasComponent	Method	BreakLock	Defined in 7.8		Mandatory
HasProperty	Variable	0:DefaultInstanceBrowseName	QualifiedName	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
Conformance Units					
DI Locking					

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*. Its Value is defined in Table 55.

Table 55 – LockingServicesType Additional Variable Attributes

Source Path	Value
0:DefaultInstanceBrowseName	Lock

A lock is typically initiated by a *Client* calling the *InitLock Method* and removed by calling the *ExitLock Method*. The lock-owner *ObjectTypes* can define mechanisms that automatically initiate and remove a lock.

A lock request will be rejected if operations are active that will be prevented by the lock.

The lock is automatically removed if the *MaxInactiveLockTime* has elapsed (see 7.4). The lock is also removed when the *Session* ends during inactivity. This is typically the case when the connection to the *Client* breaks and the *Session* times out.

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

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

When the lock is initiated by a *Client, LockingClient* contains the ApplicationUri of the *Client* as provided in the CreateSession *Service* call (see OPC 10000-4). Other options to get this information can be specified on the lock-owner *ObjectType*.

LockingUser contains information to identify the user. When the lock is initiated by a *Client* it is obtained directly or indirectly from the UserIdentityToken passed by the *Client* in the ActivateSession *Service* call (see OPC 10000-4). Other options to get this information can be specified on the lock-owner *ObjectType*.

RemainingLockTime denotes the remaining time in milliseconds after which the lock will automatically be removed by the *Server*. This time is based upon *MaxInactiveLockTime* (see 7.4).

7.3 LockingServices Object

The support of *LockingServices* for an *Object* is declared by aggregating an instance of the *LockingServicesType* as illustrated in Figure 32.

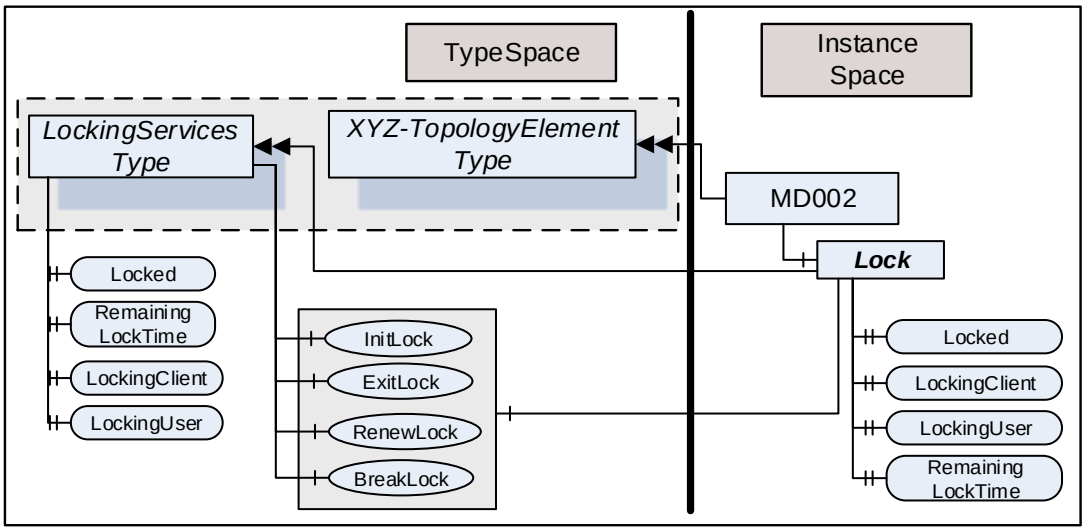

Figure 32 – LockingServices

This *Object* is used as container for the *LockingServices Methods* and *Properties* and should have the *BrowseName* **Lock**. It shall be referenced using *HasComponent* or *HasAddIn* from the lock-owner *Object* (for example, a *Device*).

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

7.4 MaxInactiveLockTime Property

The *MaxInactiveLockTime Property* shall be added to the *ServerCapabilities Object* (see OPC 10000-5). It contains a *Server*-specific period of inactivity in milliseconds after which the *Server* will revoke the lock.

The *Server* will initiate a timer based on this time as part of processing the *InitLock* request and after the last activity caused by the initiator of the lock is finished. Calling the *RenewLock Method* shall reset the timer.

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

The *MaxInactiveLockTime Property* is formally defined in Table 56.

Table 56 – MaxInactiveLockTime Property definition

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

7.5 InitLock Method

InitLock restricts access for other UA *Applications*.

A call of this *Method* for an element that is already locked will be rejected..

While locked, requests from other *Applications* to modify the locked element (e.g., writing to *Variables*, 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 *Applications* have no access at all.

The lock is removed when *ExitLock* is called. It is automatically removed when the *MaxInactiveLockTime* elapsed (see 7.4).

The signature of this *Method* is specified below. Table 57 and Table 58 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 57 – 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 58 – 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

7.6 ExitLock Method

ExitLock removes the lock. This *Method* may only be called from the same *Application* which initiated the lock.

The signature of this *Method* is specified below. Table 59 and Table 60 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 59 – ExitLock Method Arguments

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

Table 60 – ExitLock Method AddressSpace definition

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

7.7 RenewLock Method

The lock timer is automatically renewed whenever the initiator of the lock issues 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 removing the lock. This *Method* may only be called from the same *Application* which initiated the lock.

The signature of this Method is specified below. Table 61 and Table 62 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 61 – RenewLock Method Arguments

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

Table 62 – RenewLock Method AddressSpace definition

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

7.8 BreakLock Method

BreakLock allows a *Client* (with sufficiently high user rights) to break the lock. 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 63 and Table 64 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 63 – BreakLock Method Arguments

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

Table 64 – BreakLock Method AddressSpace definition

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

8 Software update model

8.1 Overview

The software update model defined in this clause is used to manage the software of a *Device*. This can include the installation of new software, the update of existing software, the update of a firmware and a limited backup and restore of parameters and firmware as far as it is needed for the update. The specific steps to perform the actual installation are only known by the device. They are not exposed by this *Information Model*.

The use cases that were considered for this *Information Model* are described in 8.2. Several options that can be combined for a concrete *SoftwareUpdateType* instance are described in 8.3. Valid combinations of these options are defined in the profiles section. 8.3.5 describes how to implement a *Software Update Client* that has to deal with several options. The types for this model are formally defined in 8.4 and 8.5.

8.2 Use Cases

The software update model is used in several scenarios. The following subsections list common use cases that are considered by this model. There are also some use cases that that are not covered. A future version might add features for them.

8.2.1 Supported Use Cases

8.2.1.1 Software Update of constraint devices

The model is intended to be applicable across devices with varying resources and constraints. This is achieved e.g. by various options for the server implementation (see 8.3.4).

8.2.1.2 Update Devices from different manufacturers with a Software Update Client

Allow devices to be updated via *Software Update Client* software. To address the domain specific constraints this can be a domain-specific client software (In the manufacturing domain a machine often needs to be stopped before the update, whereas in process domain e.g. a redundant device needs to be activated). 8.3.5 describes the workflow of a *Software Update Client*.

8.2.1.3 Update of underlying Devices (e.g. IO Link Devices)

Software update is applicable for any device or software component that is exposed in the *Server* address space. This can also represent other devices that are just connected to the device hosting the *Server*. This can be done using the *AddIns* described in 8.3.11.

8.2.1.4 Coordinated update of multiple Devices in a machine / plant

When updating several connected devices in a machine or plant the devices might first need to be set into a special “state” where they wait for the start of the update and don’t start operating again. After that the updates can be installed in an order defined by the *Client* (e.g. sensors first, switches last). Finding the best sequence is task of the client implementation or the operator and not in scope of this specification.

The “state” is defined depending on the type of machine / plant. For factory automation this normally means that production and the software on the devices is stopped. For a sensor in process automation this could mean that a replacement value is configured in the controller for the value measuered in the device. If a controller needs to be updated in process automation it often needs to be the passive part of a redundant set of controllers.

A *Client* also needs to consider the proper sequence when updating the devices. For example, if parts of the network become unreachable due to the update of an infrastructure device.

A server can support the prepare for update option (8.3.4.2) to enable this use case.

8.2.1.5 Partial update without stopping the software

For some updates it is not necessary to stop the software. This could be the case if parts of a software are replaced that are currently not used or if new software is installed. Whether an update can be installed like this is only known by the device and depends on the concrete update file. To support this, the *Client* can read the *UpdateBehavior* (8.5.2) to determine if stopping is required.

8.2.1.6 Scheduled update

In some cases, it is required to prepare the update and then plan the start for a later time or under some strategic conditions. In this case the software is transferred to the device first. Later (e.g. at the end of the shift or on the weekend) and under specific conditions (e.g. nothing to produce) the update *Client* can start the update. In this scenario the time and the conditions are known and checked by the update *Client*, not by the *Server*, so for the use of the software update options an established *Client-Server* connection is required. The scheduling is a task of the *Client* and not described in this specification.

8.2.1.7 Central distribution for later installation

It should be possible to distribute the software to several devices without actual installation. In this scenario a central tool can determine the required updates and distribute them to all devices. The actual installation can then be started later by a different *Client*. This is realized by separating the transfer (8.4.1.2) from the installation (8.3.4.6).

8.2.1.8 Update of individual parts of a software

Depending on the device there should be several options to partition a software. For example, it should be possible to structure the firmware of a device in a way that each part can be updated individually. Additionally, software update should be applicable to the firmware of devices and to the software of components. This is realized with the AddIn model (8.3.11).

If a *Software Package* becomes very large and only parts of it need to be replaced, there is a need to maintain the individual files of the *Software Package* independently on the *Server*. When all desired files are on the *Server*, the installation can be started for the set of files. Here the *FileSystem* option (8.3.4.5) can be used.

8.2.1.9 Reliable update of Devices that are out of reach

Especially for devices that are not easy to access for a manual reset or replacement, the update shall always result in a working OPC UA *Client* – *Server* connection. This requires an additional confirmation by the update *Client,* so that the *Server* can do an automatic rollback if the communication cannot be established again after a reboot. A Server can support this with the confirmation option (8.3.4.9).

8.2.1.10 Backup and restore parameters that are lost during the update

Very constraint devices may lose parameters during the update. The update *Client* needs to be aware of that and should be able to backup the parameters in advance. After the update - but before the device starts operating again - the parameters need to be restored. This can be supported using the Parameters object (8.3.4.8).

8.2.1.11 Selecting the correct version to install

An update client needs to select the correct version of the *Software Package* to install. The rules behind this decision can be complex and can include e.g. dependency checks or a release process of the distributor and / or operator of the machine. The *Server* can expose information about the device (8.3.11) and information about the *Current Version* (8.4.3.2) which is then used by the *Client* to select an update.

Selecting the new version needs to be done by the user with the help of the update client before transfer and installation. Therefore it is not in scope of this specification.

8.2.1.12 Installation of additional software

Some devices can run several software applications. The *Information Model* should allow the *Client* to transfer and install additional software applications, if the *Server* supports this. This can be done using the *FileSystem based Loading* (8.3.4.5).

8.2.2 Unsupported Use Cases

8.2.2.1 Finding devices that provide the SoftwareUpdate AddIn within a Server

If an OPC UA *Server* abstracts several devices that support the *SoftwareUpdate AddIn*, the *Information Model* shall provide a defined entry point to find all these devices in an efficient manner.

This Use Case is expected to be addressed in other working groups or in a future version of this specification.

A possible solution would be to create a SWUpdate *FolderType* below *DeviceFeatures* as it is described in the DI specification. This folder could reference all *SoftwareUpdate AddIns*.

8.2.2.2 Explicit Restarting the device

In most update scenarios the device can restart automatically during or after the installation. However, there can be situations where it is required to explicitly restart the device by the *Client*.

This use case is not supported by the current version of this specification. Since this feature might be useful outside of software update it should be realized somewhere outside this specification.

8.2.2.3 Pulling software from an external source

Sometimes it is desirable to store all files needed for software update at a central place and have the devices get the files on their own time and pace. In this case the *Client* would tell the *Server* only the location of the file. Then the actual transfer is initiated by the device.

There is no specific support for this use case in this specification. However, it is possible to use the described mechanisms to transfer a file that does not contain the actual software but the location of the external source(s) where the software file(s) should be pulled from.

8.3 General

8.3.1 System perspective

Besides specific *Clients* for specific devices, this specification also describes *Software Update Clients* that can update devices of various vendors (for additional details see 8.3.5).

For devices in operational use it is often necessary to consider the operation state of the software / machine / plant before performing the update (e.g. stop and start the operation). For this case a specialized *Client* can use additional domain-specific *Information Models* as part of the update process.

An update can be performed manually by a user for a single device. However, if a lot of devices need to be maintained on a regular basis an automatic update is desirable. For this scenario the *Information Model* also allows the transfer of software to the devices without starting the update process. For the installation a *Client* could control several devices simultaneously.

8.3.2 Types of software

This common model can describe several types of software that may need to be updated or installed. This can be the firmware or operating system of a device but also be one or more software applications that need to be updated. Configuration and parameters can be maintained as software as well. Besides the update, it is also desired to install additional software. The *Server* can expose all software as a single component or separate it into several smaller components as it is illustrated in Figure 33.

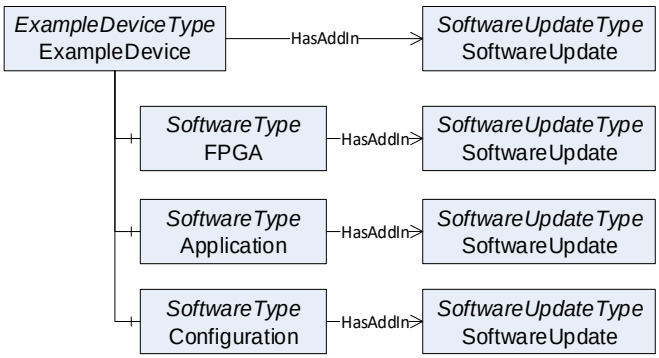

Figure 33 – Example with a device and several software components

8.3.3 Types of Devices

Devices may have different requirements regarding a firmware update, depending on their type and available resource (e.g. memory).

Memory constraint devices like sensors often cannot store an additional firmware. These devices install the new firmware while it is transferred to the device. In this specification this is called *Direct-Loading* (see 8.3.4.3).

Devices with more memory can store a new firmware in a separate memory without installing it which is referred as *Cached-Loading* in this specification (see 8.3.4.4). In this case the installation is separated from the file transfer and can be done later or with a different *Client*.

Some devices have two memory partitions for the operating system. One active partition that is used in the boot process and a second alternative fallback partition. These devices install the firmware into the fallback partition and then perform a restart after swapping the active partition. This has an advantage if the device detects an issue with the new firmware: The change can easily be reverted to the old version by switching the partitions again (with another reboot).

Constraint devices like sensors typically do not support a real file system. Devices with more memory often have a file system which can be used to store files like firmware, parameters and backups. This *Information Model* provides update mechanisms for both types of devices (see 8.3.4.5 for *FileSystem based Loading*).

8.3.4 Options for the Server

8.3.4.1 Overview

Updating software or firmware of a machine or plant is a complex task and different devices have different requirements to the update or installation of software. To support this, the *SoftwareUpdateType* provides several options where a vendor can select the parts that are necessary for the software update.

All these options are exposed as optional *References* of the *SoftwareUpdateType*. A *Server* can choose which options it wants to support (the Profiles section describes valid combinations of options).

This way the *Server* can choose between *Direct-Loading*, *Cached-Loading* or *FileSystem based Loading* and it may use additional optional features like manual power cycle, parameter backup / restore or confirmation.

A *Software Update Client* needs to check which options are exposed by the *Server* and how the *Server* behaves during the update (a *Software Update Client* is described in 8.3.5).

8.3.4.2 Prepare for update option

There are situations where it is preferable to prepare the device explicitly before the installation and resume operation explicitly after the installation. The *PrepareForUpdateStateMachine*, which is described in 8.4.8 can be used for this task.

This can be the case, when several devices of a machine should be updated at once. All devices have to be prepared first to ensure that all are waiting for an update. After that they can be updated by the *Client*. At the end after all individual updates are complete the devices can resume operation.

Or a device requires the behavior to enter a safe state (e.g. reaching a safe area) to be able to update the software.

If the installation comprises several steps (e.g. backup parameters, install firmware, restore parameters). The steps can be encapsulated by the *Prepare* and *Resume Methods* to ensure consistency between all the steps.

8.3.4.3 Direct-Loading option

The *Direct-Loading* option provides a model where the installation is part of the transfer. To support the *Direct-Loading* model the *Server* has to provide the *Current Version*. This includes parameters like the version number, a release date or patch identifiers. With this information the *Client* can decide if an update is required and which version to install.

The *Software Package* is transferred using the *TemporaryFileTransferType* (OPC 10000-5). This includes the installation itself so that the installation option is not used.

For *Direct-Loading* the *DirectLoadingType* is used, which is described in 8.4.4.

8.3.4.4 Cached-Loading option

The *Cached-Loading* option provides a model where the transfer of the *Software Package* and its installation are separate steps. To support the *Cached-Loading* model the *Server* has to provide the *Current Version* and the *Pending Version*. Optionally the *Fallback Version* can be supported.

With the *Current Version* the *Client* can decice if an update is required and which version to transfer. With the *Pending Version* the *Client* can ensure to install the desired version. With the *Fallback Version* the *Client* can install an alternative version.

Software Packages are transferred using the *TemporaryFileTransferType* (OPC 10000-5). The new software may be transferred in the background without stopping the device. The actual installation of the software can be done later using the installation option.

For *Cached-Loading* the *CachedLoadingType* is used, which is described in 8.4.5.

8.3.4.5 FileSystem option

The Cached-Loading option with a self-contained *Software Package* and concrete definition of the version information can be too restrictive for some devices. E.g. if new software should be installed. For this use case the *FileSystem based Loading* provides an open structure of files and directories where a *Client* can read and write. These files could be e.g. configuration, setup files or recipes.

Note: The *FileSystem* exposed in the address space may not be congruent with the actual file system of the device.

The purpose of the directories and files is not part of this specification. It needs to be known by the *Client* and the *Server*. Other companion specifications could add this definition for specific types of devices. If accessed by a *Software Update Client,* the *FileSystem* root can be used to store and install the files.

For *FileSystem based Loading* the *FileSystemLoadingType* is used, which is described in 8.4.6.

8.3.4.6 Installation option

Using the *Cached-Loading* option or the *FileSystem* option, a transferred *Software Package* or file needs to be installed explicitly (compared to the implicit installation of *Direct-Loading*). Therefore, the *InstallationStateMachineType* shall be used (see 8.4.9). It can either be used to install a *Software Package* (*Cached-Loading*) or a list of files from the *FileSystem (File System based Loading)*.

8.3.4.7 UpdateStatus option

The update *Clients* are often operated by human users. Since an update normally is a long process, the user would like to see the current state. At a first glance the percentage can give a hint about completion of the update, especially if several devices are updated at the same time. But if there are unexpected delays or errors the user needs a detailed textual description about the current update action or issue.

This can be accomplished with the *UpdateStatus Variable* (see 8.4.1.8). A *Client* can subscribe to it for a user display. At least if a state machine is in an error state the *UpdateStatus* should provide a meaningful error message for the user.

8.3.4.8 Parameter backup / restore option

If the device cannot keep the parameters during the update, it shall support the *Parameters Object* of the *SoftwareVersionType* (see 8.4.1.7). If supported by the *Server*, the update *Client* should perform a backup of the parameters before and restore the parameters after the software update.

8.3.4.9 Confirmation option

The confirmation option supports the use case of 8.2.1.9: A *Client* may set a *ConfirmationTimeout* before the installation. After every reboot of the *Server* caused by the update, it shall wait this time for a call to the *Confirm Method.* If the call is not received the *Server* shall perform a rollback to enable a working *Client – Server* connection again. This state machine is defined in 8.4.11.

8.3.4.10 Power cycle option

The power cycle option is intended for devices where a manual power cycle is required. During the installation the state *WaitingForPowerCycle* informs the user that it is time to turn the power off and on again. The *PowerCycleStateMachineType* is defined in 8.4.10.

If an instance of the *SoftwareUpdateType* supports the power cycle option, the *UpdateBehavior RequiresPowerCycle* shall indicate if this might happen for an installation.

This power cycle state machine is used in combination with the installation. For *Cached-Loading* it may be used in the *Installing* state of the *InstallationStateMachineType*. For *Direct-Loading* it may be used during the transfer of the new software with the *TemporaryFileTransferType* (OPC 10000-5) of the *DirectLoadingType*.

8.3.5 Software Update Client

The first task of a *Software Update Client* is to find the components that support software update. After that it can execute the update of the components one by one or in parallel. The following activity diagrams illustrate how a *Software Update Client* can perform an update using the different update types. The first task is to detect what options are supported by browsing the references of the *SoftwareUpdate AddIn*. Then the Client can check the version information to determine whether an update is necessary. This is illustrated in Figure 34.

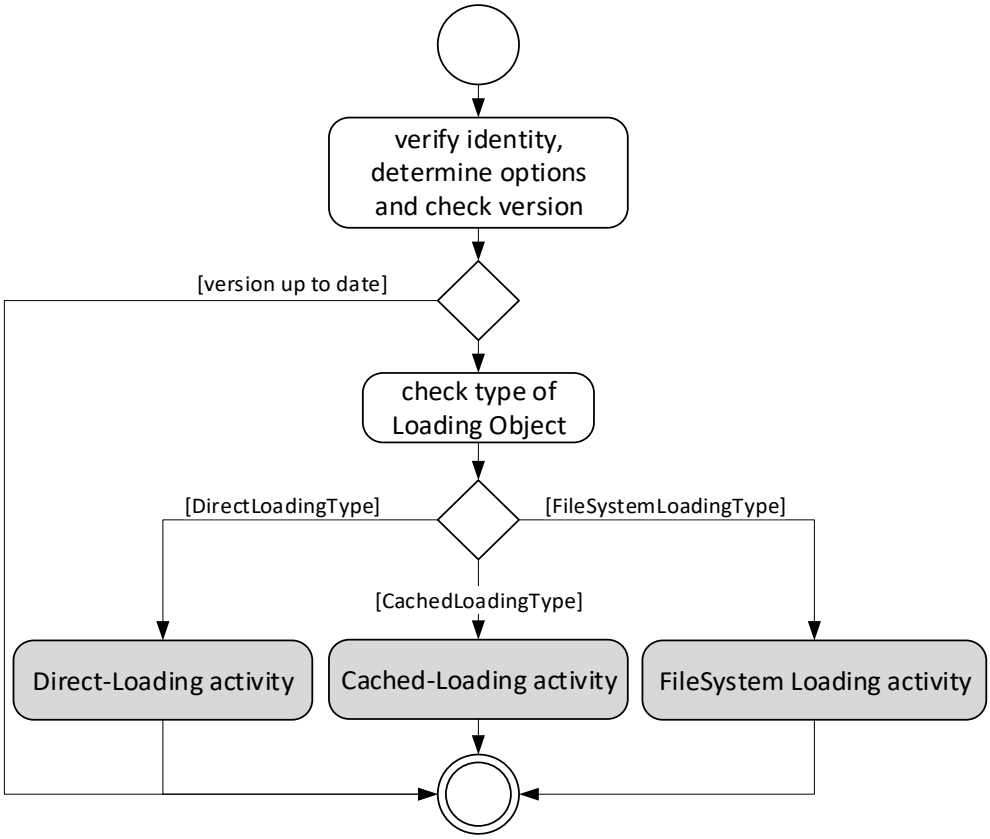

Figure 34 – Determine the type of update that the Server implements.

The activities of the different loading types are slightly different. With *Cached-Loading* the *Client* can check *CurrentVersion* and *PendingVersion Objects* to determine if the *Software Package* is already transferred. With the *FileSystem based Loading* the *Client* can browse the *FileSystem* to find out which files are already transferred. For *Cached-Loading* and *File System based Loading* the transfer can be done in advance. There are different ways to get the *UpdateBehavior*, because for *Cached-Loading* and *File System based Loading* this depends on the actual software that should be installed (with *Direct-Loading* the server has no information about the new software). For *Direct-Loading* and

Cached-Loading the validation is done during the transfer. For File System based Loading this needs to be done before the installation as an extra step. These steps are illustrated in Figure 35.

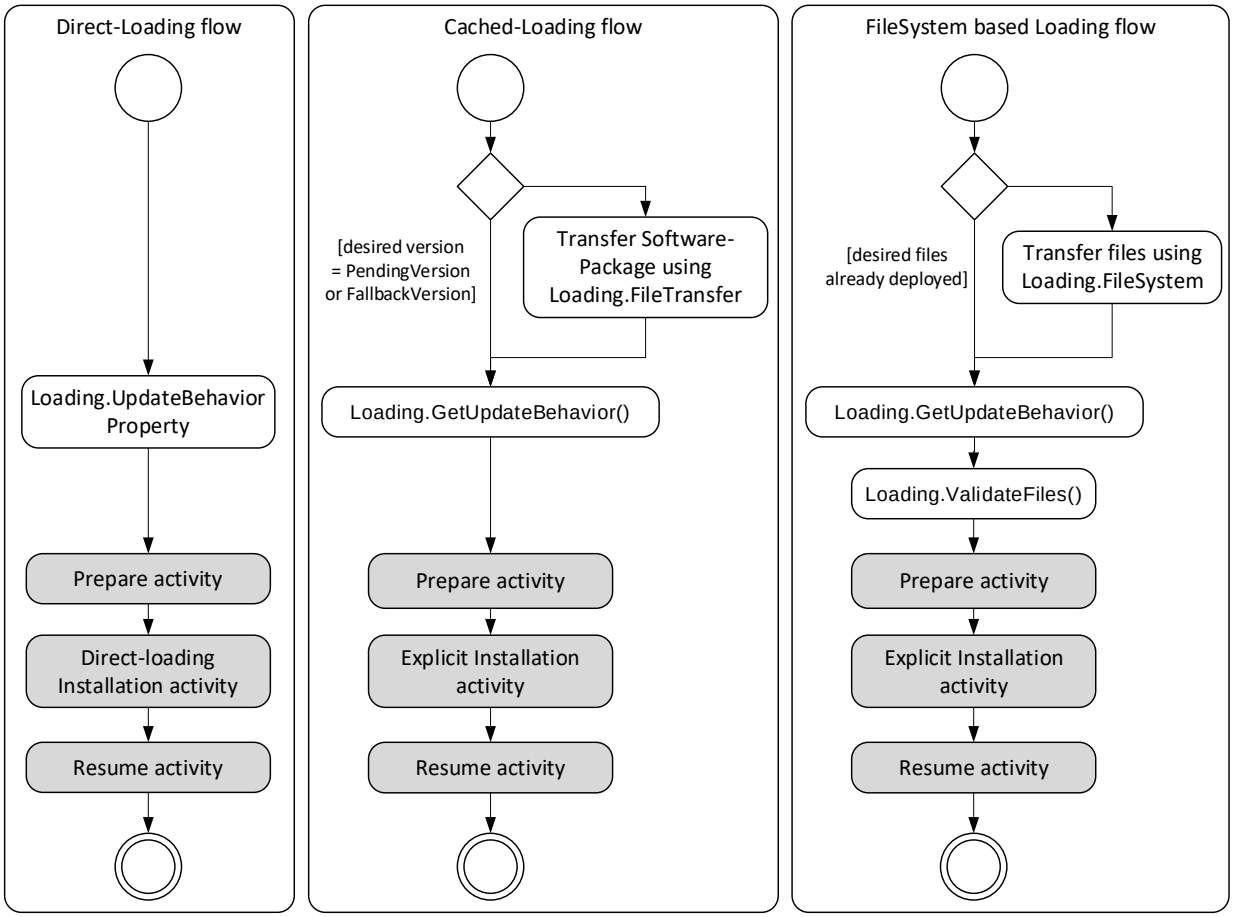

Figure 35 – Different flows of *Direct-Loading*, *Cached-Loading* and *FileSystem based Loading*

The prepare activity can be handled equal for all types of loading. This optionally includes a backup if the device cannot keep the parameters during update. The activity is shown in Figure 36.

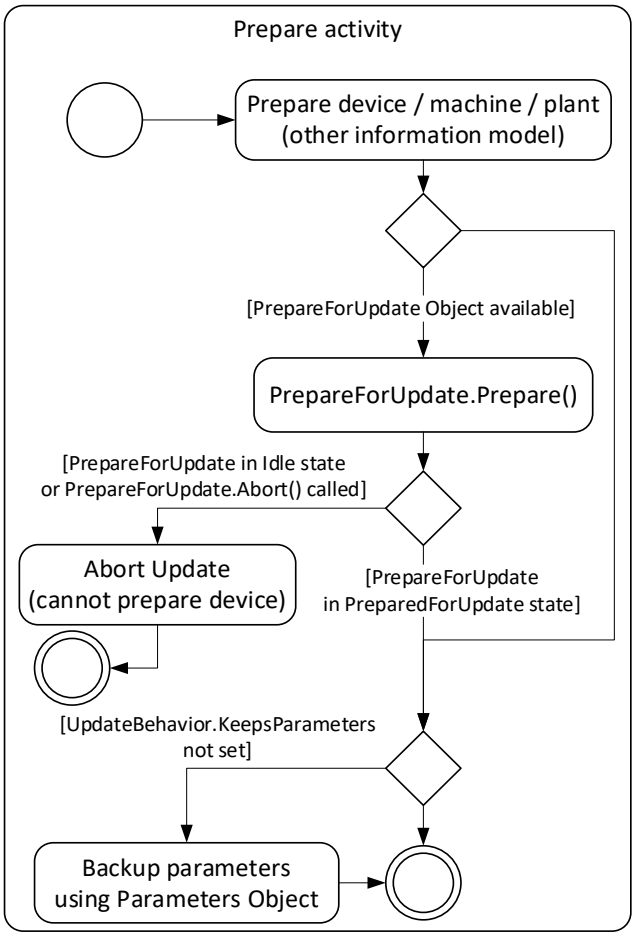

Figure 36 – Prepare and Resume activities

The actual installation of *Direct-Loading* is done during the transfer. At the end there can be a manual power cycle (option). In some cases (if the *Server* is on the device that is updated) the *Server* is rebooted and the *Client* needs to reconnect to complete the installation. This is illustrated in Figure 37.

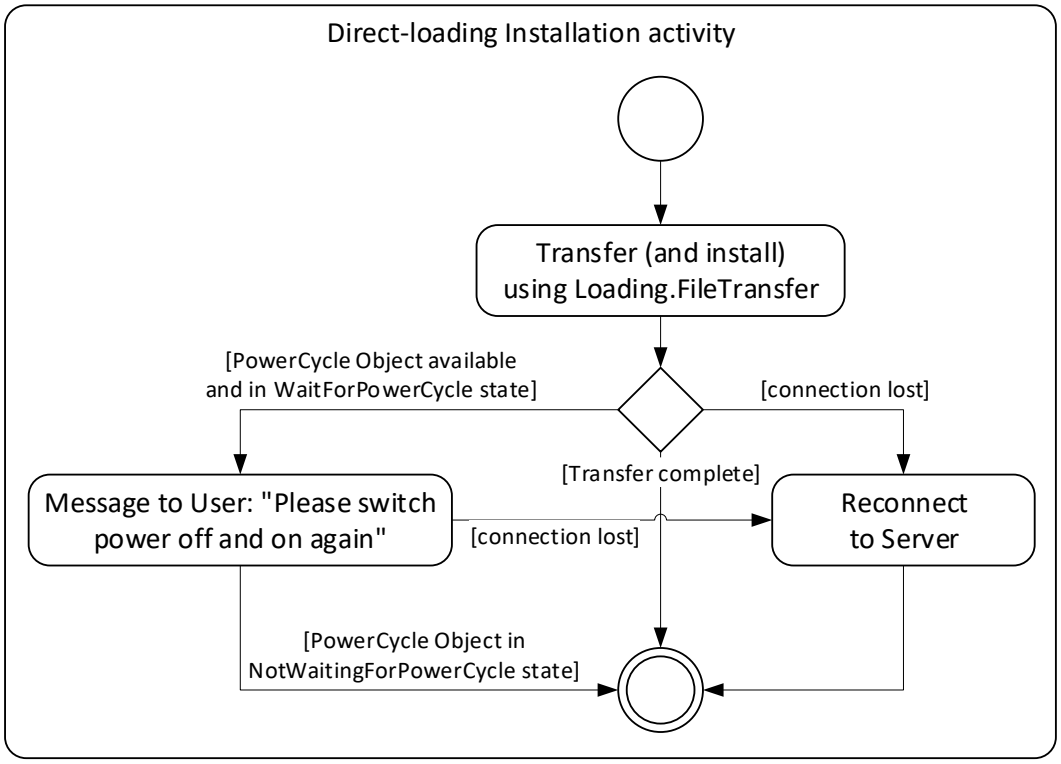

Figure 37 – Installation activity for *Direct-Loading*

For *Cached-Loading* and *File System based Loading* the installation is done using the *InstallationStateMachineType*. For *Cached-Loading* the *InstallSoftwarePackage Method* is used and for *File System based Loading* the *InstallFiles Method* is used. During this installation there may also be a manual power cycle request requiring operator input. The *Client* might also need to reconnect one or more times due to automatic reboots. If the *Confirmation Object* is available, the *Client* may use it during the installation. This is illustrated in Figure 38.

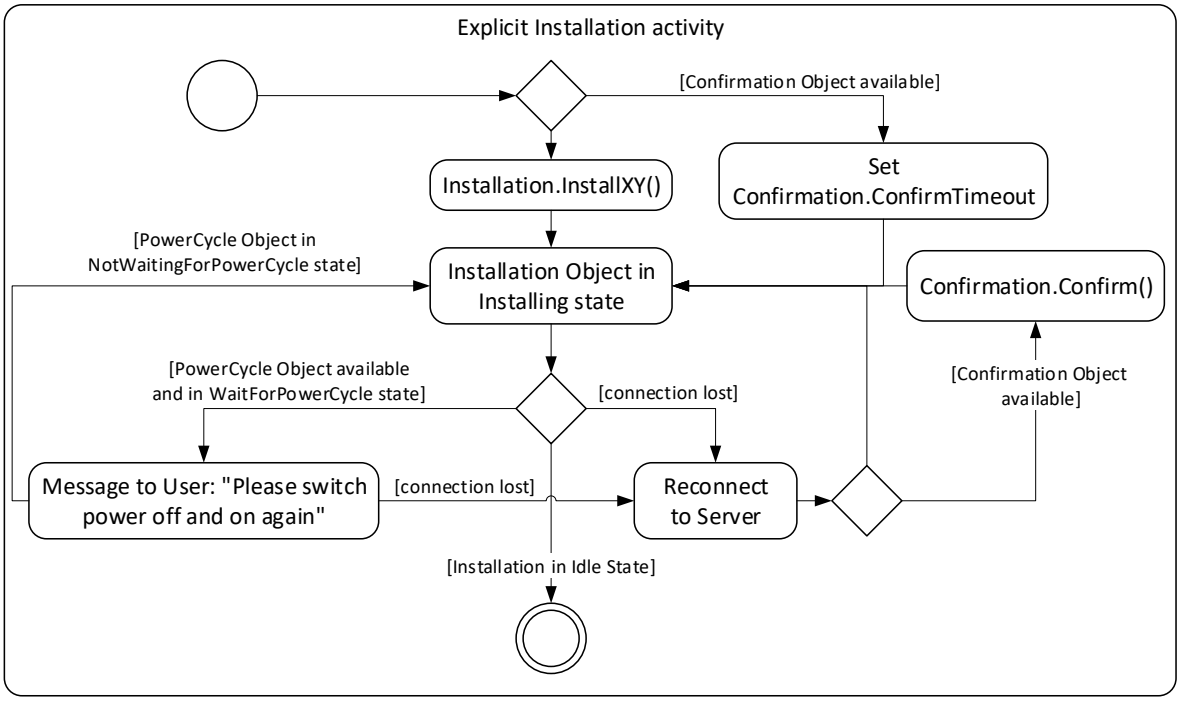

Figure 38 – Installation activity for *Cached-Loading* and *File System based Loading*

The resume activity can be handled equal for all types of loading. This optionally includes restore if the device cannot keep the parameters during update. The activity is shown in Figure 39.

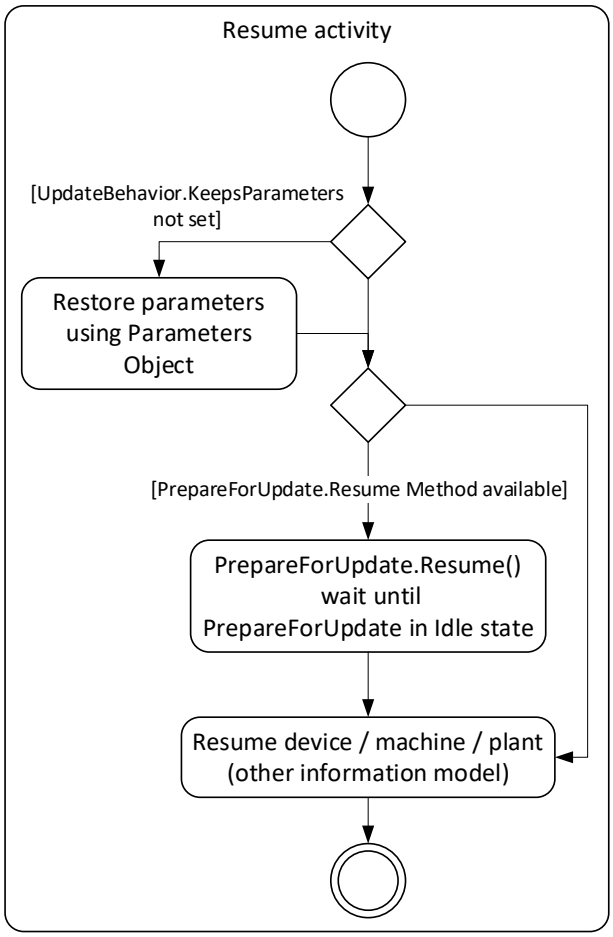

Figure 39 – Resume activity

8.3.6 Safety considerations

Especially for safety critical devices the update *Client* needs to inform the user before performing critical activities. This includes the information if a manual power cycle is required, if the device will reboot or if it will lose its parameters during the update. This information can be accessed before the actual update is started. For safety all security considerations also apply.

8.3.7 Security considerations

Security is a critical aspect of software update. The basic requirements can be solved with the existing UA security mechanisms (secure transport, authorization and role based authentication). Only authorized users shall be able to install and manage updates.

The *Client* needs to verify the identity of the device. This can be complished by identification information provided by OPC UA, by this specification or by companion specifications.

The authenticity (integrity and source) of the *Software Package* need to be verified. These aspects can be implemented by the device in a vendor specific way e.g. verify a digital signature of the *Software Package*. These mechanisms are out of scope of this specification.

8.3.8 Update Behavior

The concrete process of the installation can depend on the device and on the software that is to be installed. Therfore the server provides the *UpdateBehavior OptionSet* (see 8.5.2). The *UpdateBehavior* can be determined with the *UpdateBehavior Variable* (see 8.4.4.3) of the *DirectLoadingType* or with one of the *GetUpdateBehavior Methods* of the *CachedLoadingType* (see 8.4.5.5) or the *FileSystemLoadingType* (see 8.4.6.3).

8.3.9 Installation of patches

Instead of updating the whole software with a new version, sometimes only a part of it need to be replaced (“patched”). The installation of such a patch can be implemented in the same way as the installation of a complete version. The only difference is that the result is not a new *SoftwareRevision* but an additional entry in the list of patch-identifiers stored in the *PatchIdentifiers Variable* (see 8.4.7.5).

8.3.10 Incompatible parameters / settings

If parameters or settings of an old software do not work with the new software, the installation of the new software can complete but the device still cannot start as before. In this case the *Server* should treat the installation as successful. It can inform the incompatibility using e.g., the *IDeviceHealthType Interface* (see 4.5.4) of the device / component. This issue can be resolved later by a client that fixes or updates the parameters.

8.3.11 AddIn model

To support an individual software update for the devices of a *Server AddressSpace* the software update model is defined using the *AddIn* model as it is described in OPC 10001-7. An instance of *SoftwareUpdateType* shall be attached to either *Objects* that implement the *Interface IVendorNameplateType* (see 4.5.2) or *Objects* that support an *Identification FunctionalGroup* (see B.2) that implements *IVendorNameplateType*. For the *AddIn* instance the fixed *BrowseName* “SoftwareUpdate” shall be used. This model gives any device, hardware- or software-component the opportunity to support *SoftwareUpdate*.

With this mechanism it is also possible to update parts of a software independently: A *Server* could expose parts as additional software components with their own update *AddIn*.

To identify the device / component that is the target for the software update, the *IVendorNameplateType Interface* is used. In this *Interface* at least the *Variables Manufacturer, ManufacturerUri, ProductCode* and *SoftwareRevision* shall be supported and have valid values. Optionally *Model* and *HardwareRevision* should be supported. These *Properties* may be shown to the operator. *ManufacturerUri, ProductCode* and *HardwareRevision* should be used to identify the component.

Note that the *Properties SoftwareRevision, Manufacturer* and *ManufacturerUri* also appears in the *CurrentVersion* of the *PackageLoadingType*. Their values may be different, if the manufacturer of the *Device* is not the same as the manufacturer of the software. The *SoftwareRevision Object* shall be the same at both places.

The *ComponentType* (see 4.6) already implements the *Interface IVendorNameplateType*. This makes it a good candidate for a *SoftwareUpdate AddIn* as illustrated in the example in Figure 40.

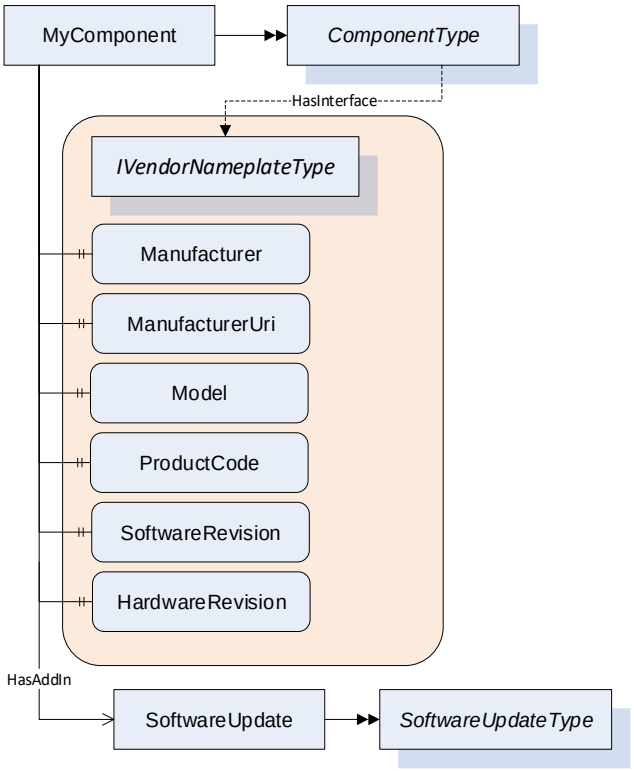

Figure 40 – Example how to add the SoftwareUpdate AddIn to a component

8.4 ObjectTypes

8.4.1 SoftwareUpdateType

8.4.1.1 Overview

The *SoftwareUpdateType* defines an *AddIn* which may be used to extend *Objects* with software update features. All software update options are exposed as references of this *AddIn*. This way a *Client* can check for the references of the *AddIn* to determine which options are provided by a *Server*. If an option is available, it shall be used as specified.

The *SoftwareUpdateType* is illustrated in Figure 41 and formally described in Table 65.

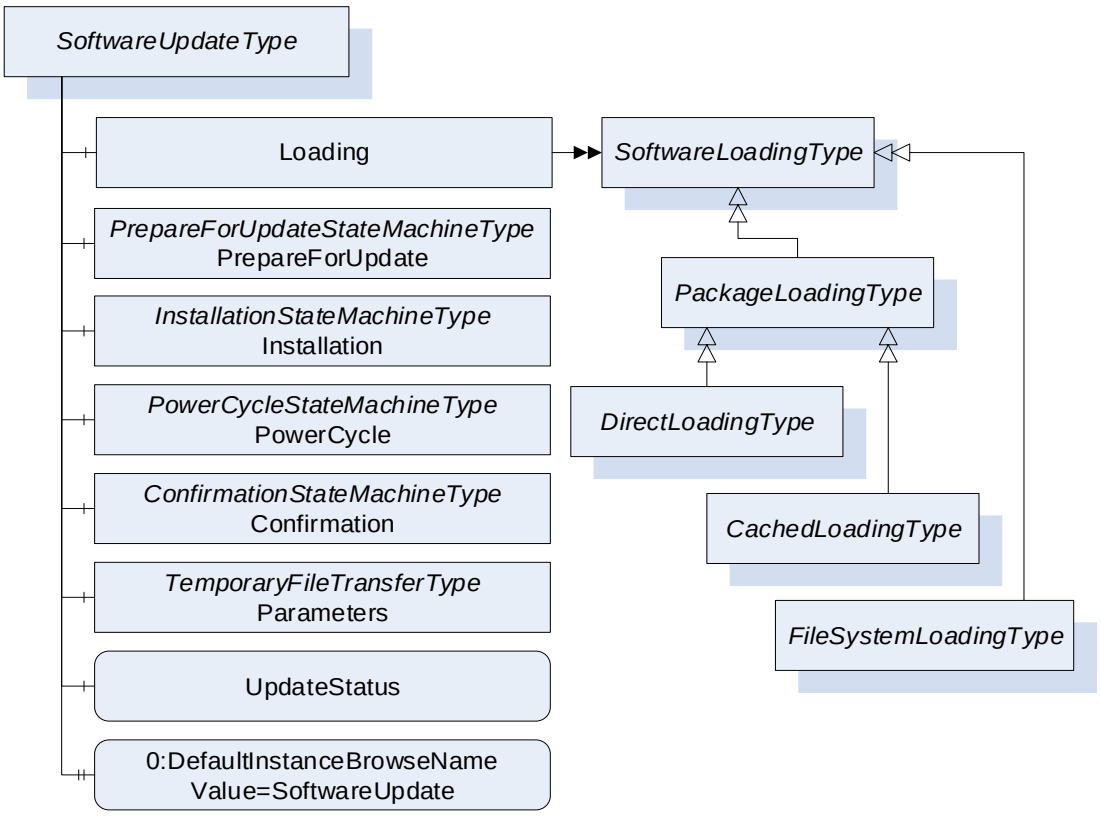

Figure 41 – SoftwareUpdateType

Table 65 – SoftwareUpdateType definition

Attribute	**Value**				
BrowseName	SoftwareUpdateType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *BaseObjectType* defined in OPC 10000-5.					
HasComponent	Object	Loading		SoftwareLoadingType	Optional
HasComponent	Object	PrepareForUpdate		PrepareForUpdateStateMachine Type	Optional
HasComponent	Object	Installation		InstallationStateMachineType	Optional
HasComponent	Object	PowerCycle		PowerCycleStateMachineType	Optional
HasComponent	Object	Confirmation		ConfirmationStateMachineType	Optional
HasComponent	Object	Parameters		TemporaryFileTransferType	Optional
HasComponent	Variable	UpdateStatus	LocalizedText	BaseDataVariableType	Optional
HasComponent	Variable	VendorErrorCode	Int32	BaseDataVariableType	Optional
HasProperty	Variable	0:DefaultInstanceBrowseName	QualifiedName	PropertyType	
Conformance Units					
DI SU Software Update					

8.4.1.2 Loading

The optional *Loading Object* is of type *SoftwareLoadingType,* which is abstract. The *Object* can be one of the concrete sub-types *DirectLoadingType* (8.4.4), *CachedLoadingType* (8.4.5) or *FileSystemLoadingType* (8.4.6). *SoftwareLoadingType* is formally defined in 8.4.2.

The *Loading Object* is required for all variations of software installation, it is not required for read or restore of device parameters using the *Parameters Object*.

8.4.1.3 PrepareForUpdate

The optional *PrepareForUpdate Object* is of type *PrepareForUpdateStateMachineType* which is formally defined in 8.4.8.

8.4.1.4 Installation

This optional *Installation Object* is of type *InstallationStateMachineType which* is formally defined in 8.4.9.

8.4.1.5 PowerCycle

This optional *PowerCycle Object* is of type *PowerCycleStateMachineType* which is formally defined in 8.4.10.

8.4.1.6 Confirmation

This optional *Confirmation Object* is of type *ConfirmationStateMachineType* which is formally defined in 8.4.11.

8.4.1.7 Parameters

This optional *Parameters Object* is of type *TemporaryFileTransferType* (OPC 10000-5). It may be supported by devices that cannot retain parameters during update. If supported by the *SoftwareUpdate AddIn* a Client can read the parameters before the update and restore them after the update. This is not a general-purpose backup and restore function. It is intended to be used in the context of software update.

The *GenerateFileForRead* and *GenerateFileForWrite Methods* accept an unspecified *generateOptions Parameter*. This argument is not used, and *Clients* shall always pass null. Future versions of this specification may define concrete *DataTypes*.

If the restore of parameters succeeds but the software cannot run properly this should not be treated as an error of the restore. Instead this should be indicated using the *IDeviceHealthType Interface of the* device / component.

8.4.1.8 UpdateStatus

This optional localized string provides status and error information for the update. This may be used whenever a long running update activity can provide detailed information to the user or when a state machine wants to provide error information to the user.

A *Server* may provide any text it wants to show to the operator of the software update. Important texts are the error messages in case anything went wrong, and the installation or preparation could not complete. These messages should explain what happened and how the operator could resolve the issue (e.g. “try again with a different version”). During preparation and installation, it is good practice to inform the operators about the current action to keep them patient and waiting for the completion. Also, if the installation gets stuck this text would help to find out the reason.

The *UpdateStatus* may be used together with the *PrepareForUpdateStateMachineType* (8.4.8)*, the InstallationStateMachineType* (8.4.9) and for *CachedLoadingType* (8.4.5), *DirectLoadingType* (8.4.4) and *FileSystemLoadingType* (8.4.6) it may be used during the transfer of the *Software Package*.

8.4.1.9 VendorErrorCode

The optional *VendorErrorCode Property* provides a machine-readable error code in case anything went wrong during the transfer, the installation or the preparation. Comparable to an error message in *UpdateStatus* this *Variable* can provide additional information about the issue. The

VendorErrorCode is an additional information for a *Client*. It is not required for normal operation and error handling.

The value 0 shall be interpreted as no error.

The *VendorErrorCode* may be used together with the *PrepareForUpdateStateMachineType* (8.4.8) for prepare and resume, in the *InstallationStateMachineType* (8.4.9) during the installation. For *CachedLoadingType* (8.4.5), *DirectLoadingType* (8.4.4) and FileSystemLoadingType (8.4.6) it may be used during the transfer of the *Software Package*.

8.4.1.10 DefaultInstanceBrowseName

The *DefaultInstanceBrowseName Property* – defined in OPC 10000-3 – is required for the *AddIn* model as specified in 8.3.11. It is used to specify the *BrowseName* of the instance of the *SoftwareUpdateType*. It always has the value “SoftwareUpdate”.

Table 66 – SoftwareUpdateType Attribute values for child Nodes

Source Path	Value
0:DefaultInstanceBrowseName	SoftwareUpdate

8.4.2 SoftwareLoadingType

8.4.2.1 Overview

The *SoftwareLoadingType* is the abstract base for all different kinds of loading. The concrete information and behavior is modeled in its sub-types.

The *SoftwareLoadingType* is formally defined in Table 71.

Table 67 – SoftwareLoadingType definition

Attribute	Value				
BrowseName	SoftwareLoadingType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in OPC 10000-5					
HasSubtype	ObjectType	PackageLoadingType			
HasSubtype	ObjectType	FileSystemLoadingType			
HasComponent	Variable	UpdateKey	String	BaseDataVariableType	Optional
Conformance Units					
DI SU Software Update					

8.4.2.2 UpdateKey

The optional write-only *UpdateKey Object* can be used if the underlying system requires some key to unlock the update feature. The format and where to get the key is vendor-specific and not described in this specification. If *UpdateKey* is supported, the *Client* shall set the key before the installation. If the *PrepareForUpdateStateMachine* is used, the *UpdateKey* shall be set before the *Prepare Method* is called. The *Server* shall not keep the value for more than one update.

8.4.3 PackageLoadingType

8.4.3.1 Overview

The *PackageLoadingType* provides information about the *Current Version* and allows transfer of a *Software Package* to and from the *Server*.

The *PackageLoadingType* is illustrated in Figure 42 and formally defined in Table 68.

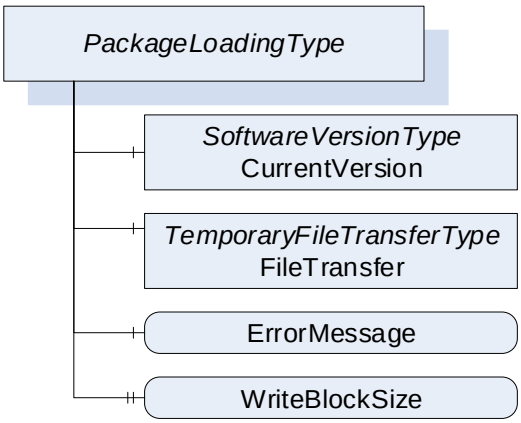

Figure 42 – PackageLoadingType

Table 68 – PackageLoadingType definition

Attribute	Value				
BrowseName	PackageLoadingType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *SoftwareLoadingType*					
HasComponent	Object	CurrentVersion		SoftwareVersionType	Mandatory
HasComponent	Object	FileTransfer		TemporaryFileTransferType	Mandatory
HasComponent	Variable	ErrorMessage	LocalizedText	BaseDataVariableType	Mandatory
HasProperty	Variable	WriteBlockSize	UInt32	PropertyType	Optional
HasSubtype	ObjectType	DirectLoadingType			
HasSubtype	ObjectType	CachedLoadingType			
Conformance Units					
DI SU Software Update					

8.4.3.2 CurrentVersion

To identify the *Current Version*, the *CurrentVersion Object* provides *ManufacturerUri*, *SoftwareRevision* and *PatchIdentifiers* along with other information that allows the user to identify the currently used software. With this information the *Client* can determine a suitable update.

Note: This version information is about the installed software. The *Manufacturer* is not necessarily the same as the *Manufacturer* of the physical device that executes the software.

8.4.3.3 FileTransfer

The *FileTransfer Object* is of type *TemporaryFileTransferType* as defined in OPC 10000-5. It is used to create temporary files for download and upload of the software.

In the *TemporaryFileTransferType* type the *GenerateFileForRead* and *GenerateFileForWrite Methods* take an unspecified *generateOptions Parameter*. For the *FileTransfer Object* an *Enumeration* of type *SoftwareVersionFileType* is used for this *Parameter*. It is used to select the file to upload or download. All allowed values are defined in Table 86. Additional *Result Codes* of the *GenerateFileForRead* and *GenerateFileForWrite Methods* are specified in Table 69.

Table 69 – TemporaryFileTransferType Result Codes

Result Code	Description
Bad_InvalidState	If the *PrepareForUpdate* is available, the *UpdateBehavior* requires preparation and the *PrepareForUpdate* state machine is not in the state *PreparedForUpdate*.
Bad_NotFound	If there is no file to read from the device.
Bad_NotSupported	If the device does not support to upload / download of the *Software Package*.

For all errors that occur during the file transfer the *ErrorMessage Variable* should provide an error message for the user.

It is implementation dependent which version (see *SoftwareVersionFileType* in 8.5.1) is readable and which one is writable. Additional restrictions are defined in the concrete sub-types of *PackageLoadingType*.

8.4.3.3.1 Transfer to the device

The software is transferred as a single package. File type and content are device specific. If *WriteBlockSize* is supported, the *Client* shall write the file in chunks of this size.

The software should be validated during the transfer process. Errors shall be indicated either in the *Write Method,* the *CloseAndCommit Method* or an asynchronous completion of the file transfer. If the validation is performed synchronous, the *Method* returns Bad_InvalidArgument; if the validation is performed asynchronous, the error is indicated by the *Error* state of the *FileTransferStateMachineType*. If the *ErrorMessage Variable* is provided, it shall contain an error message representing the validation error.

8.4.3.3.2 Transfer from the device

The *FileTransfer Object* may optionally support the transfer of a *Software Package* from the device to the *Client*.

If this transfer is not supported, the *Server* shall return the *Result Code Bad_NotSupported*. If it is supported but there is currently no data, the *Result Code Bad_NotFound* shall be used instead.

8.4.3.4 ErrorMessage

This is a textual information about errors that can occur with the file transfer. Whenever a method of the TemporaryFileTransferType returns an error, the *ErrorMessage Variable* should provide a localized error message for the user. For every new file transfer the value should be reset to an empty string.

8.4.3.5 WriteBlockSize

Optional size of the blocks (number of bytes) that a *Client* shall write to the file. The client shall write the *Software Package* in chunks of this size to the *FileType* object (the last block may be smaller).

8.4.4 DirectLoadingType

8.4.4.1 Overview

The *DirectLoadingType* provides information about the *Current Version* and allows transfer of a *Software Package* to and from the *Server*. Transfer of the *Software Package* to the *Server* also includes the installation. The *Direct-Loading* option is described in 8.3.4.3.

The *DirectLoadingType* is illustrated in Figure 43 and formally defined in Table 70.

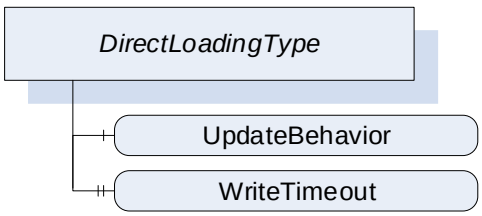

Figure 43 – DirectLoadingType

Table 70 – DirectLoadingType definition

Attribute	Value				
BrowseName	DirectLoadingType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *PackageLoadingType*					
HasComponent	Variable	UpdateBehavior	UpdateBehavior	BaseDataVariableType	Mandatory
HasProperty	Variable	WriteTimeout	Duration	PropertyType	Optional
Conformance Units					
DI SU DirectLoading					

8.4.4.2 FileTransfer

The *FileTransfer Object* is inherited from the *PackageLoadingType*. In this sub-type the *Current* version shall be writable (see *SoftwareVersionFileType* in 8.5.1). Writing to this file also includes the actual installation.

8.4.4.3 UpdateBehavior

The *UpdateBehavior OptionSet* informs the update *Client* about the specific behavior of the component during update via *Direct-Loading*.

8.4.4.4 WriteTimeout

Optional Property that informs the *Client* about the maximum duration of the call to the *Write Method* of *FileType* (maximum time the write of a block of data can take). If the write operation takes longer the *Client* can assume that the *Server* has an issue.

8.4.5 CachedLoadingType

8.4.5.1 Overview

The *CachedLoadingType* provides information about the *Current Version*, the *Pending Version* and the *Fallback Version* (if supported). Additionally, it allows upload and download of different versions of the software. The *Cached-Loading* option is described in 8.3.4.4.

The *CachedLoadingType* is illustrated in Figure 44 and formally defined in Table 71.

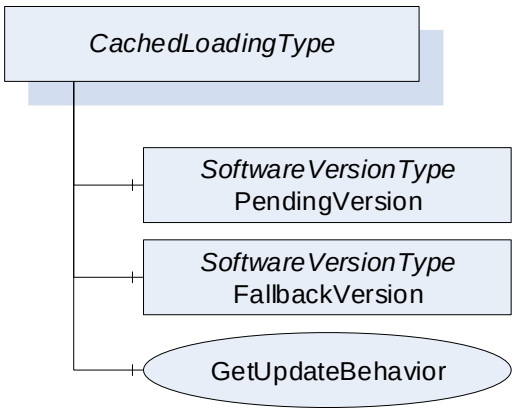

Figure 44 – CachedLoadingType

Table 71 – CachedLoadingType definition

Attribute	Value				
BrowseName	CachedLoadingType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *PackageLoadingType*					
HasComponent	Object	PendingVersion		SoftwareVersionType	Mandatory
HasComponent	Object	FallbackVersion		SoftwareVersionType	Optional
HasComponent	Method	GetUpdateBehavior			Mandatory
Conformance Units					
DI SU CachedLoading					

8.4.5.2 FileTransfer

The *FileTransfer Object* is inherited from the *PackageLoadingType*. In this sub-type the *Current* version shall not be writable and the *Pending* version shall be writable (see *SoftwareVersionFileType* in 8.5.1).

8.4.5.3 PendingVersion

The *PendingVersion Object* describes an already transferred new *Software Package* that is ready to be installed.

If there is no *Software Package* available, the values should be empty.

8.4.5.4 FallbackVersion

The optional *FallbackVersion Object* describes an alternate version on the device. This could be a factory default version or the version before the last update. Installing the *Fallback Version* may be used to revert to a reliable version of the software.

If a *Fallback Version* is supported by the device the object shall be available. If there is currently no *Fallback Version* on the device, the values should be empty.

8.4.5.5 GetUpdateBehavior Method

With this *Method* the *Client* may check the specific update behavior for a specified software version. To identify the version the *GetUpdateBehavior Method* requires the *ManufacturerUri*, *SoftwareRevision and PatchIdentifiers Properties* of the *SoftwareVersionType*.

Signature

```
GetUpdateBehavior(
  [in]   String           ManufacturerUri,
  [in]   String           SoftwareRevision,
  [in]   String[]         PatchIdentifiers,
  [out]  UpdateBehavior   UpdateBehavior);
```

Argument	Description
ManufacturerUri	*ManufacturerUri Property* of either the *Pending* or *Fallback SoftwareVersionType* that should be installed.
SoftwareRevision	*SoftwareRevision Property* of either the *Pending* or *Fallback SoftwareVersionType* that should be installed.
PatchIdentifiers	*PatchIdentifiers Property* of either the *Pending* or *Fallback SoftwareVersionType* that should be installed. (or empty array if not supported by the *SoftwareVersionType* instance)
UpdateBehavior	Update behavior option set for the specified *SoftwareVersionType* instance

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	If the *Software Package*, identified by the parameters, does not exist.

8.4.6 FileSystemLoadingType

8.4.6.1 Overview

The *FileSystemLoadingType* enables software update based on an open file system. This enables the *FileSystem based Loading* option of 8.3.4.5.

It is illustrated in Figure 45 and formally defined in Table 72.

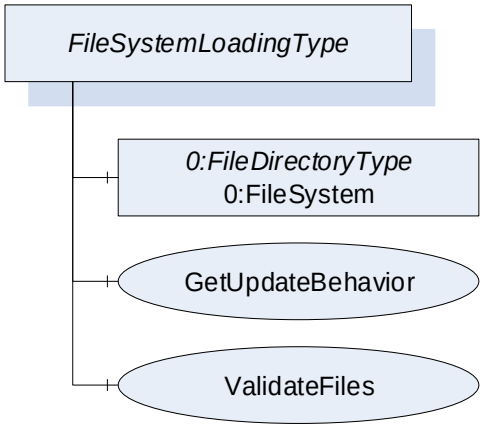

Figure 45 – FileSystemLoadingType

Table 72 – FileSystemLoadingType definition

Attribute	Value				
BrowseName	FileSystemLoadingType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *SoftwareLoadingType*					
HasComponent	Object	0:FileSystem		0:FileDirectoryType	Mandatory
HasComponent	Method	GetUpdateBehavior			Mandatory
HasComponent	Method	ValidateFiles			Optional
Conformance Units					
DI SU FileSystem Loading					

8.4.6.2 FileSystem

The *FileSystem Object* is of type *FileDirectoryType* as it is defined in OPC 10000-5. It provides access to a hierarchy of directories and files of the device. The structure may be read and written by the *Client* however the device may restrict this for specific folders or files.

8.4.6.3 GetUpdateBehavior Method

This *Method* may be used to check the specific update behavior for a set of files. The files are identified by the *NodeId* of their *FileType* instance in the *FileSystem*.

Signature

```
GetUpdateBehavior(
```

```
[in] NodeId[]          NodeIds,
[out]UpdateBehavior    UpdateBehavior);
```

Argument	Description
NodeIds	NodeIds of the files to install.
UpdateBehavior	Update behavior *OptionSet* for the files specified by *NodeId*

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	If one or more *NodeIds* are not found.

8.4.6.4 ValidateFiles Method

This *Method* may be used to check if the specified set of files are valid and complete for an installation. This should also include dependency checks if appropriate.

Note: In case of *Direct-Loading* or *Cached-Loading* these checks should be part of the transfer and this method shall not be supported since it is part of the file transfer (e.g. in *CloseAndCommit*).

Signature

```
ValidateFiles(
  [in] NodeId[]     NodeIds,
  [out]ErrorCode    Int32,
  [out]ErrorMessage LocalizedText);
```

Argument	Description
NodeIds	NodeIds of the files to validate.
ErrorCode	0 for success or device specific number for validation issues.
ErrorMessage	Message for the user that describes how to resolve the issue.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_NotFound	If one or more *NodeIds* are not found.

8.4.7 SoftwareVersionType

8.4.7.1 Overview

The *SoftwareVersionType* identifies a concrete version of a software. It is used by the *CachedLoadingType* (8.4.5) and the *DirectLoadingType* (8.4.4) to store the version information.

The *Description Attribute* on the instances of the *SoftwareVersionType* should be used to provide additional information about the concrete version of the software to the user (e.g. change notes).

The *SoftwareVersionType* is illustrated in Figure 46 and formally defined in Table 73.

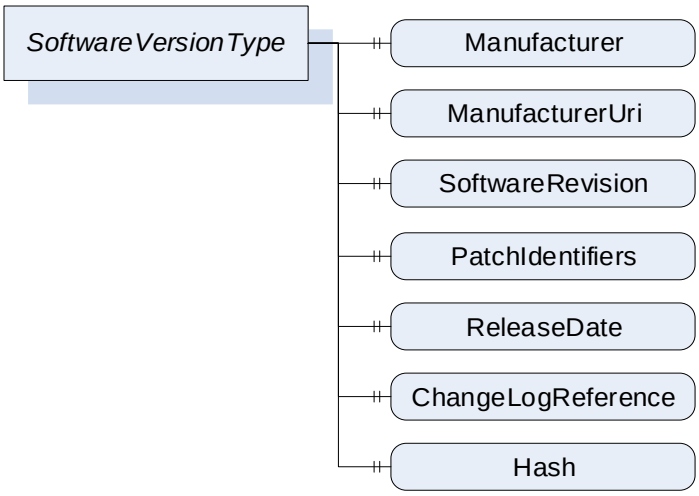

Figure 46 – SoftwareVersionType

Table 73 – SoftwareVersionType definition

Attribute	Value				
BrowseName	SoftwareVersionType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in OPC 10000-5					
HasProperty	Variable	Manufacturer	LocalizedText	PropertyType	Mandatory
HasProperty	Variable	ManufacturerUri	String	PropertyType	Mandatory
HasProperty	Variable	SoftwareRevision	String	PropertyType	Mandatory
HasProperty	Variable	PatchIdentifiers	String[]	PropertyType	Optional
HasProperty	Variable	ReleaseDate	DateTime	PropertyType	Optional
HasProperty	Variable	ChangeLogReference	String	PropertyType	Optional
HasProperty	Variable	Hash	ByteString	PropertyType	Optional
Conformance Units					
DI SU Software Update					

8.4.7.2 Manufacturer

The read only *Manufacturer Property* provides the name of the company that created the software.

In case of the *Pending Version* this shall be empty if there is no pending software to install.

8.4.7.3 ManufacturerUri

The read only *ManufacturerUri Property* provides a unique identifier for the manufacturer of the software.

In case of the *Pending Version* this shall be empty if there is no pending software to install.

8.4.7.4 SoftwareRevision

The read only *SoftwareRevision Property* defines the version of the software. The format and semantics of the string is vendor-specific. *SemanticVersionString* (a sub-type of *String* defined in OPC 10000-5) may be used when using the Semantic Versioning format.

In case of the *Pending Version* this shall be empty if there is no pending software to install.

8.4.7.5 PatchIdentifiers

The read only *PatchIdentifiers Property* identifies the list of patches that are applied to a software version. The format and semantics of the strings are vendor-specific. The order of the strings shall not be relevant.

8.4.7.6 ReleaseDate

The read only *ReleaseDate Property* defines the date when the software is released. If the version information is about patches, this should be the date of the latest patch. It is additional information for the user.

8.4.7.7 ChangeLogReference

The read only *ChangeLogReference Property* may optionally provide a URL to a web site with detailed information about the particular version of the software (change notes). In case of a patched software, the web site should also inform about the patches.

8.4.7.8 Hash

The optional read only *Hash Property* may be read by a *Client* to get the hash of a previously transferred *Software Package*. The hash value needs to be calculated by the *Server* with the SHA-256 algorithm. It can be used to verify if the transferred package matches the one at the *Client*.

8.4.8 PrepareForUpdateStateMachineType

8.4.8.1 Overview

The *PrepareForUpdateStateMachineType* may be used if the device requires to be prepared before the update. Another option is to delay the resuming of normal operation until all update actions are executed. This supports to prepare for update option of 8.3.4.2.

If a *Server* implements this state machine, a *Client* shall use it except if the *UpdateBehavior* indicates that this is not necessary for the transferred software. If preparation is required, the installation is only allowed if the *PrepareForUpdateStateMachine* is in the *PreparedForUpdate* state.

The state machine is illustrated in Figure 47, Figure 48 and formally defined in Table 74. The transitions are formally defined in Table 76.

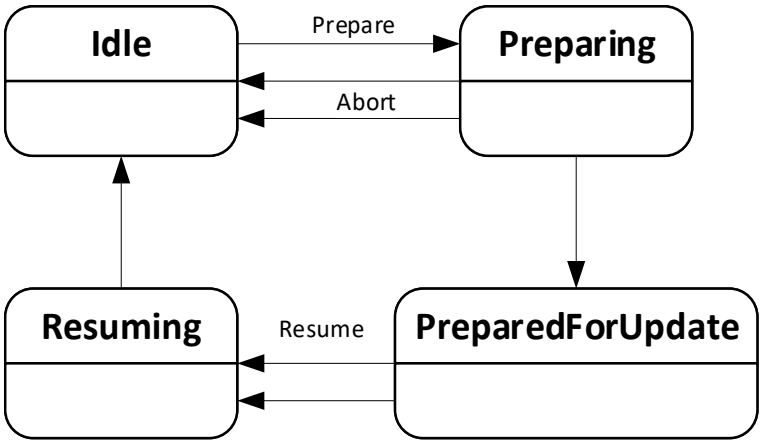

Figure 47 – PrepareForUpdate state machine

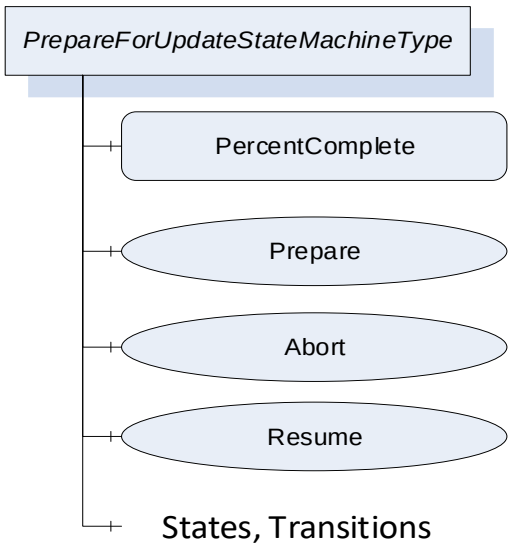

Figure 48 – PrepareForUpdateStateMachineType

Table 74 – PrepareForUpdateStateMachineType definition

Attribute	**Value**				
BrowseName	PrepareForUpdateStateMachineType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FiniteStateMachineType* defined in OPC 10000-5.					
HasComponent	Variable	PercentComplete	Byte	BaseDataVariableType	Optional
HasComponent	Method	Prepare			Mandatory
HasComponent	Method	Abort			Mandatory
HasComponent	Method	Resume			Optional
HasComponent	Object	Idle		InitialStateType	
HasComponent	Object	Preparing		StateType	
HasComponent	Object	PreparedForUpdate		StateType	
HasComponent	Object	Resuming		StateType	
HasComponent	Object	IdleToPreparing		TransitionType	
HasComponent	Object	PreparingToIdle		TransitionType	
HasComponent	Object	PreparingToPreparedForUpdate		TransitionType	
HasComponent	Object	PreparedForUpdateToResuming		TransitionType	
HasComponent	Object	ResumingToIdle		TransitionType	
Conformance Units					
DI SU PrepareForUpdate					

The component *Variables* of the *PrepareForUpdateStateMachineType* have additional *Attributes* defined in Table 75.

Table 75 – PrepareForUpdateStateMachineType Attribute values for child Nodes

BrowsePath	Value Attribute
Idle 0:StateNumber	1
Preparing 0:StateNumber	2
PreparedForUpdate 0:StateNumber	3
Resuming 0:StateNumber	4
IdleToPreparing 0:TransitionNumber	12
PreparingToIdle 0:TransitionNumber	21
PreparingToPreparedForUpdate 0:TransitionNumber	23
PreparedForUpdateToResuming 0:TransitionNumber	34
ResumingToIdle 0:TransitionNumber	41

Table 76 – PrepareForUpdateStateMachineType Additional References

SourceBrowsePath	Reference Type	Is Forward	TargetBrowsePath
Transitions			
IdleToPreparing	FromState	True	Idle
	ToState	True	Preparing
	HasEffect	True	TransitionEventType
PreparingToIdle	FromState	True	Preparing
	ToState	True	Idle
	HasEffect	True	TransitionEventType
PreparingToPreparedForUpdate	FromState	True	Preparing
	ToState	True	PreparedForUpdate
	HasEffect	True	TransitionEventType
PreparedForUpdateToResuming	FromState	True	PreparedForUpdate
	ToState	True	Resuming
	HasEffect	True	TransitionEventType
ResumingToIdle	FromState	True	Resuming
	ToState	True	Idle
	HasEffect	True	TransitionEventType

8.4.8.2 PercentComplete

This percentage is a number between 0 and 100 that informs about the progress in the *Preparing* or the *Resuming States*. It may be used whenever the activity takes longer and the user should be informed about the completion. If the state machine is in *Idle* or *PreparedForUpdate State* it shall have the value 0.

Note: This information is for the user only. It shall not be used to detect completion of the transition.

8.4.8.3 Prepare Method

The *Prepare Method* may be called to prepare a device for an update. This call transitions the device into the state *Preparing*.

After the preparation is complete the state machine may perform an automatic transition to the state *PreparedForUpdate*.

If the preparation cannot complete and the device does not get prepared for update the state machine transitions back to *Idle*. In this case a message with the reason should be provided to the user via the *UpdateStatus*.

Signature

```
Prepare();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidState	If the PrepareForUpdateStateMachineType is not in Idle state.

8.4.8.4 Abort Method

If the preparation takes too long or does not complete at all because the required internal conditions are not met the *Abort Method* may be called to abort the preparation. This call transitions the device back to the *Idle* state.

Note: If the transition from *Preparing* to *Idle* cannot complete instantly a *Client* needs to subscribe for the events or the state variable of the *PrepareForUpdateStateMachine*.

Signature

```
Abort();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidState	If the PrepareForUpdateStateMachineType is not in Preparing state.

8.4.8.5 Resume Method

A call to the optional *Resume Method* transitions the device into the state *Resuming*. After the resuming is complete the state machine performs an automatic transition to the *Idle* state. If the method is not supported, the transitions to *Resuming* and back to *Idle* shall be done by the *Server* automatically. If the method is supported, there shall not be an automatic transition to *Resuming.* Supporting this method enables the *Client* to group several activities like backup, install, restore on a single device or group the update of multiple devices before the devices are allowed to *Resume* their operation again.

Signature

```
Resume();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidState	If the PrepareForUpdateStateMachineType is not in PreparedForUpdate state or if the InstallationStateMachine is still in the state Installing.

8.4.9 InstallationStateMachineType

8.4.9.1 Overview

The *InstallationStateMachineType* may be used if the device supports explicit installation (*Cached-Loading* or *File System based Loading*). This supports the installation option of 8.3.4.6. It is illustrated in Figure 49 and Figure 50 and formally defined in Table 77. The transitions are formally defined in Table 79.

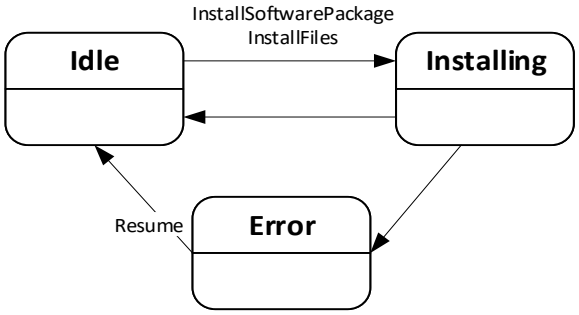

Figure 49 – Installation state machine

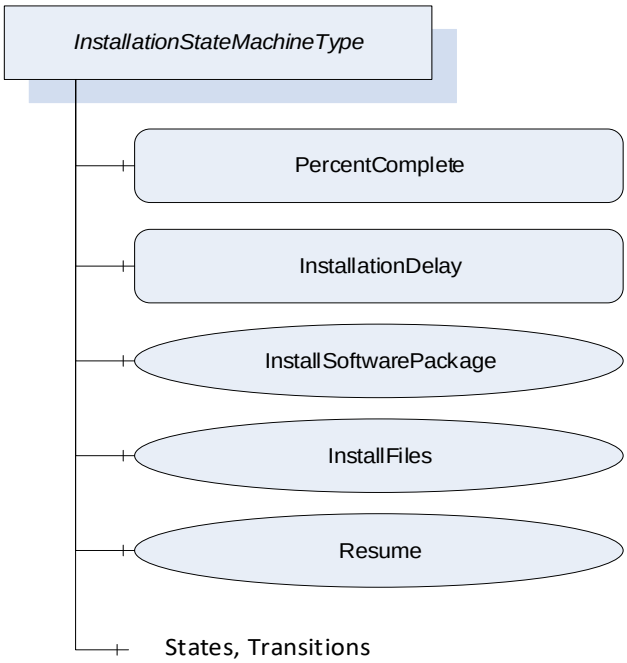

Figure 50 – InstallationStateMachine

Table 77 – InstallationStateMachineType definition

Attribute	Value				
BrowseName	InstallationStateMachineType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FiniteStateMachineType* defined in OPC 10000-5.					
HasComponent	Variable	PercentComplete	Byte	BaseDataVariableType	Optional
HasComponent	Variable	InstallationDelay	Duration	BaseDataVariableType	Optional
HasComponent	Method	InstallSoftwarePackage			Optional
HasComponent	Method	InstallFiles			Optional
HasComponent	Method	Resume			Mandatory
HasComponent	Object	Idle		InitialStateType	
HasComponent	Object	Installing		StateType	
HasComponent	Object	Error		StateType	
HasComponent	Object	IdleToInstalling		TransitionType	
HasComponent	Object	InstallingToIdle		TransitionType	
HasComponent	Object	InstallingToError		TransitionType	
HasComponent	Object	ErrorToIdle		TransitionType	
Conformance Units					
DI SU Software Update					

The component *Variables* of the *InstallationStateMachineType* have additional *Attributes* defined in Table 78.

Table 78 – InstallationStateMachineType Attribute values for child Nodes

BrowsePath	Value Attribute
Idle 0:StateNumber	1
Installing 0:StateNumber	2
Error 0:StateNumber	3
IdleToInstalling 0:TransitionNumber	12
InstallingToIdle 0:TransitionNumber	21
InstallingToError 0:TransitionNumber	23
ErrorToIdle 0:TransitionNumber	31

Table 79 – InstallationStateMachineType Additional References

SourceBrowsePath	Reference Type	Is Forward	TargetBrowsePath
Transitions			
IdleToInstalling	FromState	True	Idle
	ToState	True	Installing
	HasEffect	True	TransitionEventType
InstallingToIdle	FromState	True	Installing
	ToState	True	Idle
	HasEffect	True	TransitionEventType
InstallingToError	FromState	True	Installing
	ToState	True	Error
	HasEffect	True	TransitionEventType
ErrorToIdle	FromState	True	Error
	ToState	True	Idle
	HasEffect	True	TransitionEventType

8.4.9.2 PercentComplete

This percentage is a number between 0 and 100 that informs the user about the progress of an installation. It should be used whenever an update activity takes longer and the user should be informed about the completion. If the state machine is in *Idle State* it shall have the value 0. In case of an error the last value should be kept until the *Resume* is called.

Note: This information is for the user only. It shall not be used to detect completion of the installation.

8.4.9.3 InstallationDelay

The optional *InstallationDelay* can be set by a *Client* to delay the actual installation after the call to *InstallSoftwarePackage* or *InstallFiles* is returned by the *Server*. This can be used when the installation is started on several devices in parallel and there is a risk that a reboot of one device could harm the connection to other devices. With a delay the install methods can be called on all devices before the devices actually start the installation. The *InstallationDelay* does not delay the transition from *Idle* to *Installing*.

This value could be preconfigured. If a *Client* wants to set this value it has to be done before the install method is called.

The *Server* is expected to stay operational at least during the delay.

8.4.9.4 InstallSoftwarePackage Method

With this *Method* the *Client* requests the installation of a *Software Package*. The package can be either the previously transferred *Pending Version* or the alternative *Fallback Version*. To identify the version and to prevent conflicts with a second *Client* that transfers a different version, the *InstallSoftwarePackage Method* needs the *ManufacturerUri*, the *SoftwareRevision and PatchIdentifiers Properties* of the *SoftwareVersionType*.

Optionally an additional hash value may be passed to the *Method*. This hash could be calculated by the *Client* or taken from a trusted source. Before installation the *Server* may compare the hash against the calculated hash of the *Software Package*. This mechanism can be used if there is a risk that the *Software Package* is altered during the transfer to the device and if the *Server* has no other mechanism to ensure that the *Software Package* is from a trustworthy source.

If the installation succeeds but the software cannot run properly this should not be treated as an error of the installation. Instead this should be indicated using the *IDeviceHealthType Interface of the* device / component.

This *Method* shall not return before the state has changed to the *Installing* state.

Signature

```
InstallSoftwarePackage(
  [in] String       ManufacturerUri,
  [in] String       SoftwareRevision,
  [in] String[]     PatchIdentifiers,
  [in] ByteString   Hash);
```

Argument	Description
ManufacturerUri	*ManufacturerUri Property* of either the *Pending* or *Fallback SoftwareVersionType* that should be installed.
SoftwareRevision	*SoftwareRevision Property* of either the *Pending* or *Fallback SoftwareVersionType* that should be installed.
PatchIdentifiers	*PatchIdentifiers Property* of either the *Pending* or *Fallback SoftwareVersionType* that should be installed. (or empty array if not supported on the SoftwareVersionType instance).
Hash	Hash of the *Software Package* that should be installed (or empty if not used).

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidState	If the *InstallationStateMachineType* is not in *Idle* state or if the *PrepareForUpdate Object* is available and the *PrepareForUpdate* state machine is not in the state *PreparedForUpdate*.
Bad_NotFound	If the specified *Software Package* does not exist.
Bad_InvalidArgument	If the Hash does not match the calculated hash of the *Software Package*.

8.4.9.5 InstallFiles Method

This *Method* may be called to request the installation of one or more files. The files are identified by the *NodeId* of their *FileType* instance in the *FileSystem*.

If the installation succeeds but the software cannot run properly this should not be treated as an error of the installation. Instead this should be indicated using the *IDeviceHealthType Interface of the* device / component.

Signature

```
InstallFiles(
  [in] NodeId[]          NodeIds);
```

Argument	Description
NodeIds	NodeIds of the files to install.

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidState	If the *InstallationStateMachineType* is not in *Idle* state or if the *PrepareForUpdate Object* is available and the *PrepareForUpdate* state machine is not in the state *PreparedForUpdate*.
Bad_NotFound	If one or more NodeIds are not found.

8.4.9.6 Resume Method

This *Method* may be called to resume from the *Error* state. The *Error* state can be reached if there are issues during the installation. The state machine remains in this state until the *Client* calls the *Resume Method* to get back to the *Idle* state immediately.

Signature

```
Resume();
```

Method Result Codes (defined in Call Service)

Result Code	Description
Bad_InvalidState	If the *InstallationStateMachineType* is not in *Error* state.

8.4.10 PowerCycleStateMachineType

The *PowerCycleStateMachineType* is used to inform the user to perform a manual power cycle.

When the server needs a manual power cycle it indicates that to the client by changing the state to *WaitingForPowerCycle*. After restart of the device it transitions to *NotWaitingForPowerCycle* automatically.

There are no methods, all transitions originate from the installation process. The state machine is illustrated in Figure 51 and formally defined in Table 80. The transitions are formally defined in Table 82.

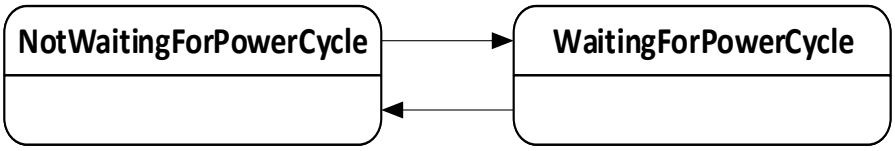

Figure 51 – PowerCycle state machine

Table 80 – PowerCycleStateMachineType definition

Attribute	**Value**				
BrowseName	PowerCycleStateMachineType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FiniteStateMachineType* defined in OPC 10000-5.					
HasComponent	Object	NotWaitingForPowerCycle		InitialStateType	
HasComponent	Object	WaitingForPowerCycle		StateType	
HasComponent	Object	NotWaitingForPowerCycleTo WaitingForPowerCycle		TransitionType	
HasComponent	Object	WaitingForPowerCycleToNot WaitingForPowerCycle		TransitionType	
Conformance Units					
DI SU Manual Power Cycle					

The component *Variables* of the *PowerCycleStateMachineType* have additional *Attributes* defined in Table 81.

Table 81 – PowerCycleStateMachineType Attribute values for child Nodes

BrowsePath	**Value Attribute**
NotWaitingForPowerCycle 0:StateNumber	1
WaitingForPowerCycle 0:StateNumber	2
NotWaitingForPowerCycleToWaitingForPowerCycle 0:TransitionNumber	12
WaitingForPowerCycleToNotWaitingForPowerCycle 0:TransitionNumber	21

Table 82 – PowerCycleStateMachineType Additional References

SourceBrowsePath	Reference Type	Is Forward	TargetBrowsePath
Transitions			
NotWaitingForPowerCycleToWaitingForPowerCycle	FromState	True	NotWaitingForPowerCycle
	ToState	True	WaitingForPowerCycle
	HasEffect	True	TransitionEventType
WaitingForPowerCycleToNotWaitingForPowerCycle	FromState	True	WaitingForPowerCycle
	ToState	True	NotWaitingForPowerCycle
	HasEffect	True	TransitionEventType

8.4.11 ConfirmationStateMachineType

8.4.11.1 Overview

The *ConfirmationStateMachineType* is used to prove a valid *Client* – *Server* connection after a restart of the OPC UA *Server*. This supports the confirmation option of 8.3.4.9.

If several instances of this state machine are provided on a device (due to several instances of the *SoftwareUpdateType*), all instances should behave as if it is only a single instance. In particular it is sufficient to call one of the confirm methods after reboot.

The *ConfirmationStateMachineType* is illustrated in Figure 52 and Figure 53 and formally defined in Table 83. The transitions are formally defined in Table 85.

Figure 52 – Confirmation state machine

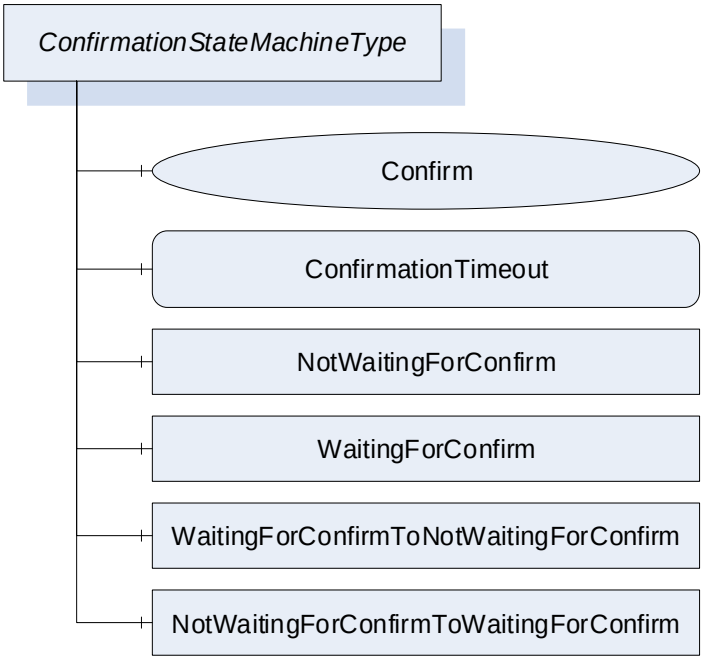

Figure 53 – ConfirmationStateMachineType

Table 83 – ConfirmationStateMachineType

Attribute	**Value**				
BrowseName	ConfirmationStateMachineType				
IsAbstract	False				
References	**Node Class**	**BrowseName**	**DataType**	**TypeDefinition**	**Modelling Rule**
Subtype of the *FiniteStateMachineType* defined in OPC 10000-5.					
HasComponent	Method	Confirm			Mandatory
HasComponent	Variable	ConfirmationTimeout	Duration	BaseDataVariableType	Mandatory
HasComponent	Object	NotWaitingForConfirm		InitialStateType	
HasComponent	Object	WaitingForConfirm		StateType	
HasComponent	Object	NotWaitingForConfirmToWaitingForConfirm		TransitionType	
HasComponent	Object	WaitingForConfirmToNotWaitingForConfirm		TransitionType	
Conformance Units					
DI SU Update Confirmation					

The component *Variables* of the *ConfirmationStateMachineType* have additional *Attributes* defined in Table 84.

Table 84 – ConfirmationStateMachineType Attribute values for child Nodes

BrowsePath	Value Attribute
NotWaitingForConfirm	1
0:StateNumber	
WaitingForConfirm	2
0:StateNumber	
NotWaitingForConfirmToWaitingForConfirm	12
0:TransitionNumber	
WaitingForConfirmToNotWaitingForConfirm	21
0:TransitionNumber	

Table 85 – ConfirmationStateMachineType TargetBrowsePath

SourceBrowsePath	Reference Type	Is Forward	TargetBrowsePath
Transitions			
NotWaitingForConfirmToWaitingForConfirm	FromState	True	NotWaitingForConfirm
	ToState	True	WaitingForConfirm
	HasEffect	True	TransitionEventType
WaitingForConfirmToNotWaitingForConfirm	FromState	True	WaitingForConfirm
	ToState	True	NotWaitingForConfirm
	HasEffect	True	TransitionEventType

8.4.11.2 ConfirmationTimeout

The *ConfirmationTimeout* may be set by a *Client* to a value other then 0 to enable the confirmation feature. If the value is not 0 and the *Client – Server* connection is lost, the *ConfirmationTimeout* represents the maximum time that the *Client* may need to reconnect and call the *Confirm Method*. The *Server* shall automatically reset the value to 0 when the installation is complete.

8.4.11.3 Confirm Method

After a reboot and with a *ConfirmationTimeout* other than 0 a *Client* shall call this *Method* to inform the *Server* that it has successfully reconnected. If this *Method* is not called after a lost connection the *Server* shall regard the update as unsuccessful and shall revert it. A *Client* needs to react within the time specified in the *ConfirmationTimeout Variable*.

Signature

```
Confirm();
```

8.5 DataTypes

8.5.1 SoftwareVersionFileType

This enumeration is used to identify the version in the methods of the *TemporaryFileTransferType* that is used in the *PackageLoadingType* (8.4.3). The *Enumeration* is defined in Table 86.

Table 86 – SoftwareVersionFileType Items

Name	Value	Description
Current	0	The currently used version of the software identified by the *CurrentVersion Object*.
Pending	1	The *Pending Version* of the software that could be installed identified by the *PendingVersion Object*.
Fallback	2	The *Fallback Version* of the software identified by the *FallbackVersion Object*.

8.5.2 UpdateBehavior OptionSet

The *UpdateBehavior OptionSet* is based on UInt32. It describes how the device can perform the update. All possible options are described in Table 87. All other values are reserved for future versions of this specification. The *OptionSet* is used in the *UpdateBehavior Property* of the *DirectLoadingType* (8.4.4.3) and in the *GetUpdateBehavior Methods* on the *CachedLoadingType* (8.4.5.5) and in the *FileSystemLoadingType* (8.4.6.3).

Table 87 – UpdateBehavior OptionSet

Value	Bit No.	Description
KeepsParameters	0	If KeepsParameters is not set, the device will lose its configuration during update. The *Client* should do a backup of the parameters before the update and restore them afterwards.
WillDisconnect	1	If WillDisconnect is set, the OPC UA *Server* will restart during installation. This can be the case if the update is about the firmware of the device that hosts the OPC UA *Server*.
RequiresPowerCycle	2	If RequiresPowerCycle is set, the devices require a manual power off / power on for installation.
WillReboot	3	If WillReboot is set, the device will reboot during the update, inclusive of embedded infrastructure elements like an integrated switch. An update *Client* should take this into account since the devices behind an integrated switch are not reachable for that time.
NeedsPreparation	4	If NeedsPreparation is not set, the *Client* can install the update without maintaining the PrepareForUpdateStateMachine. This can be used to support an installation without stopping the software.

The *UpdateBehavior OptionSet* representation in the *AddressSpace* is defined in Table 88.

Table 88 – UpdateBehavior OptionSet Definition

Attribute	Value				
BrowseName	UpdateBehavior				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Other**
Subtype of UInt32 defined in OPC 10000-5.					
HasProperty	Variable	OptionSetValues	LocalizedText []	PropertyType	
Conformance Units					
DI SU Software Update					

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 54 illustrates these principles.

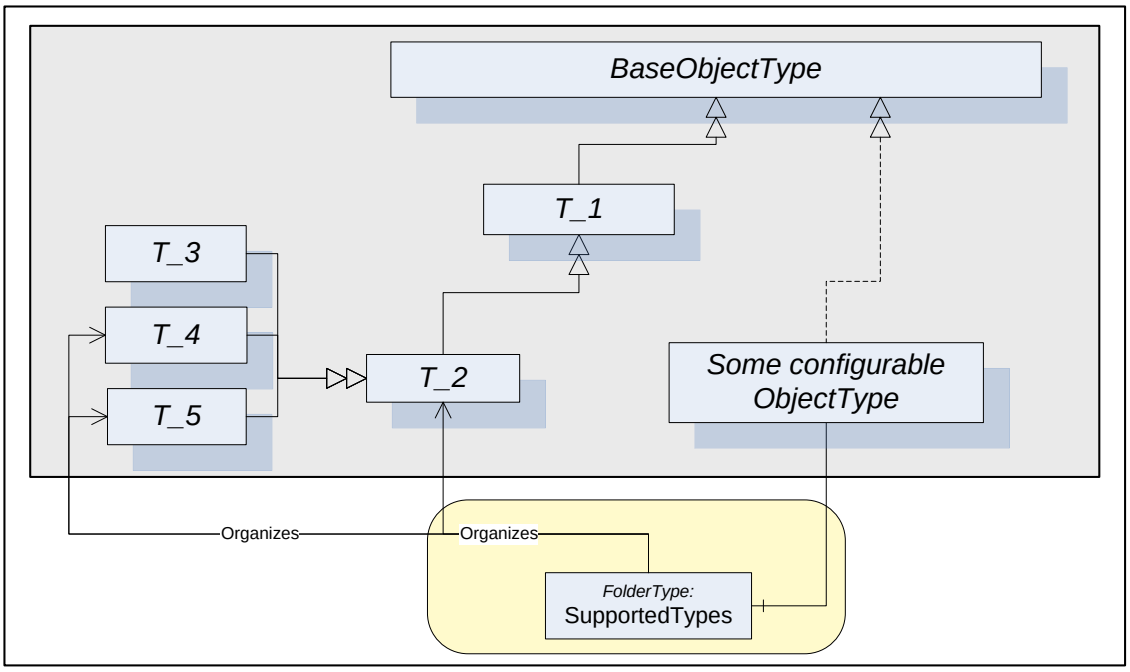

Figure 54 – Configurable component pattern

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.

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 55 illustrates the *ConfigurableObjectType*. It is formally defined in Table 89. Concrete examples are in Clauses 9.3 and 9.4.

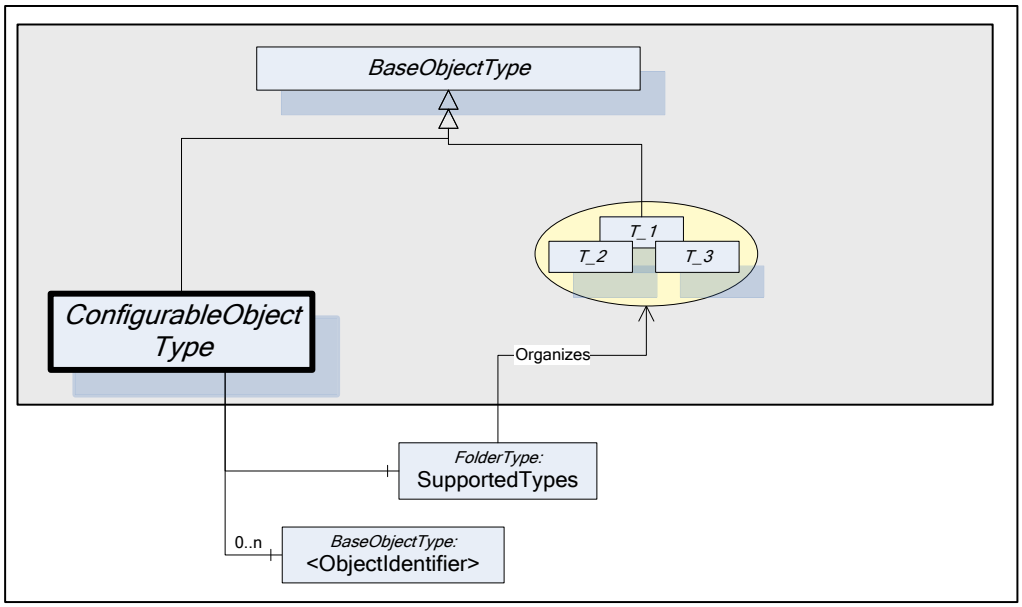

Figure 55 – ConfigurableObjectType

Table 89 – 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
Conformance Units					
DI Information Model					

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 56 shows the general structure of block-oriented *Devices*.

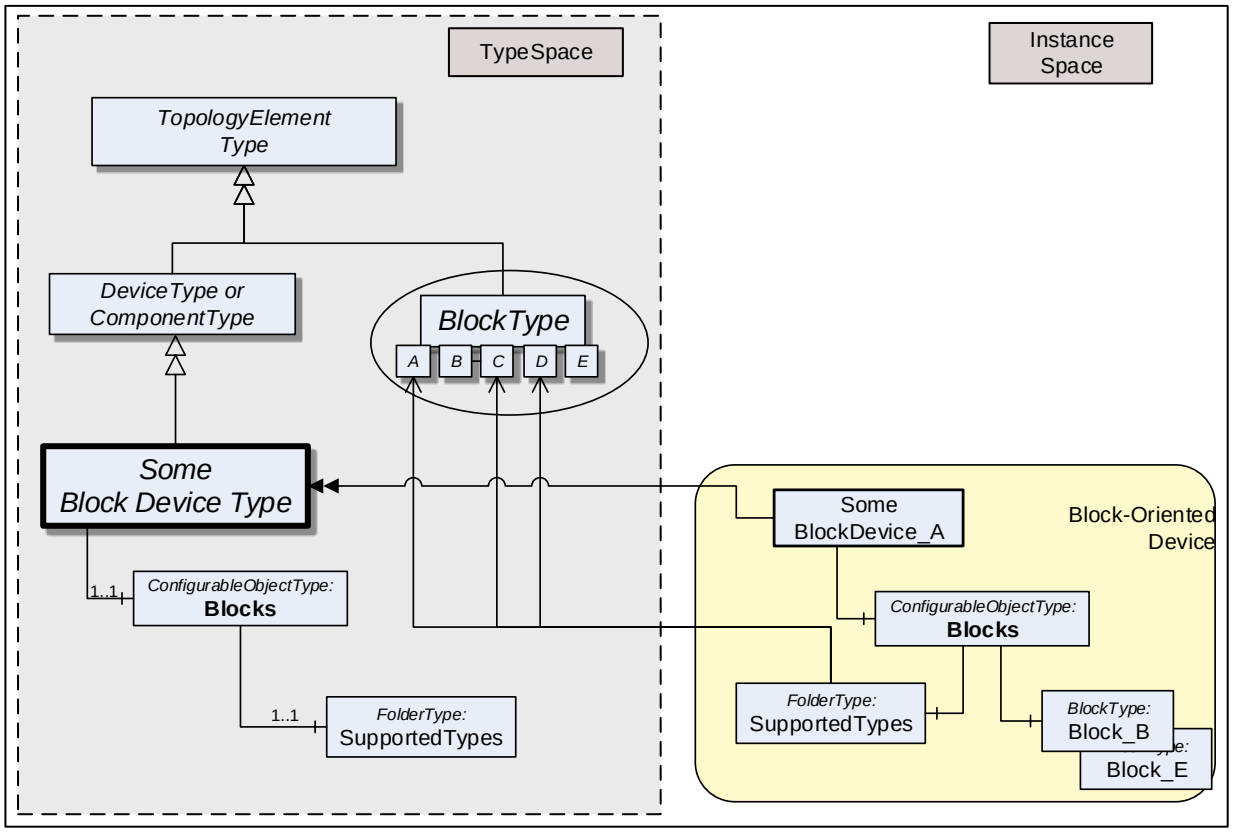

Figure 56 – Block-oriented Device structure example

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 56 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 57 shows the general structure of *Modular Devices*.

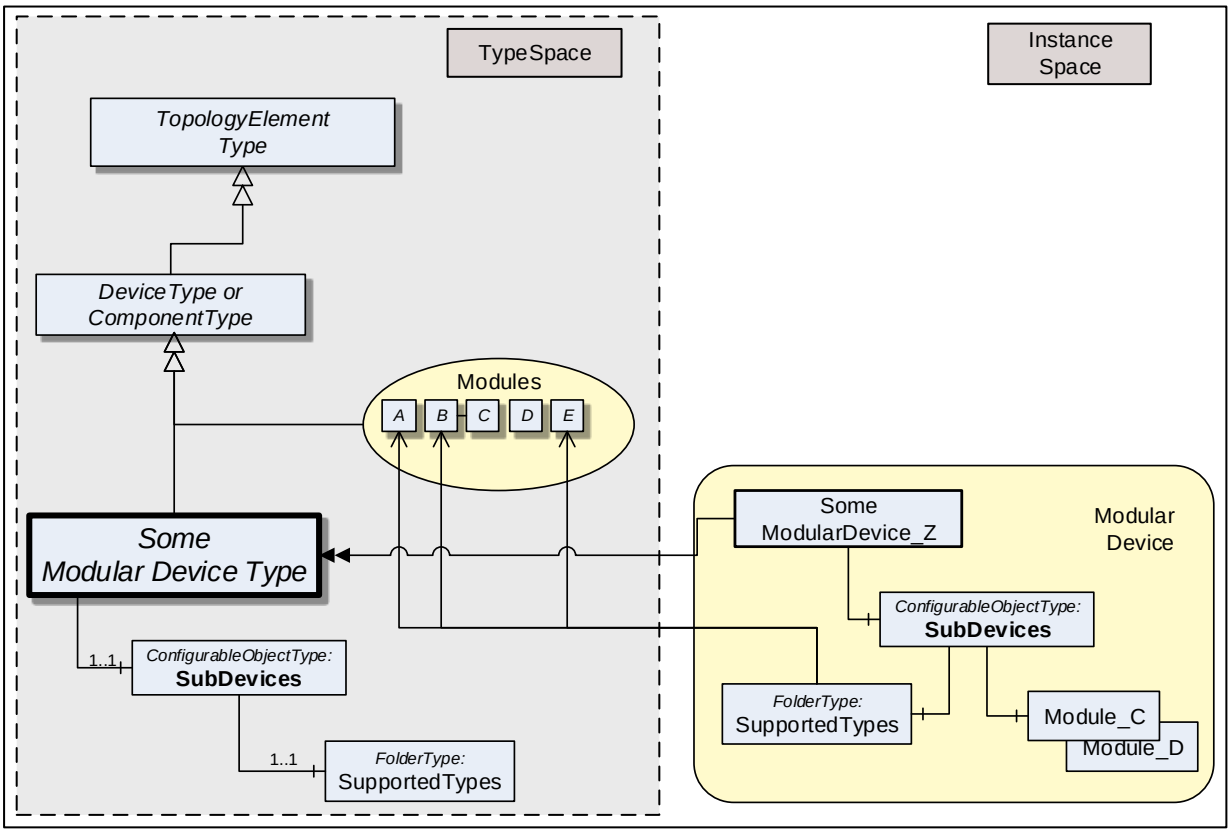

Figure 57 – Modular Device structure example

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 57 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 and ConformanceUnits

10.1 Conformance Units

Table 90 defines the corresponding *Conformance Units* for the OPC UA Information Model for Devices.

Table 90 – Conformance Units for Devices

Category	Title	Description
Server	DI Information Model	Supports *Objects* that conform to the *Device* model of this document. This includes in particular *Objects* of (subtypes of) *ComponentType* and *FunctionalGroups*.
Server	DI DeviceType	Supports *Objects* of *DeviceType* or a subtype.
Server	DI DeviceSet	Supports the **DeviceSet** object to aggregate *Device* instances.
Server	DI Nameplate	Supports *Properties* of the *VendorNameplate Interface* defined in 4.5.2.
Server	DI TagNameplate	Supports the *TagNameplate Interface* defined in 4.5.3.
Server	DI Software Component	Supports *Objects* of *SoftwareType* or a subtype.
Server	DI DeviceHealth	Supports the DeviceHealth Interface defined in 4.5.4.
Server	DI DeviceHealthProperty	Supports the DeviceHealth Property defined in 4.5.4.
Server	DI HealthDiagnosticsAlarm	Supports DeviceHealth Alarms defined in 4.12.
Server	DI DeviceSupportInfo	*Server* provides additional data for its *Devices* as defined in 4.5.5.
Server	DI Identification	Supports the **Identification** *FunctionalGroup* for *Devices*.
Server	DI Protocol	Supports the *ProtocolType* and instances of it to identify the used communication profiles for specific instances.
Server	DI Blocks	Supports the *BlockType* (or subtypes respectively) and the *Blocks Object* in some of the instantiated *Devices*.
Server	DI Locking	Supports the *LockingService* for certain *TopologyElements*.
Server	DI BreakLocking	Supports the *BreakLock Method* to break the lock held by another *Client*.
Server	DI Network	Supports the *NetworkType* to instantiate *Network* instances.
Server	DI ConnectionPoint	Supports subtypes of the *ConnectionPointType*.
Server	DI NetworkSet	Supports the *NetworkSet Object* to aggregate all *Network* instances.
Server	DI ConnectsTo	Supports the *ConnectsTo Reference* to associate *Devices* with a *Network*.
Server	DI DeviceTopology	Supports the *DeviceTopology Object* as starting *Node* for the communication topology of the *Devices* to integrate.
Server	DI Offline	Supports offline and online representations of *Devices* including the *Methods* to transfer data from or to the *Device*.
Server	DI SU Software Update	The Address Space contains at least one instance of the *SoftwareUpdateType* as *AddIn* and provides the required *Parameters* of *IVendorNamePlateType* as defined in 8.4.1.
Server	DI SU DirectLoading	At least one instance of the *SoftwareUpdateType* supports the *DirectLoadingType* as *Loading Object*.
Server	DI SU CachedLoading	At least one instance of the *SoftwareUpdateType* supports the *CachedLoadingType* as *Loading Object*.
Server	DI SU FileSystem Loading	At least one instance of the *SoftwareUpdateType* supports the *FileSystemLoadingType* as *Loading Object*.
Server	DI SU PrepareForUpdate	At least one instance of the *SoftwareUpdateType* supports the *PrepareForUpdate Object*.
Server	DI SU Manual Power Cycle	At least one instance of the *SoftwareUpdateType* supports the *PowerCycle Object*.
Server	DI SU Update Parameter Backup	At least one instance of the *SoftwareUpdateType* supports the *Parameters Object*.
Server	DI SU UpdateStatus	At least one instance of the *SoftwareUpdateType* supports the *UpdateStatus Variable*.
Server	DI SU VendorErrorCode	At least one instance of the *SoftwareUpdateType* supports the *VendorErrorCode Variable*.
Server	DI SU Installation for Cached Loading	At least one instance of the *SoftwareUpdateType* supports the *Installation Object*. The *Method InstallSoftwarePackage* is mandatory. The *Method InstallFiles* shall not be available.
Server	DI SU Installation for File System	At least one instance of the *SoftwareUpdateType* supports the *Installation Object* of *SoftwareUpdateType*. The *Method InstallFiles* is mandatory. The *Method InstallSoftwarePackage* shall not be available.
Server	DI SU InstallationDelay	At least one instance of the *InstallationStateMachineType* supports the *InstallationDelay Variable*.
Server	DI SU Update Confirmation	At least one instance of the *SoftwareUpdateType* supports the *Confirmation Object*.
Server	DI SU FallbackVersion	At least one instance of the *CachedLoadingType* supports the *FallbackVersion Object*.
Server	DI SU UpdateKey	At least one instance of the *SoftwareLoadingType* supports the *UpdateKey Variable*.

Category	Title	Description
Server	DI SU Installation PercentComplete	At least one instance of the *InstallationStateMachineType* supports the *PercentComplete Variable.*
Server	DI SU Resume Update	At least one instance of the *PrepareForUpdateStateMachineType* supports the *Resume Method*.
Server	DI SU Prepare for Update PercentComplete	At least one instance of the *PrepareForUpdateStateMachineType* supports the *PercentComplete Variable*.
Server	DI SU Update WriteBlockSize	At least one instance of a subtype of the *PackageLoadingType* supports the *WriteBlockSize Variable.*
Server	DI SU Update WriteTimeout	At least one instance of *DirectLoadingType* supports the *WriteTimeout Variable.*
Server	DI SU PatchIdentifiers	At least one instance of the *SoftwareVersionType* support the *PatchIdentifiers Property.* If implemented on a *SoftwareUpdate Object*, all supported versions (*CurrentVersion*, *PendingVersion* and *FallbackVersion*) shall support the *Property*.
Server	DI SU Update ReleaseDate	At least one instance of *SoftwareVersionType* of a *SoftwareUpdate Object* supports the *ReleaseDate Property.*
Server	DI SU ChangeLogReference	At least one instance of *SoftwareVersionType* of a *SoftwareUpdate Object* supports the *ChangeLogReference Property*.
Server	DI SU Update Hash	At least one instance of *SoftwareVersionType* of a *SoftwareUpdate Object* supports the *Hash Property.*
Server	DI SU ValidateFiles	At least one instance of the *FileSystemLoadingType* supports the *ValidateFiles Method*.
Client	DI Client Information Model	Consumes *Objects* that conform to the *Device* model in this document. This includes in particular *Objects* of (subtypes of) *ComponentType* and *FunctionalGroups*.
Client	DI Client DeviceSet	Uses the **DeviceSet** *Object* to detect available *Devices*.
Client	DI Client Nameplate	Consumes *Properties* of the *VendorNameplate Interface* defined in 4.5.2.
Client	DI Client TagNameplate	Consumes the *VendorNameplate Interface* defined in 4.5.3.
Client	DI Client Software Component	Consumes *Objects* of *SoftwareType* or a subtype.
Client	DI Client DeviceHealth	Uses the *DeviceHealth Interface* defined in 4.5.4.
Client	DI Client DeviceHealthProperty	Uses the DeviceHealth Property defined in 4.5.4.
Client	DI Client HealthDiagnosticsAlarm	Uses DeviceHealth Alarms defined in 4.12.
Client	DI Client DeviceSupportInfo	Uses available additional data for *Devices* as defined in 4.5.5.
Client	DI Client Identification	Consumes the **Identification** *FunctionalGroup* for *Devices* including the (optional) reference to supported protocol(s).
Client	DI Client Blocks	Understands and uses *BlockDevices* and their *Blocks* including *FunctionalGroups* on both *Device* and *Block* level.
Client	DI Client Locking	Uses the *LockingService* where available.
Client	DI Client BreakLocking	Support use of the *BreakLock Method* to break the lock held by another *Client*.
Client	DI Client Network	Uses the *NetworkType* to instantiate *Network* instances.
Client	DI Client ConnectionPoint	Uses subtypes of the *ConnectionPointType*.
Client	DI Client NetworkSet	Uses the *NetworkSet Object* to store or find *Network* instances.
Client	DI Client ConnectsTo	Uses the *ConnectsTo Reference* to associate *Devices* with a *Network*.
Client	DI Client DeviceTopology	Uses the *DeviceTopology Object* as starting *Node* for the communication topology of the *Devices* to integrate.
Client	DI Client Offline	Uses offline and online representations of *Devices* including the *Methods* to transfer data from or to the *Device*.
Client	DI SU Client SoftwareUpdate	Uses the *IVendorNameplate* and the *SoftwareUpdate AddIn* to perform a software update.
Client	DI SU Client DirectLoading	Can use the *DirectLoadingType* to update the software using *Direct-Loading* if supported by the server.
Client	DI SU Client CachedLoading	Uses the *CachedLoadingType and InstallationStateMachineType* to update the software using *Cached-Loading* if supported by the server.
Client	DI SU Client FileSystem Loading	Uses the *FileSystemLoadingType and InstallationStateMachineType* to update the software using *FileSystem based Loading* if supported by the server.
Client	DI SU Client PrepareForUpdate	Uses the *PrepareForUpdate Object* of *SoftwareUpdateType* if supported by the server.
Client	DI SU Client Manual Power Cycle	Uses the *PowerCycle Object* of *SoftwareUpdateType* if supported by the server.
Client	DI SU Client Update Parameter Backup	Uses the *Parameters Object* of *SoftwareUpdateType* if supported by the server.
Client	DI SU Client Update Confirmation	Can use the *Confirmation Object* of *SoftwareUpdateType* if supported by the server.
Client	DI SU Client FallbackVersion	Supports the installation of the *Fallback Version* if supported by the server.
Client	DI SU Client UpdateKey	Supports update of devices that need an *UpdateKey* if supported by the server.
Client	DI SU Client Resume Update	Can use the *Resume Method* on the *PrepareForUpdate Object* of *SoftwareUpdateType* if supported by the server.

Category	Title	Description
Client	DI SU Client WriteBlockSize	Respects the *WriteBlockSize* of *PackageLoadingType* if supported by the server.
Client	DI SU Client Update Hash	Can provide the *Hash* value to the *Install Method* for verification.
Client	DI SU Client ValidateFiles	Uses the *ValidateFiles Method* of the *InstallationStateMachineType* if supported by the server.

10.2 Profiles

10.2.1 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 10.2.3 and 10.2.4.

10.2.2 Profile list

Table 91 lists all Profiles defined in this document and defines their URIs.

Table 91 – Profile URIs for Devices

Profile	URI
DI BaseDevice Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/BaseDevice
DI DeviceIdentification Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/DeviceIdentification
DI BlockDevice Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/BlockDevice
DI Locking Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/Locking
DI DeviceCommunication Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/DeviceCommunication
DI DeviceIntegrationHost Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/DeviceIntegrationHost
DI SU Software Update Base Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/SoftwareUpdateBase
DI SU Direct Loading Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/DirectLoading
DI SU Cached Loading Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/CachedLoading
DI SU FileSystem Loading Server Facet	http://opcfoundation.org/UA-Profile/DI/Server/FileSystemLoading
DI BaseDevice Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/BaseDevice
DI DeviceIdentification Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/DeviceIdentification
DI BlockDevice Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/BlockDevice
DI Locking Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/Locking
DI DeviceCommunication Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/DeviceCommunication
DI DeviceIntegrationHost Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/DeviceIntegrationHost
DI SU Software Update Base Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/SoftwareUpdateBase
DI SU Direct Loading Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/DirectLoading
DI SU Cached Loading Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/CachedLoading
DI SU FileSystem Loading Client Facet	http://opcfoundation.org/UA-Profile/DI/Client/FileSystemLoading

10.2.3 Device Server Facets

The following tables specify the *Facets* available for *Servers* that implement the *Devices* information model. Table 92 describes *Conformance Units* included in the minimum needed *Facet*. It includes the organisation of instantiated *Devices* in the *Server AddressSpace*.

Table 92 – DI BaseDevice Server Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Information Model	M
DI	DI DeviceSet	M
DI	DI DeviceType	O
DI	DI Nameplate	O
DI	DI TagNameplate	O
DI	DI Software Component	O
DI	DI DeviceHealth	O
DI	DI DeviceHealthProperty	O
DI	DI HealthDiagnosticsAlarm	O
DI	DI DeviceSupportInfo	O

Table 93 defines a *Facet* for the identification *FunctionalGroup* of *Devices*. This includes the option of identifying the *Protocol*(s).

Table 93 – DI DeviceIdentification Server Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Identification	M
DI	DI Protocol	O

Table 94 defines extensions specifically needed for *BlockDevices*.

Table 94 – DI BlockDevice Server Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Blocks	M

Table 95 defines a *Facet* for the Locking *AddIn Capability*. This includes the option of breaking a lock.

Table 95 – DI Locking Server Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Locking	M
DI	DI BreakLocking	O

Table 96 defines a *Facet* for the support of the Device Communication model.

Table 96 – DI DeviceCommunication Server Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Network	M
DI	DI ConnectionPoint	M
DI	DI NetworkSet	M
DI	DI ConnectsTo	M

Table 97 defines a *Facet* for the support of the Device Integration Host model.

Table 97 – DI DeviceIntegrationHost Server Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI DeviceTopology	M
DI	DI Offline	M

Table 98 defines a *Facet* that describes the basic infrastructure for software update. It contains the common part of the Direct Loading, Cached Loading and FileSystem Loading *Server Profiles*.

Table 98 – DI SU Software Update Base Server Facet

Group	Conformance Unit / Profile Title	M / O
DI	DI SU Software Update	M
DI	DI SU PrepareForUpdate	O
DI	DI SU Resume Update	O
DI	DI SU Prepare for Update PercentComplete	O
DI	DI SU Manual Power Cycle	O
DI	DI SU Update Parameter Backup	O
DI	DI SU UpdateKey	O

Table 99 defines a *Facet* with additional *Conformance Units* for a *Server* that implements *Direct-Loading*.

Table 99 – DI SU Direct Loading Server Facet

Group	Conformance Unit / Profile Title	M / O
Profile	DI SU Software Update Base Server Facet	M
DI	DI SU DirectLoading	M
DI	DI SU UpdateStatus	M
DI	DI SU Update WriteBlockSize	O
DI	DI SU Update WriteTimeout	O
DI	DI SU PatchIdentifiers	O
DI	DI SU Update ReleaseDate	O
DI	DI SU ChangeLogReference	O
DI	DI SU Update Hash	O

Table 100 defines a *Facet* with additional *Conformance Units* for a *Server* that implements *Cached-Loading*.

Table 100 – DI SU Cached Loading Server Facet

Group	Conformance Unit / Profile Title	M / O
Profile	DI SU Software Update Base Server Facet	M
DI	DI SU CachedLoading	M
DI	DI SU Installation for Cached Loading	M
DI	DI SU UpdateStatus	M
DI	DI SU Installation PercentComplete	O
DI	DI SU InstallationDelay	O
DI	DI SU Update Confirmation	O
DI	DI SU FallbackVersion	O
DI	DI SU Update WriteBlockSize	O
DI	DI SU PatchIdentifiers	O
DI	DI SU Update ReleaseDate	O
DI	DI SU ChangeLogReference	O
DI	DI SU Update Hash	O

Table 101 defines a *Facet* with additional *Conformance Units* for a *Server* that implements *File System based Loading*.

Table 101 – DI SU FileSystem Loading Server Facet

Group	Conformance Unit / Profile Title	M / O
Profile	DI SU Software Update Base Server Facet	M
DI	DI SU FileSystem Loading	M
DI	DI SU Installation for File System	M
DI	DI SU UpdateStatus	O
DI	DI SU Installation PercentComplete	O
DI	DI SU InstallationDelay	O
DI	DI SU Update Confirmation	O
DI	DI SU Validate Files	O

10.2.4 Device Client Facets

The following tables specify the *Facets* available for *Clients* that implement the *Devices* information model. Table 102 describes *Conformance Units* included in the minimum needed *Facet*.

Table 102 – DI BaseDevice Client Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Client Information Model	M
DI	DI Client DeviceSet	M
DI	DI Client Nameplate	O
DI	DI Client Software Component	O
DI	DI Client DeviceHealth	O
DI	DI DeviceHealthProperty	O
DI	DI HealthDiagnosticsAlarm	O
DI	DI Client DeviceSupportInfo	O

Table 103 defines a *Facet* for the **identification** *FunctionalGroup* of *Devices*. This includes the option of identifying the *Protocol*(s).

Table 103 – DI DeviceIdentification Client Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Client Identification	M

Table 104 defines extensions specifically needed for *BlockDevices*.

Table 104 – DI BlockDevice Client Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Client Blocks	M

Table 105 defines a *Facet* for the Locking *AddIn Capability*. This includes the option of breaking a lock.

Table 105 – DI Locking Client Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Client Locking	M
DI	DI Client BreakLocking	O

Table 106 defines a *Facet* for the use of the Device Communication model.

Table 106 – DI DeviceCommunication Client Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Client Network	M
DI	DI Client ConnectionPoint	M
DI	DI Client NetworkSet	M
DI	DI Client ConnectsTo	M

Table 107 defines a *Facet* for the use of the Device Integration Host model.

Table 107 – DI DeviceIntegrationHost Client Facet definition

Group	Conformance Unit / Profile Title	M / O
DI	DI Client DeviceTopology	M
DI	DI Client Offline	M

Table 98 defines a *Facet* that describes the basic features of a software update client. It contains the common part of the Direct Loading, Cached Loading and FileSystem Loading *Client Profiles*.

Table 108 – DI SU Software Update Base Client Facet

Group	Conformance Unit / Profile Title	M / O
DI	DI SU Client SoftwareUpdate	M
DI	DI SU Client PrepareForUpdate	O
DI	DI SU Client Resume Update	O
DI	DI SU Client Manual Power Cycle	O
DI	DI SU Client Update Parameter Backup	O
DI	DI SU Client UpdateKey	O

Table 99 defines a *Facet* with additional *Conformance Units* for a *Client* that supports *Direct-Loading*.

Table 109 – DI SU Direct Loading Client Facet

Group	Conformance Unit / Profile Title	M / O
Profile	DI SU Software Update Base Client Facet	M
DI	DI SU Client DirectLoading	M
DI	DI SU Client WriteBlockSize	O
DI	DI SU Client Update Hash	O

Table 100 defines a *Facet* with additional *Conformance Units* for a *Client* that supports *Cached-Loading*.

Table 110 – DI SU Cached Loading Client Facet

Group	Conformance Unit / Profile Title	M / O
Profile	DI SU Client SoftwareUpdate	M
DI	DI SU Client CachedLoading	M
DI	DI SU Client Update Confirmation	O
DI	DI SU Client FallbackVersion	O
DI	DI SU Client WriteBlockSize	O
DI	DI SU Client Update Hash	O

Table 101 defines a *Facet* with additional *Conformance Units* for a *Client* that supports *File System based Loading*.

Table 111 – DI SU FileSystem Loading Client Facet

Group	Conformance Unit / Profile Title	M / O
Profile	DI SU Client SoftwareUpdate	M
DI	DI SU Client FileSystem Loading	M
DI	DI SU Client Update Confirmation	O
DI	DI SU Client ValidateFiles	O

11 Namespaces

11.1 Namespace Metadata

Table 112 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 112 – NamespaceMetadata Object for this Specification

Attribute	**Value**	
BrowseName	http://opcfoundation.org/UA/DI/	
Property	**DataType**	**Value**
0:NamespaceUri	0:String	http://opcfoundation.org/UA/DI/
0:NamespaceVersion	0:String	1.03.1
0:NamespacePublicationDate	0:DateTime	2021-09-07
0:IsNamespaceSubset	0:Boolean	False
0:StaticNodeIdTypes	0:IdType[]	0
0:StaticNumericNodeIdRange	0:NumericRange[]	
0:StaticStringNodeIdPattern	0:String	

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

Table 113 – 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 114 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 114 – 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/1.3/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/1.3/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:*

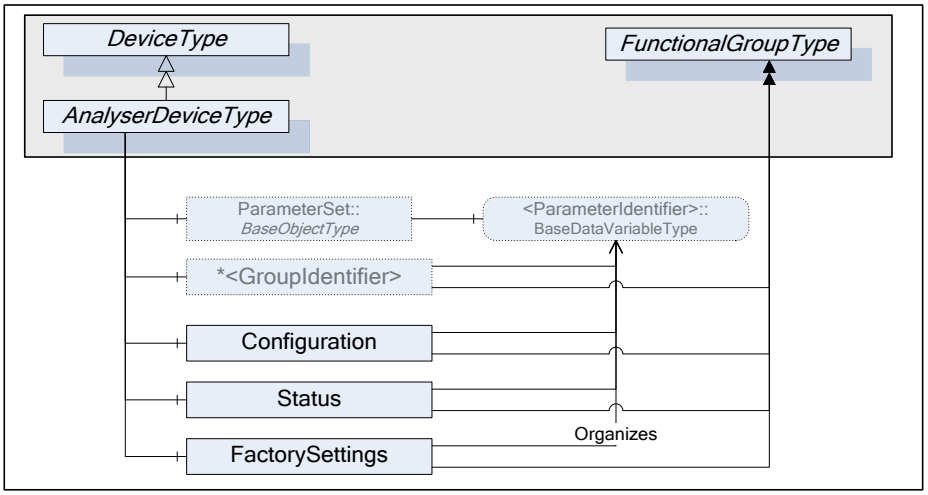

Figure B.1 – Analyser Device use for FunctionalGroups

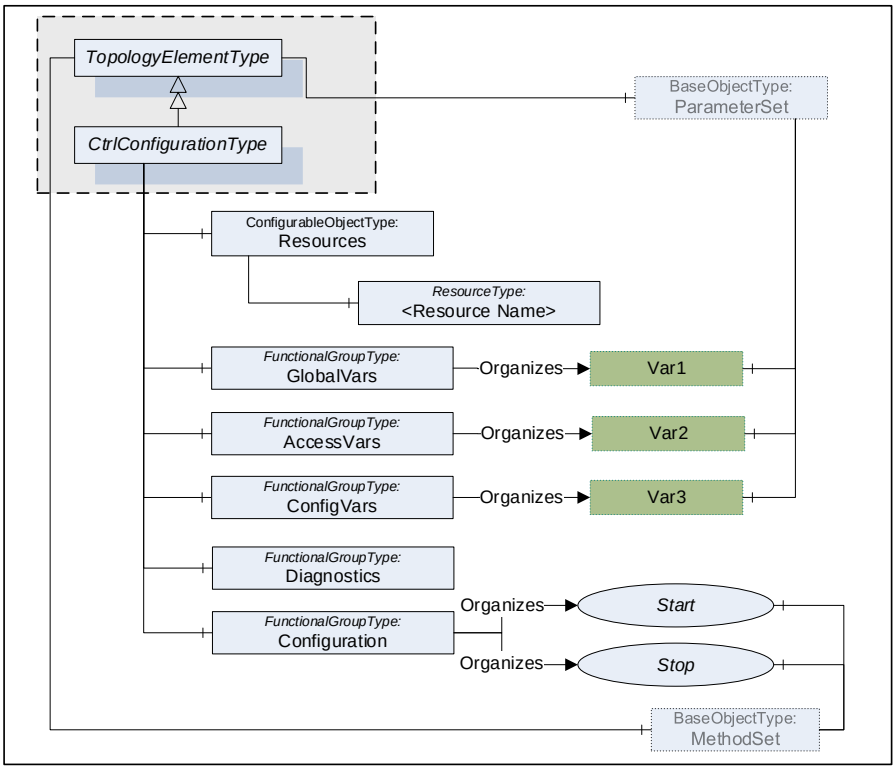

Figure B.2 – PLCopen use for FunctionalGroups

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 organized 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.

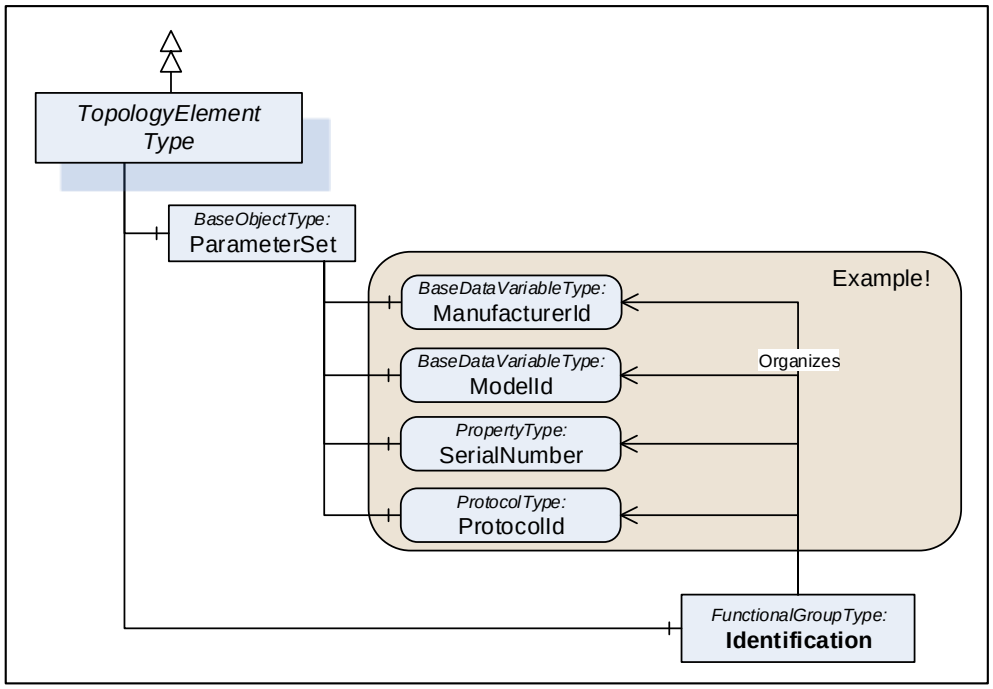

Figure B.3 – Example of an Identification FunctionalGroup

B.3 Software Update examples

B.3.1 Factory Automation Example

This example illustrates the use of software update of several devices from the *Client* point of view.

This is only one example for a specific domain. There will be different *Clients* for different types of systems or industries (e.g. for process domain the process will not be stopped and before a sensor is updated a replacement value needs to be configured in the controller).

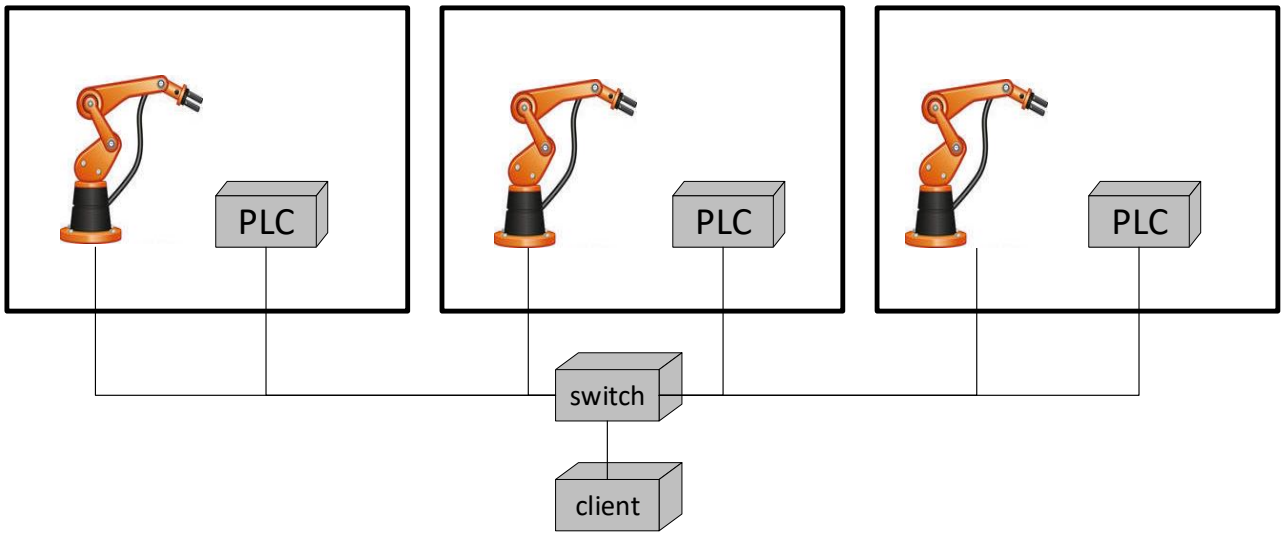

Figure B.4 – Example

The example (illustrated in Figure B.4) describes a production line with several production cells. Each cell contains a robot and a main PLC that can be updated. A switch connects the cells and is also updateable via OPC UA.

A *Client* would perform the following steps:

1. Analyze the system
    - Determine the network topology with all devices.
    - Determine currently installed software and how the devices can perform the update (using *IVendorNameplateType Interface* and *Loading Object*).
    - Determine technical preconditions for the update. E.g. if the device uses *Direct-, Cached- or File System based Loading* (using the *Loading Object*).
2. Prepare installation
    - The user selects the software to be installed
    - Transfer the software and firmware updates to the PLCs, the robots and the switch, except for *Direct-Loading (using CachedLoadingType, FileSystem)*.
3. Schedule installation (*Client* only)
    - Determine how the update can be executed (using *GetUpdateBehavior Methods* of *CachedLoadingType and FileSystemLoadingType*).
    - Wait for strategic condition (e.g., end of shift; no task in queue).
    - Plan the order of update (e.g., robots and PLCs first; infrastructure components last).
4. Prepare devices for installation
    - Stop production line software (using an application specific *Information Model*).

- Bring the robots and PLCs into a state for update (using the *PrepareForUpdate* state machine and/or branch specific state machine).

- Wait for technical starting conditions (e.g., robot in standstill) (using the *PrepareForUpdate* state machine).

5. Execute installation

   - Start the installation of all robots and all PLCs simultaneously (using the *Installation* state machine).

   - Update the switch when robots & PLCs are done (using the *Installation* state machine).

6. Restore device state after installation

   - Restart robots and PLCs (using the *PrepareForUpdate* state machine and/or branch specific state machine).

   - Restart production line software (using an application specific *Information Model*).

B.3.2 Update sequence using Direct-Loading

An example sequence of *Direct-Loading* is shown in Figure B.5.

If the *Server* does not implement the properties *PrepareForUpdate*, *PowerCycle* or *Parameters* of the *SoftwareUpdateType*, the associated options are not supported by the component and *Client-Server* interaction becomes simpler.

In the first steps the device identity and the kind of supported *Server* options of the device must be discovered as described in Figure 34.

How to look up and transfer files for an installation is described in Figure 35.

The preparation can be done as described in Figure 36.

The installation itself is described in Figure 37.

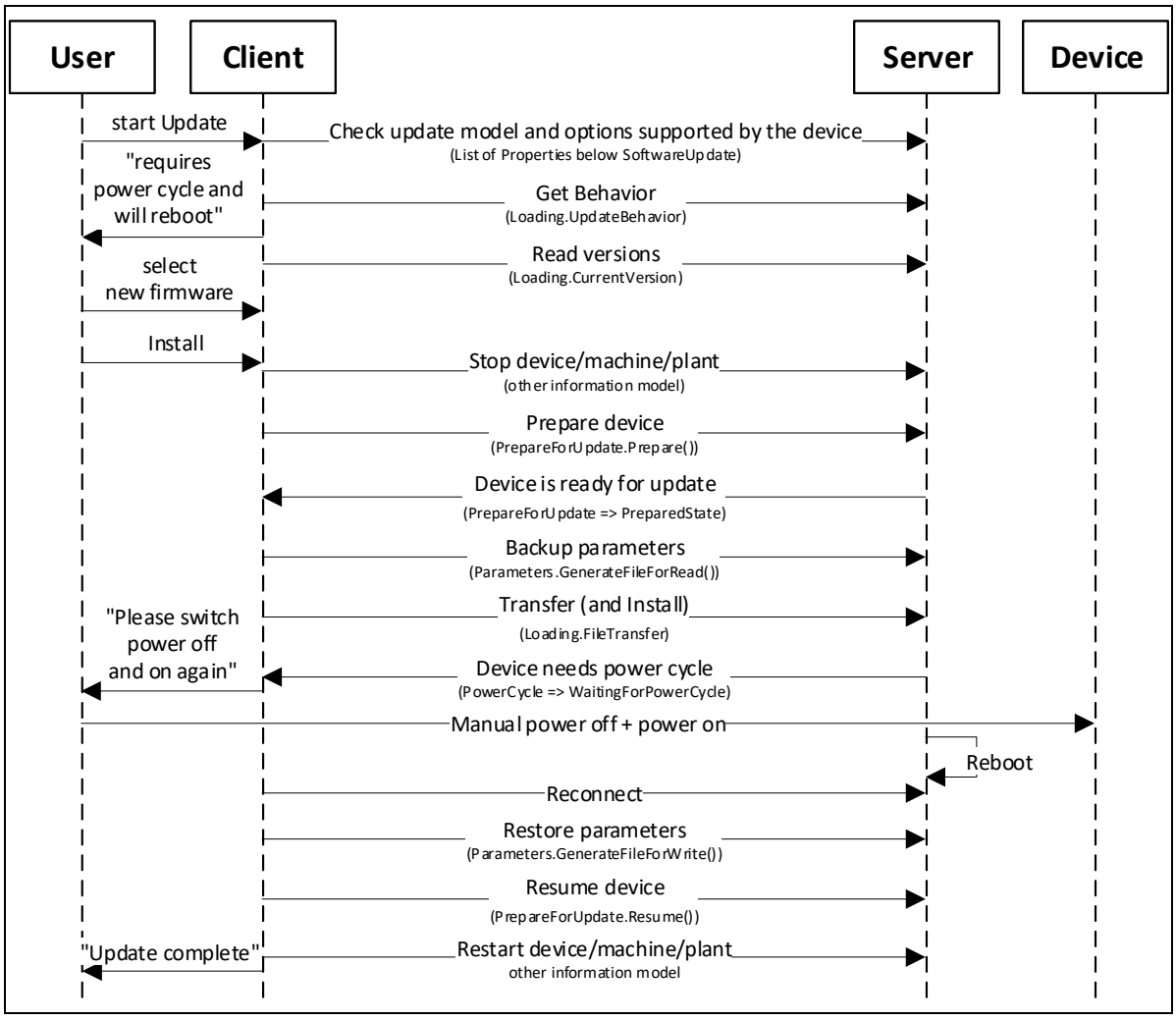

Figure B.5 – Example sequence of Direct-Loading

B.3.3 Update sequence using Cached-Loading

An example sequence of *Cached-Loading* is shown in Figure B.6.

If the *Server* does not implement the properties *PrepareForUpdate*, *PowerCycle* or *Parameters* of the *SoftwareUpdateType*, the associated options are not supported by the component and *Client-Server* interaction becomes simpler.

In the first steps the device identity and the kind of supported *Server* options of the device must be discovered as described in Figure 34.

How to look up and transfer files for an installation is described in Figure 35.

The preparation can be done as described in Figure 36.

The installation itself is described in Figure 38.

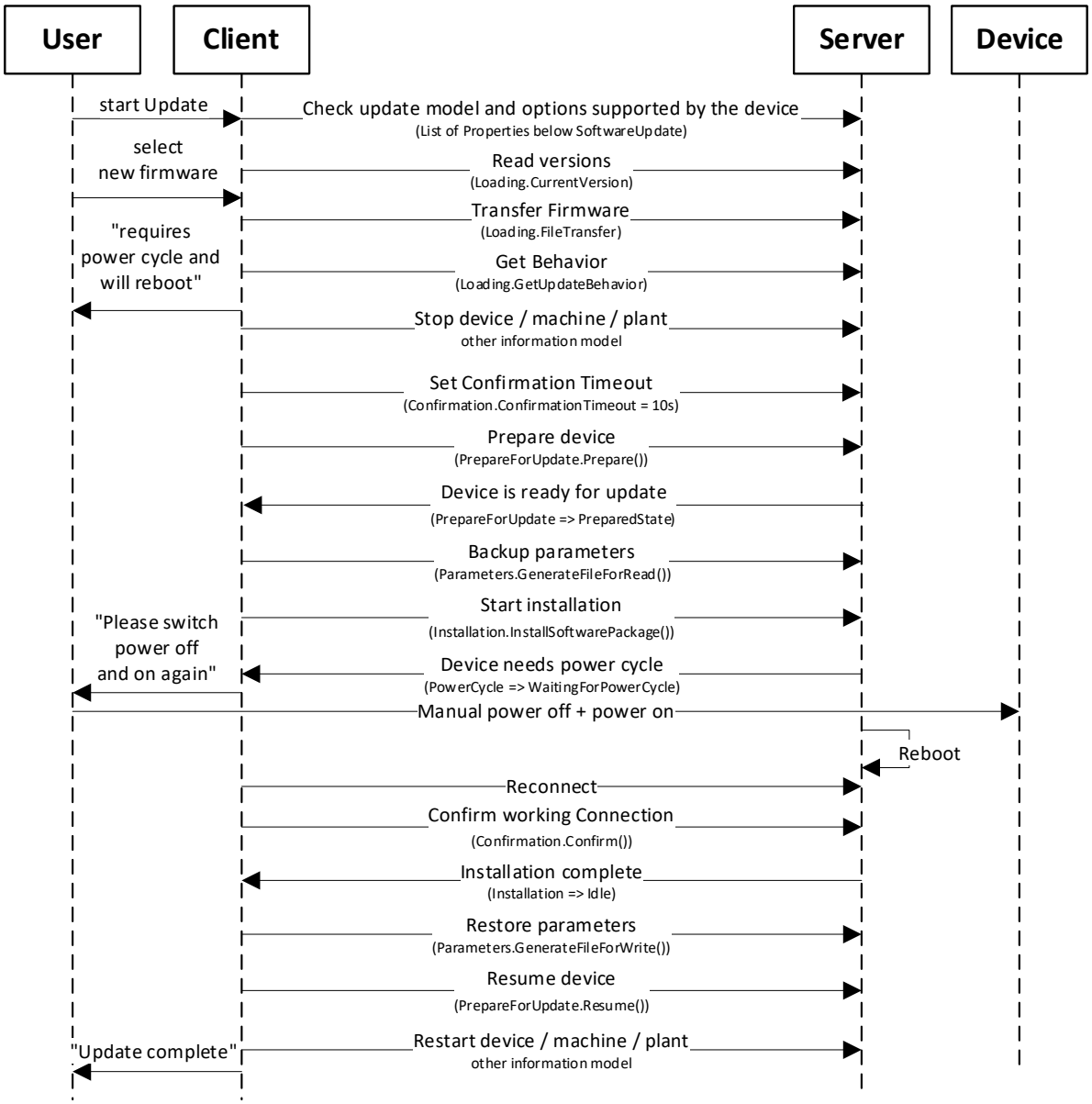

Figure B.6 – Example sequence of Cached-Loading

B.3.4 Update sequence using File System based Loading

An example sequence of *File System based Loading* is shown in Figure B.7.

In this example the server provides the PrepareForUpdate state machine and a preparation for an installation can only be done locally at the device. So the Resume activity described in Figure 38 cannot be commanded by a *Client.*

In the first steps the device identity and the kind of supported *Server* options of the device must be discovered as described in Figure 34.

How to look up and transfer files for an installation is described in Figure 35.

The preparation can be done as described in Figure 36.

The installation itself is described in Figure 38.

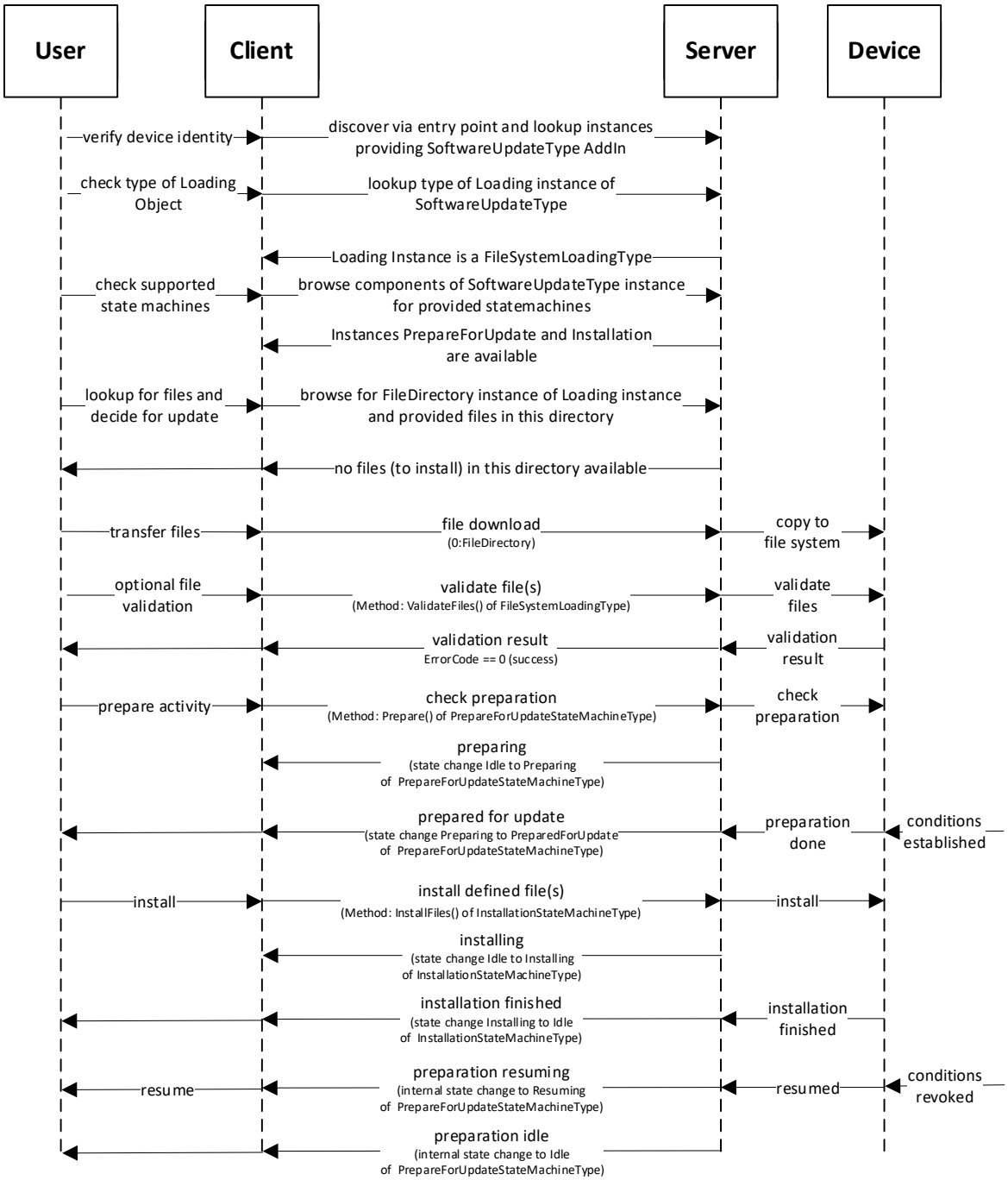

Figure B.7 – Example sequence of File System based Loading

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.

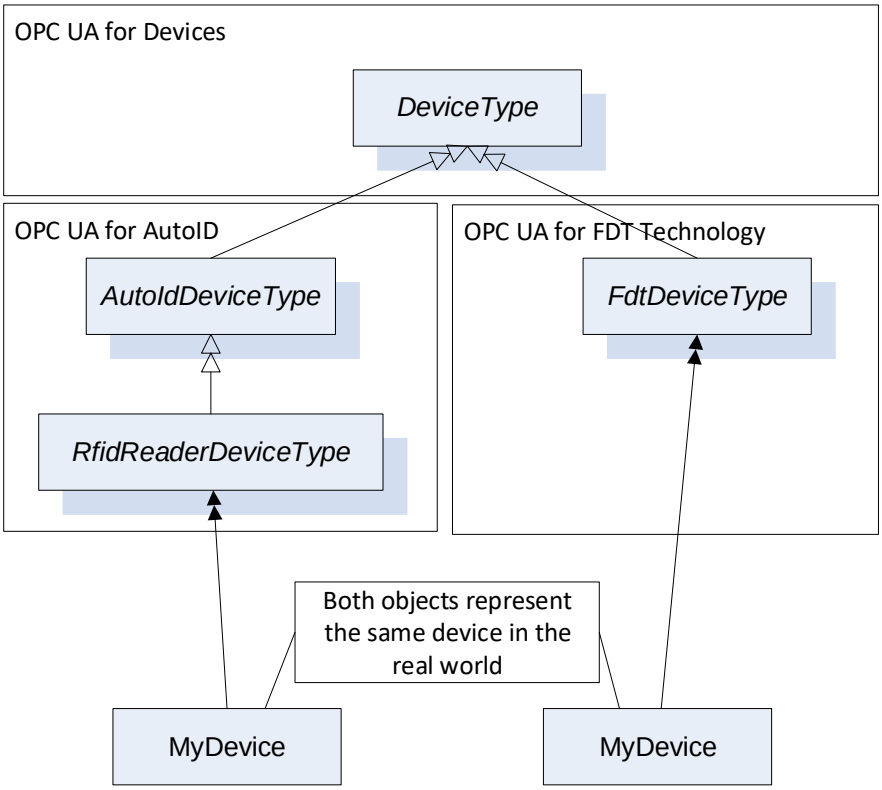

Figure C.1 – Example of applying two companion specifications based on OPC UA for Devices

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.

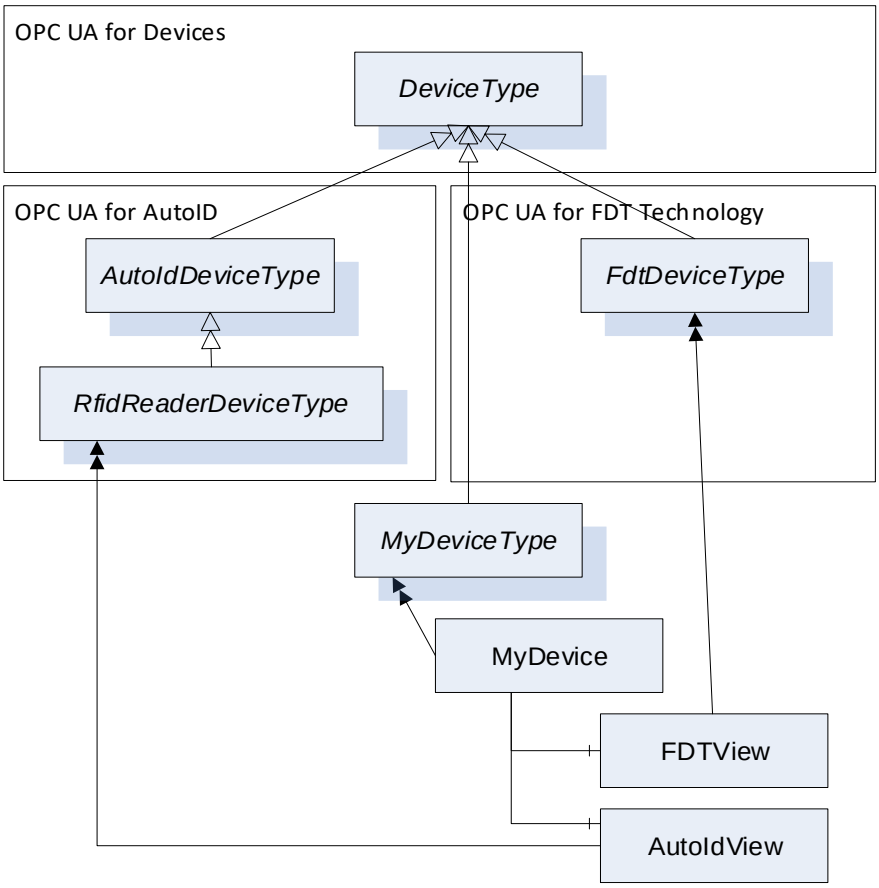

Figure C.2 – Using composition to compose one device representation defined by two companion specifications

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.

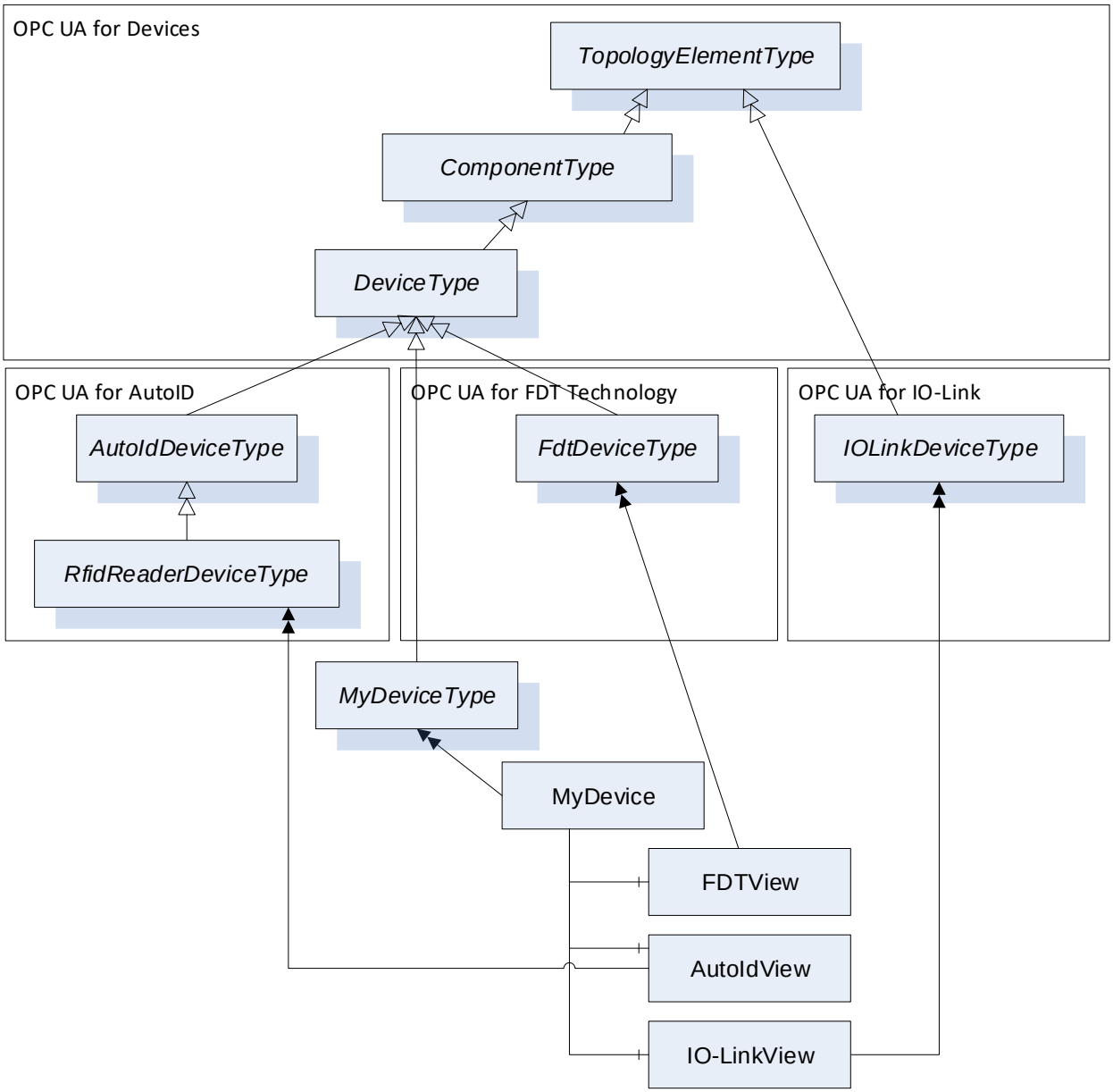

Figure C.3 – Example of applying several companion specifications (I)

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.

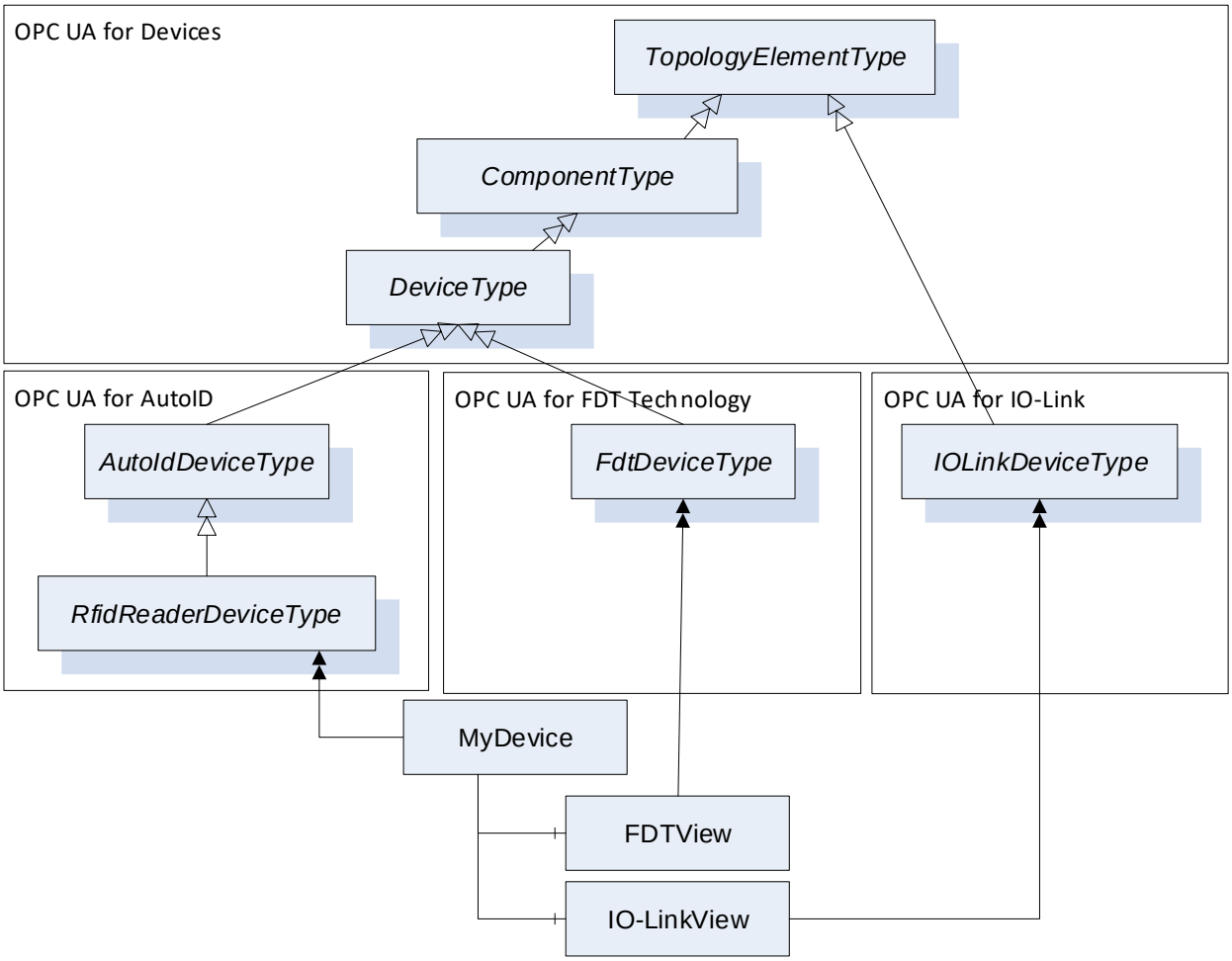

Figure C.4 – Example of applying several companion specifications (II)

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

C.4 Guidelines to manage the same Variables defined in different places

Deploying several Information Models based on this specification on the same device may lead to the situation, that the same *Variable* (e.g. the *Property SerialNumber*) for the same device is used in several places.

When the *Property* is the same, and the value of the *Property* is the same, it is recommended to avoid, that the value is managed in the *Server* in two different places (see Figure C.5, left). One solution is, that the two *Variables* reference the same internal memory managing the value (see Figure C.5, middle). Another solution is, that the *Variable* is only managed once in the *Server*, just referenced from different places (see Figure C.5, right). The solution using the same *Node* is the most optimized one in terms of memory consumption.

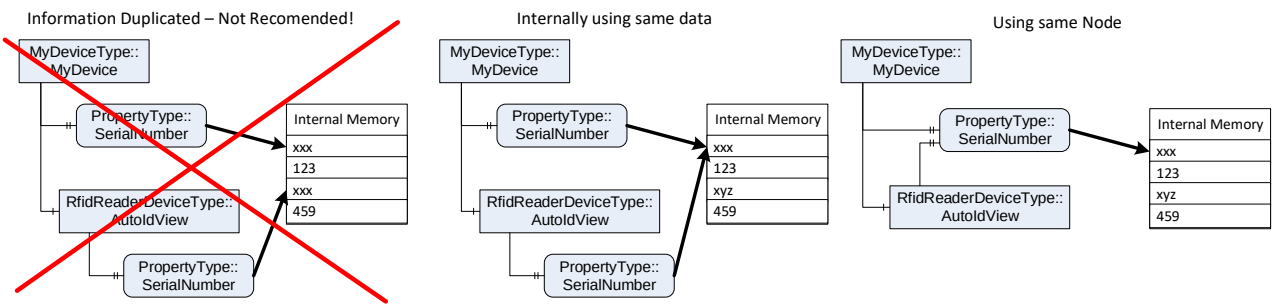

Figure C.5 – Options how to manage the same Variable

C.5 Guidelines on how to use functionality in companion specifications

In the previous sections it was shown how to use this specification when you want to use at least the *TopologyElementType*, providing you the capabilities to manage Parameters and *Methods* via ParameterSet and MethodSet and FunctionalGroups.

If the companion specification only wants to reuse other aspects of this specification, defined in the *Interfaces* in 4.5 or the *AddIns* "Locking" in 7 or Software update in 8, the companion specification does not need to derive from the *ObjectTypes* defined in this specification. Instead of, it can just implement the *Interfaces* or use the *AddIns* in their *ObjectTypes* and build an *ObjectType*-Hierarchy independent of this specification.

In Figure C.5, an example is given. The companion specification defines an *ObjectType* hierarchy, and uses the *AddIns* in the appropriate places (Lock and Transfer). The *Interfaces* can either be implemented by the *ObjectTypes* directly (Figure C.5), or by a sub-component in order to group the functionality (Figure C.7). In the second approach, the RootType does not implement the *IVendorNameplate* directly, but uses a component (Identification) implementing the *Interface*. Here, the *FunctionalGroupType* and the predefined name Identification is used. The B_Type extends the Identification and also implements the *ITagNameplateType*.

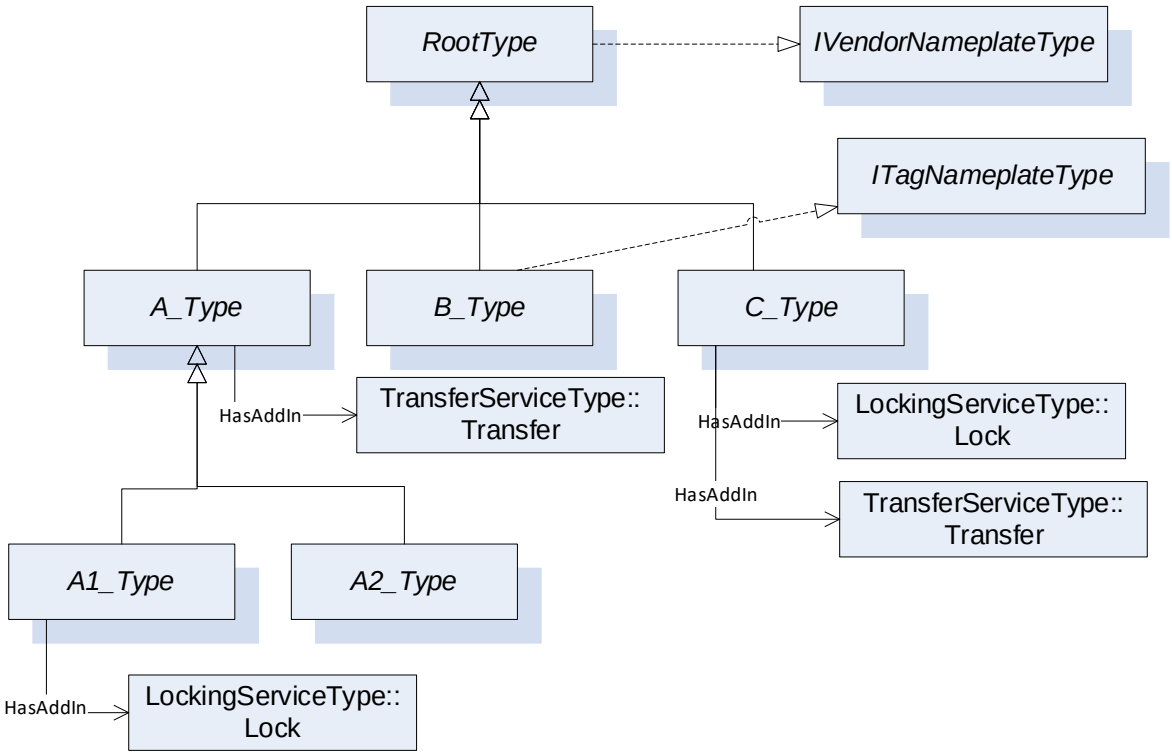

Figure C.6 – Example on how to use AddIns and Interface

The advantage of the first approach is, that the content of the *Interface* is directly at the *ObjectType*, whereas the advantage of the second approach is, that the content of the *Interface* is grouped in the sub-component. When the content of the *Interface* and the additional content of the *ObjectType* and its expected subtypes is rather small, the first approach is recommended. If the content of the *Interface* or the additional content of the *ObjectType* or its subtypes is rather large, the additional grouping *Object* is recommended, as it does not provide a flat list of sub-components, but groups them accordingly and thus makes it easier to use.

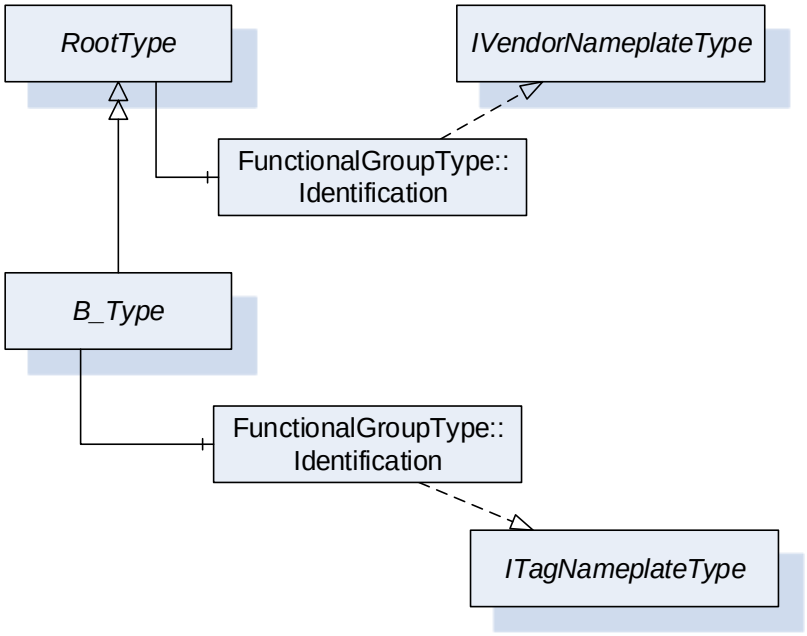

Figure C.7 – Example on how to use Interface with additional Object

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)*

ISO/IEC 11179-6 Information technology - Metadata registries (MDR) - Part 6: Registration