IEC 62541-100

Edition 1.0 2015-03

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 100: Device Interface

Architecture unifiée OPC – Partie 100: Interface d'appareils

IEC 62541-100:2015-03(en-fr)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland

Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

About the IEC
The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications
The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

IEC Catalogue - webstore.iec.ch/catalogue
The stand-alone application for consulting the entire bibliographical information on IEC International Standards, Technical Specifications, Technical Reports and other documents. Available for PC, Mac OS, Android Tablets and iPad.

IEC publications search - www.iec.ch/searchpub
The advanced search enables to find IEC publications by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, replaced and withdrawn publications.

IEC Just Published - webstore.iec.ch/justpublished
Stay up to date on all new IEC publications. Just Published details all new publications released. Available online and also once a month by email.

Electropedia - www.electropedia.org
The world's leading online dictionary of electronic and electrical terms containing more than 30 000 terms and definitions in English and French, with equivalent terms in 15 additional languages. Also known as the International Electrotechnical Vocabulary (IEV) online.

IEC Glossary - std.iec.ch/glossary
More than 60 000 electrotechnical terminology entries in English and French extracted from the Terms and Definitions clause of IEC publications issued since 2002. Some entries have been collected from earlier publications of IEC TC 37, 77, 86 and CISPR.

IEC Customer Service Centre - webstore.iec.ch/csc
If you wish to give us your feedback on this publication or need further assistance, please contact the Customer Service Centre: csc@iec.ch.

A propos de l'IEC
La Commission Electrotechnique Internationale (IEC) est la première organisation mondiale qui élabore et publie des Normes internationales pour tout ce qui a trait à l'électricité, à l'électronique et aux technologies apparentées.

A propos des publications IEC
Le contenu technique des publications IEC est constamment revu. Veuillez vous assurer que vous possédez l'édition la plus récente, un corrigendum ou amendement peut avoir été publié.

Catalogue IEC - webstore.iec.ch/catalogue
Application autonome pour consulter tous les renseignements bibliographiques sur les Normes internationales, Spécifications techniques, Rapports techniques et autres documents de l'IEC. Disponible pour PC, Mac OS, tablettes Android et iPad.

Recherche de publications IEC - www.iec.ch/searchpub
La recherche avancée permet de trouver des publications IEC en utilisant différents critères (numéro de référence, texte, comité d'études,…). Elle donne aussi des informations sur les projets et les publications remplacées ou retirées.

IEC Just Published - webstore.iec.ch/justpublished
Restez informé sur les nouvelles publications IEC. Just Published détaille les nouvelles publications parues. Disponible en ligne et aussi une fois par mois par email.

Electropedia - www.electropedia.org
Le premier dictionnaire en ligne de termes électroniques et électriques. Il contient plus de 30 000 termes et définitions en anglais et en français, ainsi que les termes équivalents dans 15 langues additionnelles. Egalement appelé Vocabulaire Electrotechnique International (IEV) en ligne.

Glossaire IEC - std.iec.ch/glossary
Plus de 60 000 entrées terminologiques électrotechniques, en anglais et en français, extraites des articles Termes et Définitions des publications IEC parues depuis 2002. Plus certaines entrées antérieures extraites des publications des CE 37, 77, 86 et CISPR de l'IEC.

Service Clients - webstore.iec.ch/csc
Si vous désirez nous donner des commentaires sur cette publication ou si vous avez des questions contactez-nous: csc@iec.ch.

IEC 62541-100

Edition 1.0 2015-03

INTERNATIONAL STANDARD

NORME INTERNATIONALE

colour inside

OPC unified architecture – Part 100: Device Interface

Architecture unifiée OPC – Partie 100: Interface d'appareils

INTERNATIONAL ELECTROTECHNICAL COMMISSION

COMMISSION ELECTROTECHNIQUE INTERNATIONALE

ICS 25.040.40; 35.100

ISBN 978-2-8322-2299-7

Warning! Make sure that you obtained this publication from an authorized distributor.
Attention! Veuillez vous assurer que vous avez obtenu cette publication via un distributeur agréé.

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

_____

OPC UNIFIED ARCHITECTURE –

Part 100: Device Interface

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC itself does not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC is not responsible for any services carried out by independent certification bodies.
6) All users should ensure that they have the latest edition of this publication.
7) No liability shall attach to IEC or its directors, employees, servants or agents including individual experts and members of its technical committees and IEC National Committees for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC Publication or any other IEC Publications.
8) Attention is drawn to the Normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.
9) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC 62541-100 has been prepared by subcommittee 65E: Devices and integration in enterprise systems, of IEC technical committee 65: Industrial-process measurement, control and automation.

The text of this standard is based on the following documents:

CDV	Report on voting
65E/372/CDV	65E/412/RVC

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

A list of all parts of the IEC 62541 series, published under the general title , can be found on the IEC website.

The committee has decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IMPORTANT – The 'colour inside' logo on the cover page of this publication indicates that it contains colours which are considered to be useful for the correct understanding of its contents. Users should therefore print this document using a colour printer.

OPC UNIFIED ARCHITECTURE –

Part 100: Device Interface

1 Scope

This part of IEC 62541 is an extension of the overall OPC Unified Architecture standard series and defines the information model associated with *Devices*. This part of IEC 62541 describes three models which build upon each other as follows:

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

2 Reference documents

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

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts*

IEC 62541-3, *OPC Unified Architecture – Part 3: Address Space Model*

IEC 62541-4 *OPC Unified Architecture – Part 4: Services*

IEC 62541-5, *OPC Unified Architecure – Part 5: Information Model*

IEC 62541-6, *OPC Unified Architecure – Part 6: Mappings*

IEC 62541-7, *OPC Unified Architecure – Part 7: Profiles*

IEC 62541-8, *OPC Unified Architecure – Part 8: Data Access*

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

3 Terms, definitions, abbreviations and used data types

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in IEC TR 62541-1, IEC 62541-3, and IEC 62541-8 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 (all parts), *Field Device Integration (FDI)*

) 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 IEC 62541-6.

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.

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

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

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

Note 1 to entry: See IEC 61784.

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

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

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

3.1.10
network
means used to communicate with one specific protocol

3.2 Abbreviations

ADI Analyser Device Integration

CP Communication Processor (hardware module)

CPU Central Processing Unit (of a *Device*)

DA Data Access

DI Device Integration (the short name for this standard)

UA Unified Architecture

UML Unified Modelling Language

XML Extensible Mark-up Language

3.3 Used data types

Table 1 describes the *DataTypes* that are used throughout this document.

Table 1 – DataTypes defined in IEC 62541-3

Parameter Type
Argument
Boolean
Duration
LocalizedText
String
Int32

4 Fundamentals

4.1 OPC UA

The main use case for OPC standards is the online data exchange between devices and HMI or SCADA systems using Data Access functionality. In this use case the device data is provided by an OPC *Server* and is consumed by an OPC *Client* integrated into the HMI or SCADA system. OPC DA provides functionality to browse through hierarchical namespaces containing data items and to read, write and to monitor these items for data changes. The classic OPC standards are based on Microsoft COM/DCOM technology for the communication between software components from different vendors. Therefore classic OPC *Server* and *Clients* are restricted to Windows PC based automation systems.

OPC UA incorporates all features of classic OPC standards like OPC DA, A&E and HDA but defines platform independent communication mechanisms and generic, extensible and object-oriented modelling capabilities for the information a system wants to expose.

The OPC UA network communication part defines different mechanisms optimized for different use cases. The current version of OPC UA is defining an optimized binary protocol for high performance intranet communication as well as Web Services. It allows adding new protocols in the future. Features like security, access control and reliability are directly built into the transport mechanisms. Based on the platform independence of the protocols, OPC UA *Servers* and Clients can be directly integrated into devices and controllers.

The OPC UA *Information Model* provides a standard way for *Servers* to expose *Objects* to *Clients*. *Objects* in OPC UA terms are composed of other *Objects*, *Variables* and *Methods*. OPC UA also allows relationships to other *Objects* to be expressed.

The set of *Objects* and related information that an OPC UA *Server* makes available to *Clients* is referred to as its *AddressSpace*. The elements of the OPC UA *Object* Model are represented in the *AddressSpace* as a set of *Nodes* described by *Attributes* and

interconnected by *References*. OPC UA defines eight classes of *Nodes* to represent *AddressSpace* components. The classes are *Object*, *Variable*, *Method*, *ObjectType*, *VariableType*, *DataType*, *ReferenceType* and *View*. Each *NodeClass* has a defined set of *Attributes*.

This standard makes use of almost all OPC UA *NodeClasses*.

Objects are used to represent real-world entities such as *Devices* and (communication) Networks as well as software entities such as *Blocks*. An *Object* is associated to a corresponding *ObjectType* that provides definitions for that *Object*.

Variables are used to represent values. Two categories of *Variables* are defined, *Properties* and *DataVariables*.

Properties are *Server*-defined characteristics of *Objects*, *DataVariables* and other *Nodes*. *Properties* are not allowed to have *Properties* defined for them. Examples for *Properties* of *Objects* are the device serial number and the block tag.

DataVariables represent the contents of an *Object*. *DataVariables* may have component *DataVariables*. This is typically used by *Servers* to expose individual elements of arrays and structures. This standard uses *DataVariables* to represent the *Parameters* of both *Blocks* and *Devices*.

4.2 Conventions used in this document

4.2.1 Conventions for Node descriptions

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

Table 2 – Type definition table

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

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 IEC 62541-3). 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*}". The *ValueRank* is set to the corresponding value (see IEC 62541-3) and the *ArrayDimensions* is set to null or is omitted. In Table 3 examples are given.

Table 3 – Examples of DataTypes

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

- The TypeDefinition is specified for *Objects* and *Variables*.
- The TypeDefinition column specifies a *NodeId* of a *TypeDefinitionNode*, i.e. the specified *Node* points with a *HasTypeDefinition Reference* to the corresponding *TypeDefinitionNode*. The symbolic name of the *NodeId* is used in the table.
- 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.

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

Components of *Nodes* can be complex, i.e. contain components by themselves. The *TypeDefinition*, *NodeClass*, *DataType* and *ModellingRule* can be derived from the type definitions, and the symbolic name can be created as defined in 4.2.2.1. Therefore those containing components are not explicitly specified; they are implicitly specified by the type definitions.

4.2.2 NodeIds and BrowseNames

4.2.2.1 NodeIds

The *NodeIds* of all *Nodes* described in this document 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 namespace for this standard is defined in Annex A. The *NamespaceIndex* for all *NodeIds* and *BrowseNames* defined in this standard is server specific and depends on the position of the namespace URI in the server namespace table.

4.2.2.2 BrowseNames

The text part of the *BrowseNames* for all *Nodes* defined in this standard is specified in the tables defining the *Nodes*. The *NamespaceIndex* for all *BrowseNames* defined in this standard is server specific and depends on the position of the namespace URI in the server namespace table.

5 Device model

5.1 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 so as to give a rough idea of the overall structure.

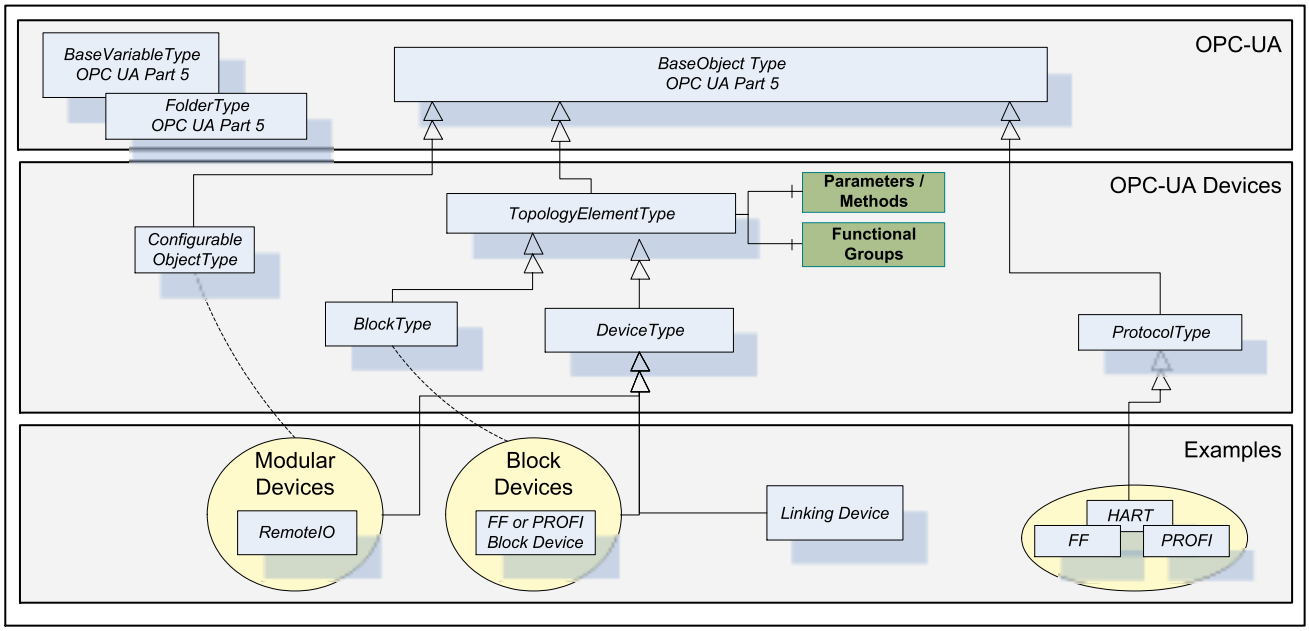


Figure 1 – Device model overview

The boxes in this drawing show the *ObjectTypes* used in this standard as well as some elements from other standards that help understand some modelling decisions. The upper grey box shows the OPC UA core *ObjectTypes* from which the *TopologyElementType* and *ProtocolType* are derived. The grey box in the second level shows the main *ObjectTypes* that this standard 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. It introduces *Parameters* and *Methods*. This standard also defines a functional grouping concept to provide alternative viewpoints.

The *DeviceType ObjectType* provides a general type definition for any *Device*. *Devices* – in addition to *Parameters* and *Methods* – may support subdevices and may support *Blocks*. *Blocks* are typically used as means to organise the functionality within a *Device*. Specific types of *Blocks* will be specified by the various field communication foundations.

A *ProtocolType ObjectType* represents a specific communication/*FieldBus* protocol implemented by a certain *TopologyElement*. Examples are *FFBusType*, *HARTBusType*, or PROFIBUS/PROFINETType.

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

5.2 TopologyElementType

This *ObjectType* defines the basic information components for all configurable elements in a device topology. It introduces *ParameterSet*, *MethodSet*, and *FunctionalGroups*. Figure 2 shows the *TopologyElementType*. It is formally defined in Table 4.

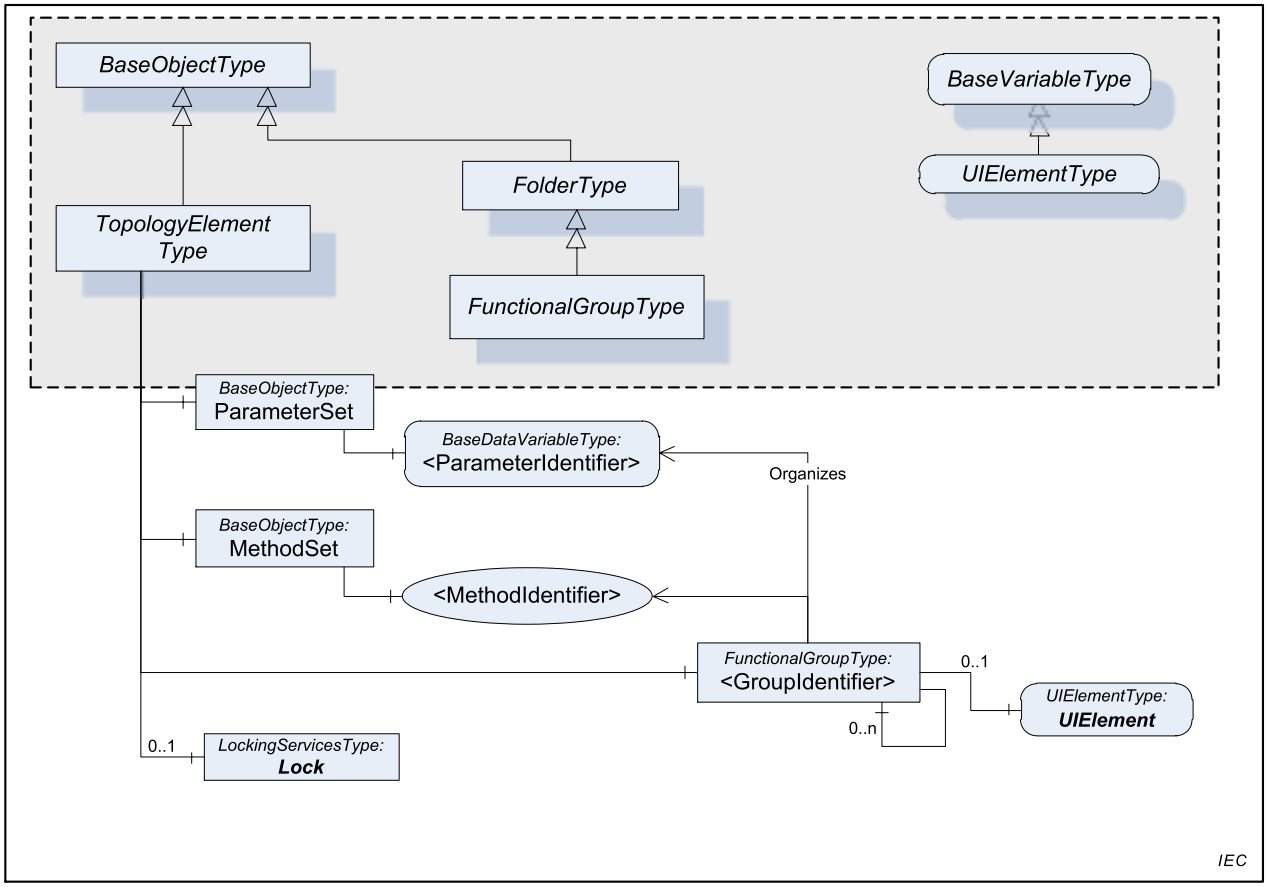


Figure 2 – Components of the TopologyElementType

All elements in a topology may have *Parameters* and *Methods*. If such an element has *Parameters* they are kept in an *Object* called "*ParameterSet*" as a flat list of *Parameters*. If it has *Methods* they are kept the same way in an *Object* called "*MethodSet*". *FunctionalGroups* can be used to organise the *Parameters* and *Methods* to reflect the structure of the *TopologyElement*. The same *Parameter* or *Method* might be referenced from more than one *FunctionalGroup*. Each group may reference an optional user interface for the represented functionality. *FunctionalGroups* are specified in 5.3.

- Rule for all *Methods* organised in *FunctionalGroups*:

  Since *Methods* are components of the *MethodSet Object*, *Clients* have to specify the *NodeId* of the *MethodSet* instance in the *ObjectId* argument of the Call *Service*.

Parameters are modelled with OPC UA *DataVariable* nodes. A *Parameter* can be an instance of *BaseDataVariableType* (defined in IEC 62541-5) or any sub-type defined by OPC UA or other standard organisations. Some of the most common *VariableTypes* (*DataItemType, AnalogItemType,* and *DiscreteItemTypes*) are defined in IEC 62541-8. In this standard, the term *Parameter* is used generically – regardless of *VariableType*.

The *TopologyElement ObjectType* is formally defined in Table 4

Table 4 – TopologyElementType definition

Attribute	**Value**				
BrowseName	TopologyElementType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in IEC 62541-5					
HasSubtype	ObjectType	DeviceType	Defined in 5.3		
HasSubtype	ObjectType	BlockType	Defined in 5.10		
HasComponent	Object	ParameterSet		BaseObjectType	Optional
HasComponent	Object	MethodSet		BaseObjectType	Optional
HasComponent	Object	<GroupIdentifier>		FunctionalGroupType	OptionalPlaceHolder
HasComponent	Object	Identification		FunctionalGroupType	Optional
HasComponent	Object	Lock		LockingServicesType	Optional

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

ParameterSet gathers the references to all *Parameters* that are exposed to the *Client*. The *ParameterSet* is mandatory if *Parameters* exist for a *TopologyElement*. The "*ParameterSet*" *Object* is formally defined in Table 5.

Table 5 – ParameterSet definition

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

MethodSet gathers the references to all *Methods* that are exposed to the *Client*. The "*MethodSet*" *Object* is formally defined in Table 6.

Table 6 – MethodSet definition

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

TopologyElements may have an arbitrary number of *FunctionalGroups* to organise *Parameters* and *Methods* (see 5.3). A special *FunctionalGroup* called **Identification** shall be used to organise *Parameter*s for identification of this *TopologyElement* (see 5.4).

In addition to the elements described in 5.2, TopologyElements may also support *LockingServices* (defined in 8.3.3).

5.3 FunctionalGroupType

This sub-type of the OPC UA *FolderType* is used to organise the *Parameters* and *Methods* from the complete set (*ParameterSet*, *MethodSet*) with regard to their application (e.g. maintenance, diagnosis). It can also reference a *Property* of the *TopologyElement* that it belongs to. It may contain a user interface element. The same *Property*, *Parameter* or *Method* can be referenced from more than one *FunctionalGroup*.

FunctionalGroups can be nested.

Figure 3 shows the *FunctionalGroupType* components. It is formally defined in Table 7.

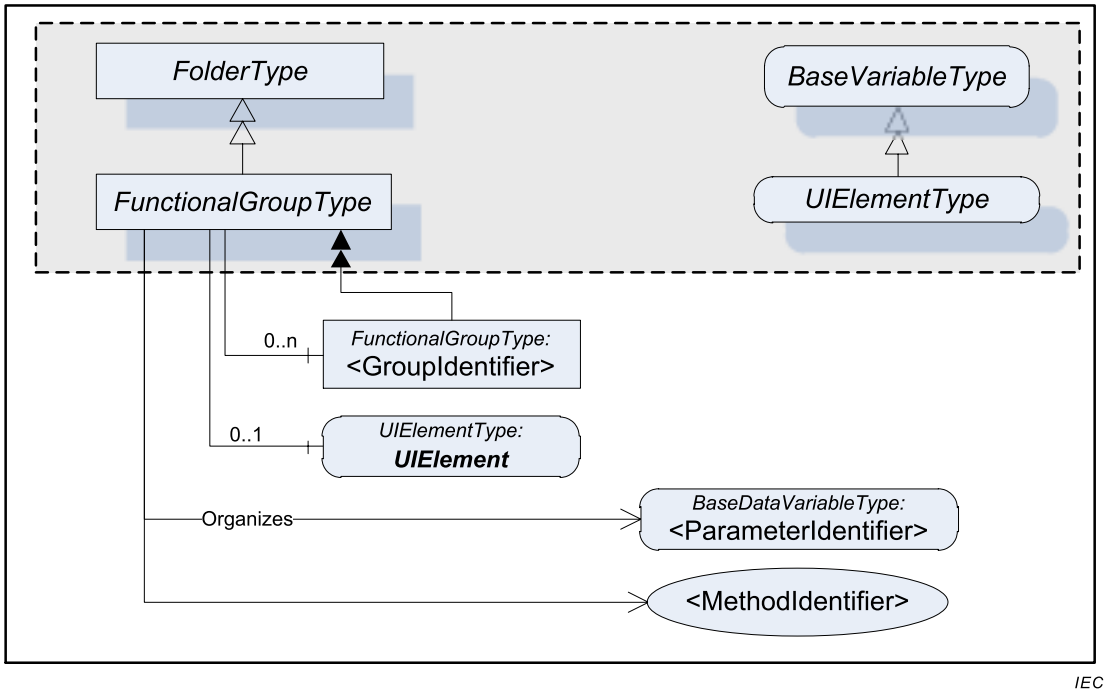


Figure 3 – FunctionalGroupType

Table 7 – FunctionalGroupType definition

Attribute	**Value**				
BrowseName	FunctionalGroupType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *FolderType* defined in IEC 62541-5					
HasComponent	Object	<GroupIdentifier>		FunctionalGroupType	OptionalPlaceHolder
Organizes	Variable	<ParameterIdentifier>	BaseDataType	BaseDataVariableType	OptionalPlaceHolder
Organizes	Method	<MethodIdentifier>			OptionalPlaceHolder
HasComponent	Variable	UIElement	BaseDataType	UIElementType	Optional

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”.

The *Description Attribute* is used to describe the intended purpose of the *FunctionalGroup*.

UIElement is the user interface element for this *FunctionalGroup*. See 5.5 for the definition of *UIElements*.

The examples in Figure 4 and Figure 5 illustrate the use of *FunctionalGroups:*

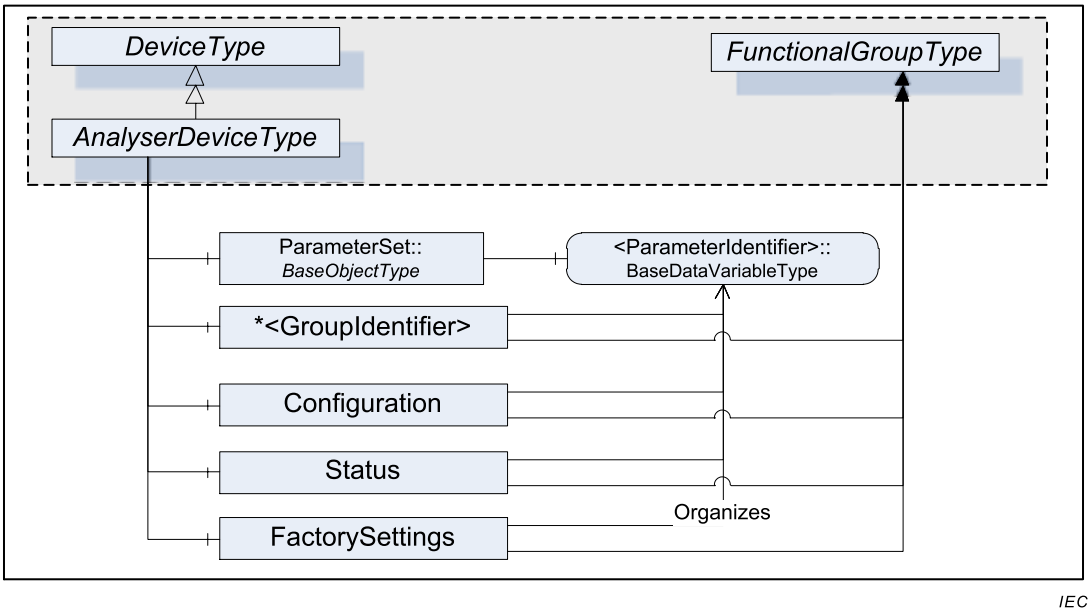


Figure 4 – Analyser Device use for FunctionalGroups (UA Companion ADI)

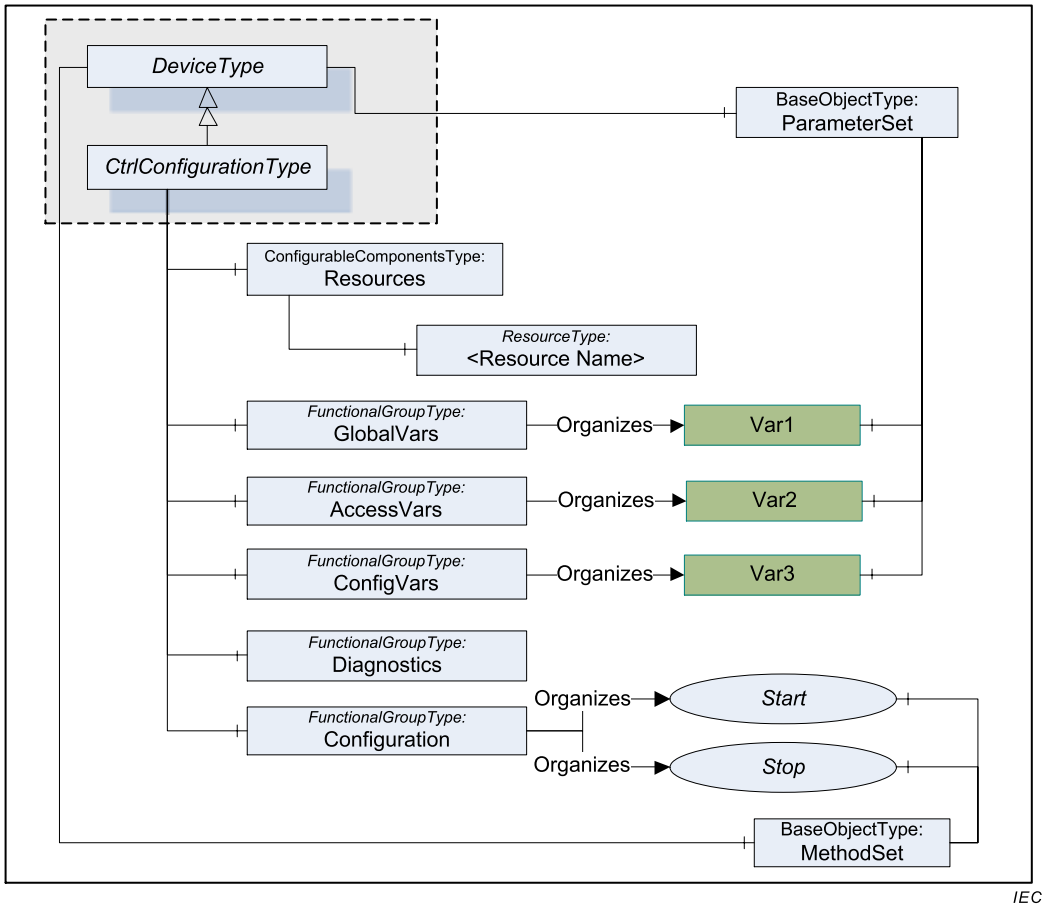


Figure 5 – PLCopen use for FunctionalGroups (UA Companion PLCopen)

5.4 Identification FunctionalGroup

*Parameter*s for identification of a *TopologyElement* shall be organised in a *FunctionalGroup* called **Identification**. As an example *Clients* can use the values of these *Parameter*s to find certain elements or detect mismatches between configuration data and the currently connected element. Figure 6 illustrates the **Identification** *FunctionalGroup* with an example.

NOTE Standardization bodies are expected to define the Identification contents for specific bus protocols.

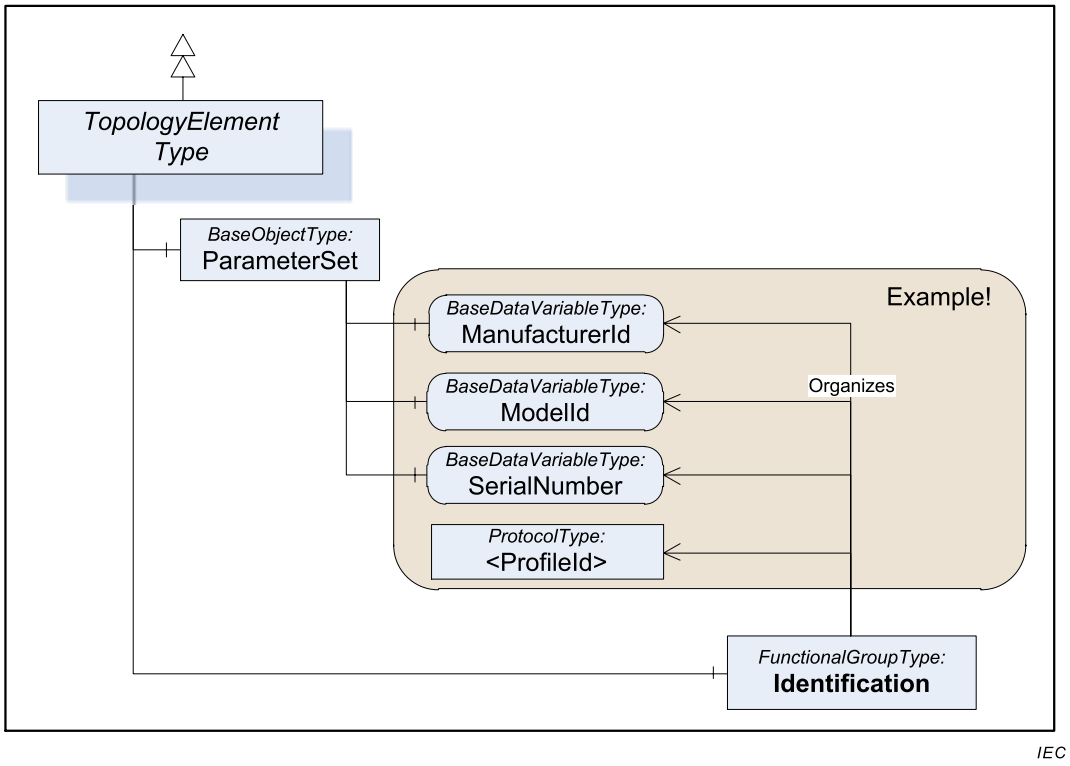


Figure 6 – Example of an Identification FunctionalGroup

5.5 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 sub-types can be used for instances. No concrete *UIElements* are defined in this standard. An FDI (Field Device Integration, see IEC 62769) specifies two concrete sub-types

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

The *UIElementType* is specified in Table 8.

Table 8 – UIElementType definition

Attribute	**Value**				
BrowseName	UIElementType				
IsAbstract	True				
DataType	BaseDataType				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of BaseVariableType defined in IEC 62541-5					

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

5.6 DeviceType

This *ObjectType* defines the structure of a *Device*. Figure 7 shows the *DeviceType*. It is formally defined in Table 9.

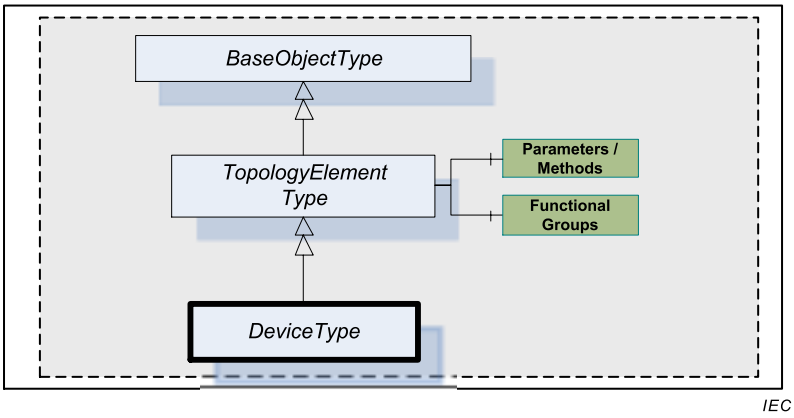


Figure 7 – DeviceType

Table 9 – DeviceType definition

Attribute	**Value**				
BrowseName	DeviceType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the *Properties* of the *TopologyElementType* defined in 5.2					
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
HasComponent	Variable	DeviceHealth	DeviceHealth	BaseDataVariableType	Optional
HasComponent	Object	DeviceTypeImage		FolderType	Optional
HasComponent	Object	Documentation		FolderType	Optional
HasComponent	Object	ProtocolSupport		FolderType	Optional
HasComponent	Object	ImageSet		FolderType	Optional
HasComponent	Object	<CPIdentifier>		ConnectionPointType	OptionalPlaceHolder

The *DeviceType ObjectType* is abstract. There will be no instances of a *DeviceType* itself, but there will be instances of sub-types of this type. In this standard, the term *Device* generically refers to an instance of any *ObjectType* derived from the *DeviceType*.

Devices may have *Parameters*, *Methods*, and *FunctionalGroups* as defined for the *TopologyElementType*.

The following *Properties* provide a way for a *Client* to get common *Device* information. This is not necessarily a replacement for this information appearing in the *ParameterSet* and/or *FunctionalGroups* of the *Device*. Note that this standard does not make other than the following assumptions concerning the semantic. Standard organisations may further specify the contents.

Although most *Properties* are mandatory for all *DeviceType* instances, some of them 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**

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

The *RevisionCounter Property* is an incremental counter indicating the number of times the static data within the *Device* has been modified.

The *Manufacturer Property* provides the name of the company that manufactured the *Device*.

The *Model Property* provides the model name of the *Device*.

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

The *DeviceRevision Property* provides the overall revision level of the *Device*.

The *SoftwareRevision Property* provides the revision level of the software/firmware of the *Device*.

The *HardwareRevision Property* provides the revision level of the hardware of the *Device*.

The *DeviceClass Property* indicates in which domain or for what purpose a certain *Device* is used. Examples are “ProgrammableController”, “RemoteIO”, and “TemperatureSensor”. This standard does not predefine any *DeviceClass* names. More specific standards that utilize the *DeviceType* will likely introduce such classifications (e.g. IEC 62769, UA Companion PLCopen, or UA Companion ADI).

The *Description* attribute of the *Device Object* provides information that serves to further identify, manage, locate, and/or explain the device whose contents are defined by the user.

Other organisations may specify additional semantics for the contents of these *Properties*.

Parameters like *ManufacturerId* (numeric identifier of the company), *ModelId* and *SubModelId* (numeric identifiers of the model) are not provided as *Properties*. Instead, these *Parameters* shall be included into the **Identification** *FunctionalGroup* defined in 5.4.

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

The *DeviceHealth DataType* is an enumeration that defines the device condition. Its values are defined in Table 10.

Table 10 – DeviceHealth values

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

ConnectionPoints (see 6.3) represent the interface (interface card) of a *Device* to a *Network*. A *Device* can have multiple *ConnectionPoints* if it supports multiple protocols and/or multiple *Communication Profiles*.

5.7 Device support information

5.7.1 General

Each *DeviceType* may have a set of additional data. These are mainly images, documents, or protocol-specific data. The various types of information are organised into different folders. Each information element is represented by a read-only *Variable*. The information can be retrieved by reading the *Variable* value. The *Variables* are owned by the *DeviceType*, i.e., all *Device* instances will typically share the same *Variable Node* and when read will return the same value.

Figure 8 illustrates how the additional data is integrated into a *DeviceType*.

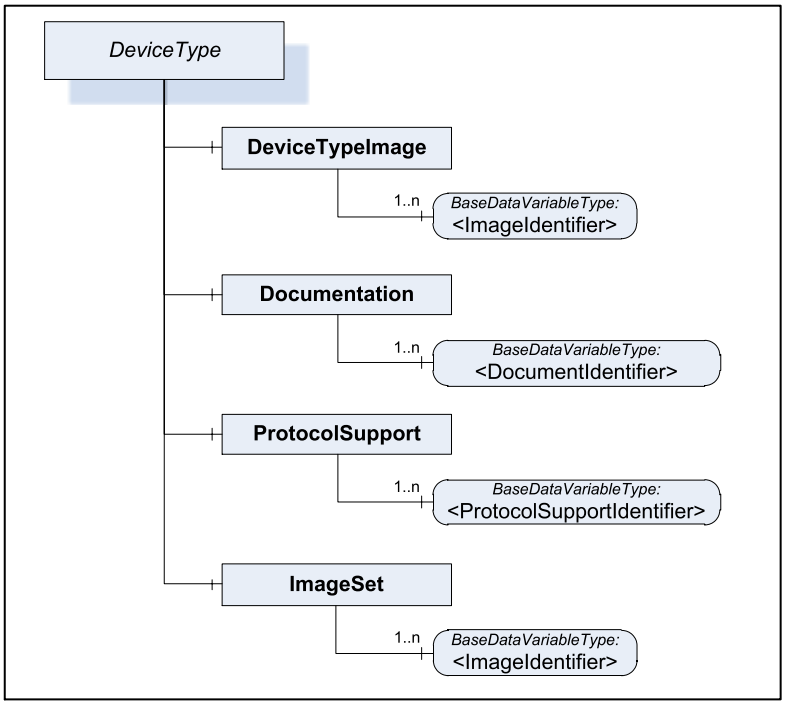


Figure 8 – Integration of support information within a DeviceType

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 IEC 62541-4) to read these *Variables* in chunks, for example, one-megabyte chunks. It is up to the *Client* whether it starts without an index and repeats with an IndexRange only after an error or whether it always uses an IndexRange.

The different types of support information are specified in 5.7.2 to 5.7.5.

5.7.2 Device Type Image

Pictures of a *Device* can be exposed as *Variables* organised in the *DeviceTypeImage* folder. There may be multiple images of different resolutions. Each image is a separate *Variable*. The "*DeviceTypeImage*" *Folder* is formally defined in Table 11.

Table 11 – DeviceTypeImage definition

Attribute	**Value**				
BrowseName	DeviceTypeImage				
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**DataType**	**ModellingRule**
HasTypeDefinition	ObjectType	FolderType (defined in IEC 62541-5)			
HasComponent	Variable	<ImageIdentifier>	BaseDataVariableType	Image	MandatoryPlaceholder

All images are transferred as a *ByteString*. The *DataType* of the *Variable* specifies the image format. OPC UA defines BMP, GIF, JPG and PNG (see IEC 62541-3).

5.7.3 Documentation

Documents provided for a *Device* are exposed as *Variables* organised in the *Documentation* folder. In most cases they will represent a product manual, which can exist as a set of individual documents. The "*Documentation*" *Folder* is formally defined in Table 12.

Table 12 – Documentation definition

Attribute	**Value**				
BrowseName	Documentation				
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**DataType**	**ModellingRule**
HasTypeDefinition	ObjectType	FolderType (defined in IEC 62541-5)			
HasComponent	Variable	<DocumentIdentifier>	BaseDataVariableType	ByteString	MandatoryPlaceholder

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

5.7.4 Protocol support files

Protocol support files provided for a *Device* are exposed as *Variables* organised in the *ProtocolSupport* folder. They may represent various types of information as defined by a protocol. Examples are a GSD or a CFF file. The "*ProtocolSupport*" *Folder* is formally defined in Table 13.

Table 13 – ProtocolSupport definition

Attribute	**Value**				
BrowseName	ProtocolSupport				
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**DataType**	**ModellingRule**
HasTypeDefinition	ObjectType	FolderType (defined in IEC 62541-5)			
HasComponent	Variable	<ProtocolSupportIdentifier>	BaseDataVariableType	ByteString	MandatoryPlaceholder

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

5.7.5 Images

Images that are used within *UIElements* are exposed as separate *Variables* rather than embedding them in the element. All image *Variables* will be aggregated by the *ImageSet* folder. The *UIElement* shall specify an image by its name that is also the *BrowseName* of the image *Variable*. *Clients* can cache images so they do not have to be transferred more than once. The "*ImageSet*" *Folder* is formally defined in Table 14.

Table 14 – ImageSet definition

Attribute	**Value**				
BrowseName	ImageSet				
References	**NodeClass**	**BrowseName**	**TypeDefinition**	**DataType**	**ModellingRule**
HasTypeDefinition	ObjectType	FolderType (defined in IEC 62541-5)			
HasComponent	Variable	<ImageIdentifier>	BaseDataVariableType	Image	MandatoryPlaceholder

The *DataType* of the *Variable* specifies the image format. OPC UA defines BMP, GIF, JPG and PNG (see IEC 62541-3).

5.8 DeviceSet entry point

To promote the interoperability of *Client*s and *Server*s, all instantiated *Devices* shall be aggregated in an *Object* called "**DeviceSet**". For complex *Devices* that are composed of various components that are also *Devices*, only the top-level *Device* shall be referenced from the *DeviceSet Object*. The component *Devices* shall be locatable by following the *HasComponent References* from the top-level *Device*. An example is the *Modular Device* defined in 9.3 and also illustrated in Figure 9).

Figure 9 shows the *AddressSpace* organisation with this standard entry point.

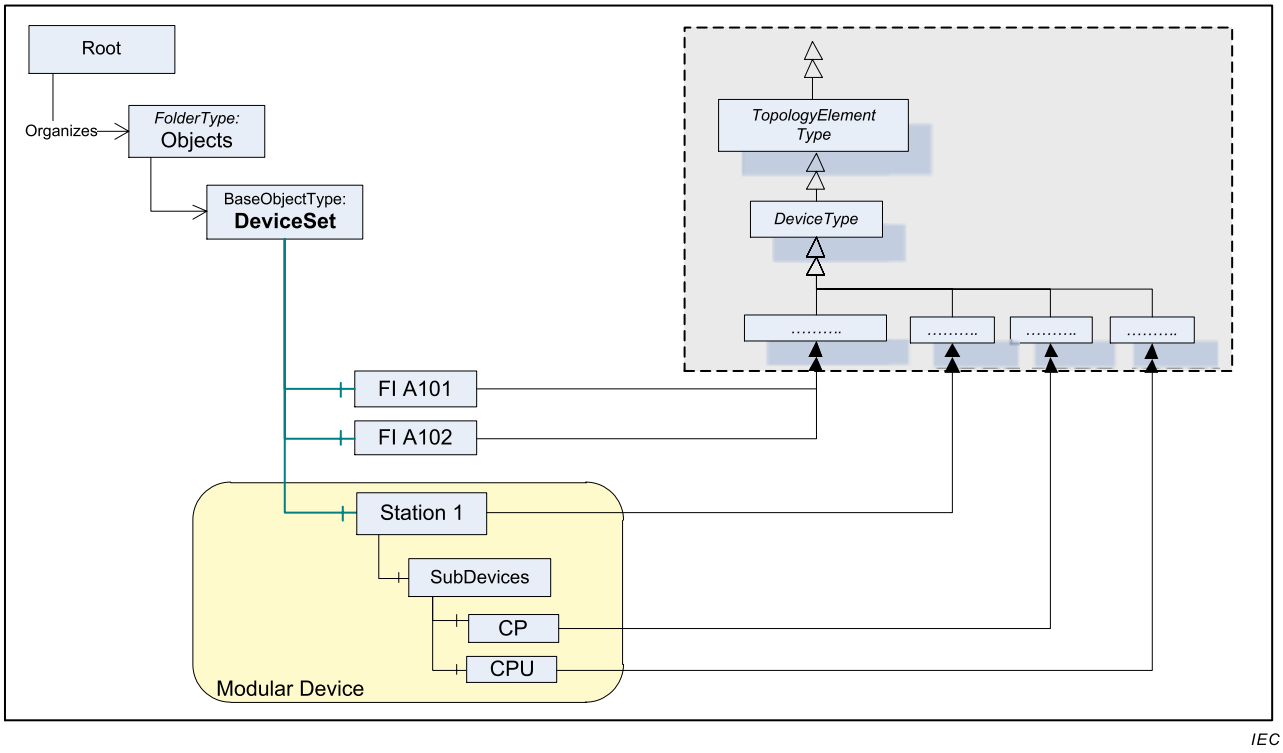


Figure 9 – Standard entry point for Devices

The **DeviceSet** node is formally defined in Table 15.

Table 15 – DeviceSet definition

Attribute	**Value**		
BrowseName	DeviceSet		
References	**NodeClass**	**BrowseName**	**TypeDefinition**
OrganizedBy by the Objects Folder defined in IEC 62541-5			
HasTypeDefinition	ObjectType	BaseObjectType	

5.9 ProtocolType

The *ProtocolType ObjectType* and its sub-types are used to specify a specific communication/*FieldBus* protocol that is supported by a *Device* or *Network*. The *BrowseName* of each instance of a *ProtocolType* shall define the *Communication Profile* (see Figure 10).

Figure 10 shows the *ProtocolType* including some specific types and instances that represent *Communication Profiles* of that type. It is formally defined Table 16.

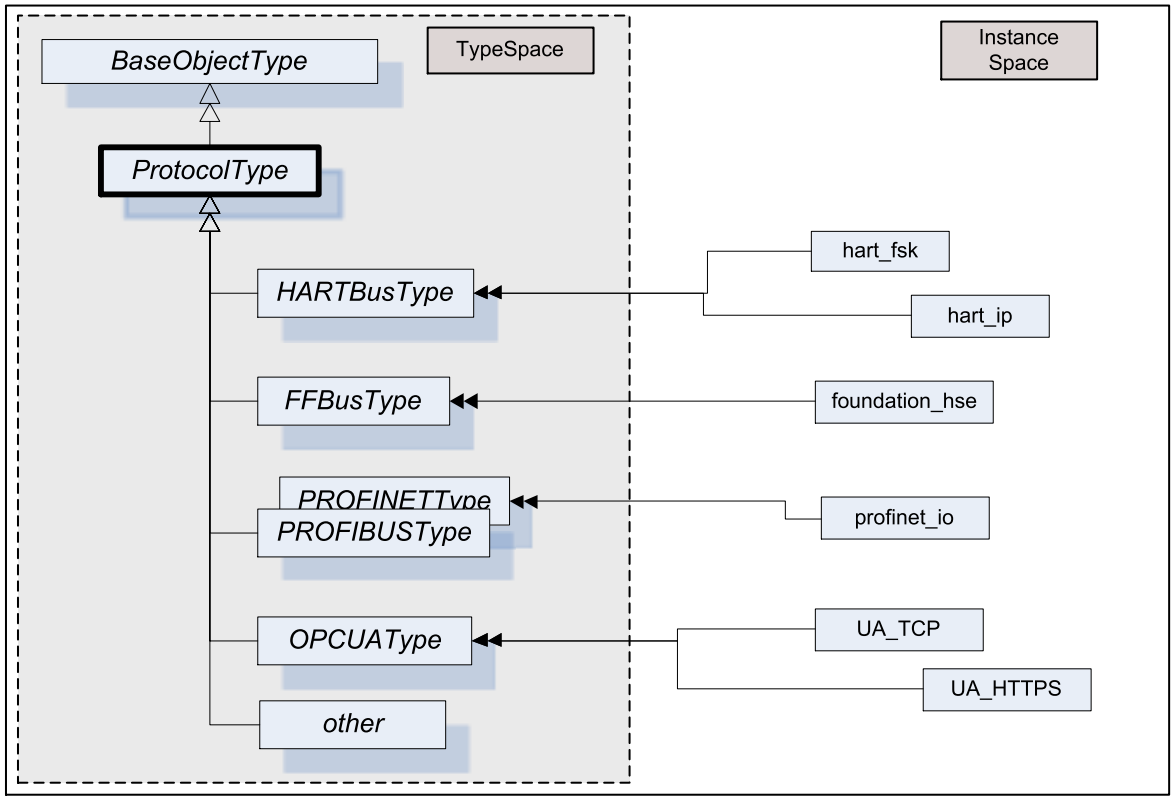


Figure 10 – Example of a ProtocolType hierarchy with instances that represent specific communication profiles

Table 16 – ProtocolType definition

Attribute	**Value**				
BrowseName	ProtocolType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in IEC 62541-5					

5.10 BlockType

This *ObjectType* defines the structure of a *Block Object*. Figure 11 depicts the *BlockType* hierarchy. It is formally defined in Table 17.

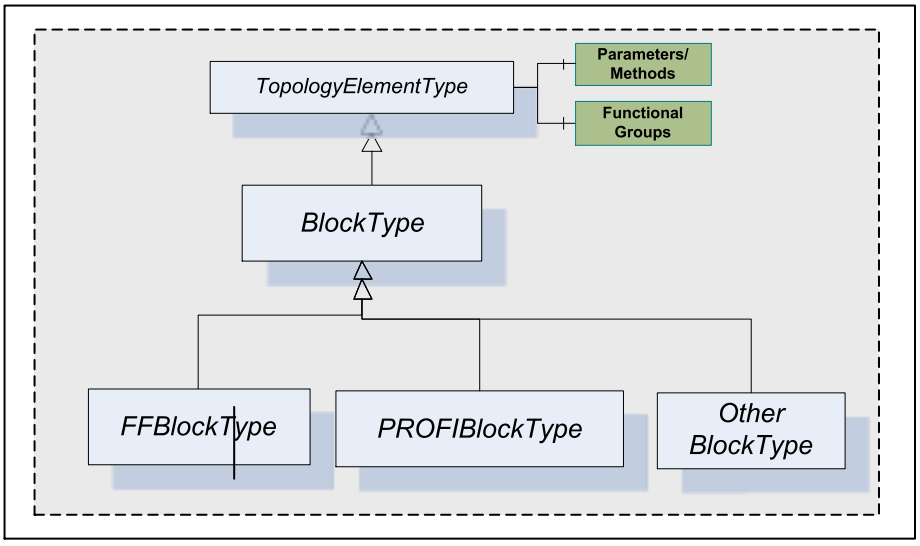


Figure 11 – BlockType hierarchy

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

Table 17 – BlockType definition

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

BlockType is a sub-type of *TopologyElement* 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 sub-types of this *Type*. In this standard, the term *Block* generically refers to an instance of any sub-type of the *BlockType*.

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

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

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

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

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

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

5.11 Configurable components

5.11.1 General pattern

This subclause 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 12 illustrates these principles.

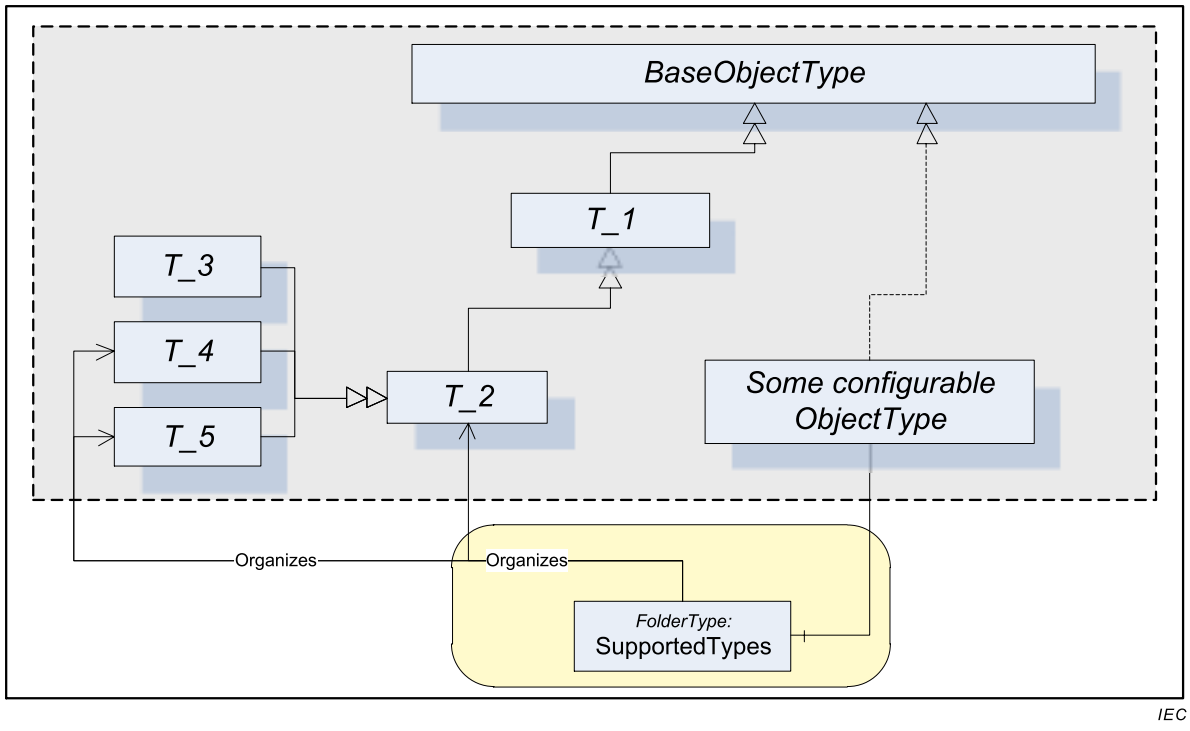


Figure 12 – Configurable component pattern

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

5.11.2 ConfigurableObjectType

This *ObjectType* implements the configurable component pattern and is used when an *Object* or an instance declaration needs nothing but configuration capability. Figure 13 illustrates the *ConfigurableObjectType*. It is formally defined in Table 18. A concrete example is provided in Clause 9.

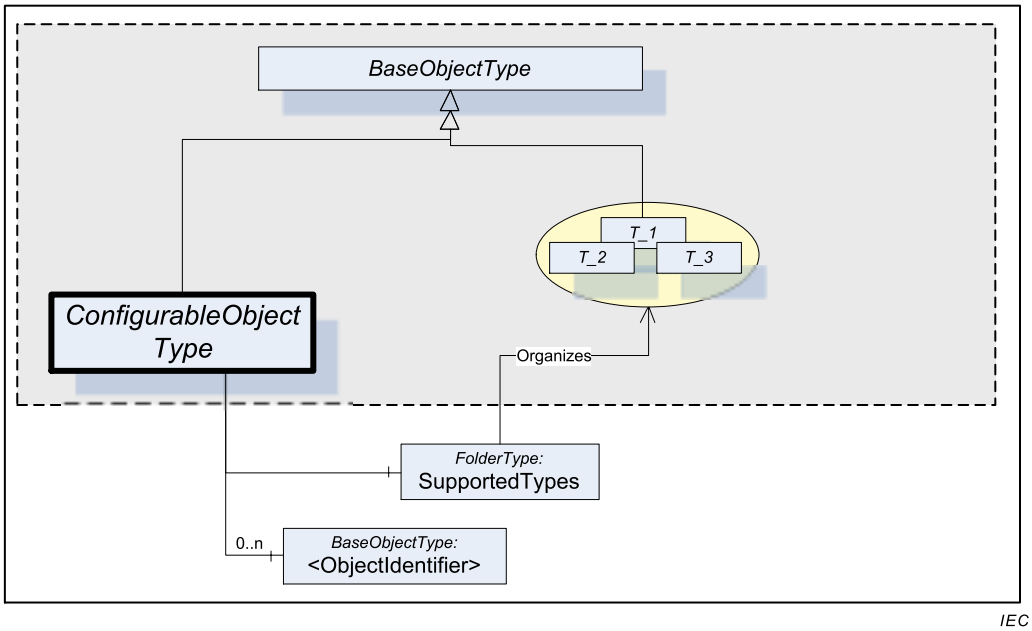


Figure 13 – ConfigurableObjectType

Table 18 – ConfigurableObjectType definition

Attribute	**Value**				
BrowseName	ConfigurableObjectType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the *BaseObjectType* defined in IEC 62541-5					
HasComponent	Object	SupportedTypes		FolderType	Mandatory
HasComponent	Object	<ObjectIdentifier>		BaseObjectType	OptionalPlaceholder

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

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

6 Device vommunication model

6.1 General

Clause 6 introduces *References* and basic *TopologyElementTypes* needed to create a communication topology. Figure 14 provides an initial insight into topological relationships. More specific examples will follow.

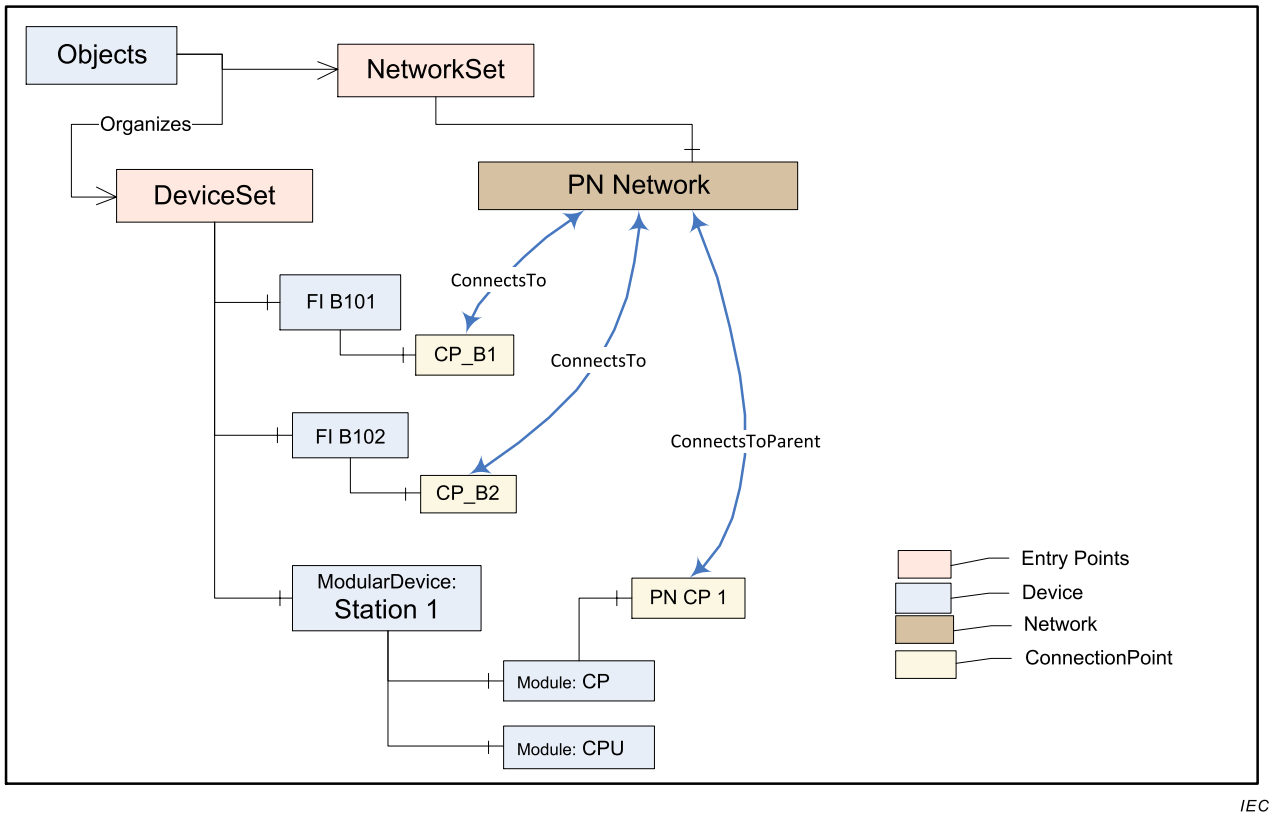


Figure 14 – Initial example of a communication topology

6.2 Network

A *Network* represents the communication means for *Devices* that are connected to it. It includes wired and wireless technologies. A *Network* instance is qualified by the protocol that it references.

Figure 15 shows the type hierarchy and the *NetworkType* components. It is formally defined in Table 19.

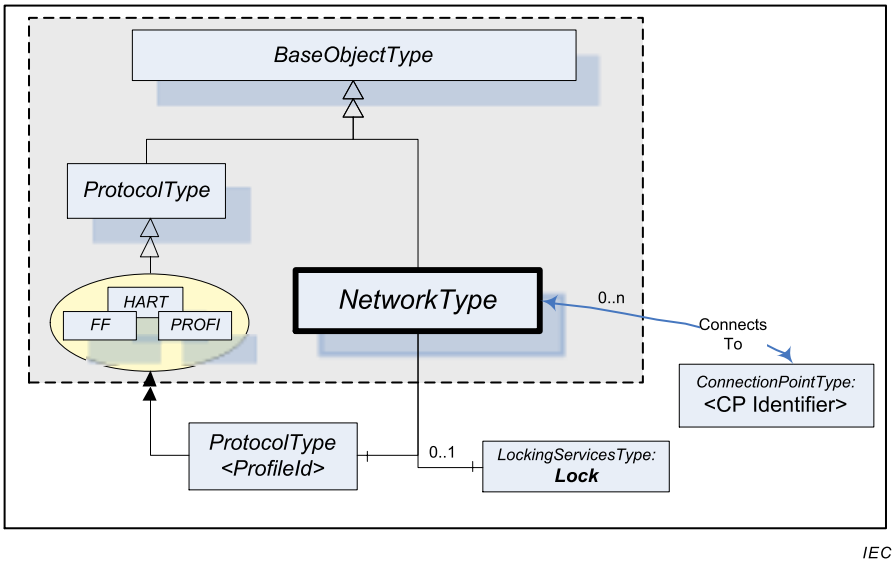


Figure 15 – NetworkType

Table 19 – NetworkType definition

Attribute	Value				
BrowseName	NetworkType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseObjectType defined in IEC 62541-5					
HasComponent	Object	<ProfileIdentifier>		ProtocolType	MandatoryPlaceholder
ConnectsTo	Object	<CPIdentifier>		ConnectionPointType	OptionalPlaceholder
HasComponent	Object	Lock		LockingServicesType	Optional

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

<CPIdentifier> (referenced by a *ConnectsTo Reference*) references the *ConnectionPoint*(s) that have been configured for this *Network*. All *ConnectionPoint*s shall adhere to the same *Protocol* as the *Network*. See also Figure 18 for a usage example. They represent the protocol-specific access points for the connected *Devices*.

In addition, *Networks* may also support *LockingServices* (defined in 8.3.3).

6.3 ConnectionPoint

This *ObjectType* represents the interface (interface card) of a *Device* to a *Network*. A specific sub-type shall be defined for each protocol. Figure 16 shows the *ConnectionPointType* including some specific types.

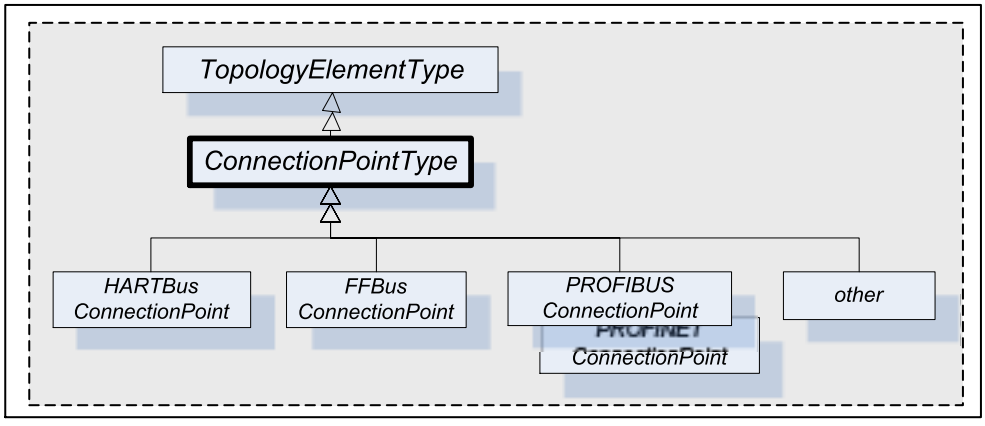


Figure 16 – Example of ConnectionPointType hierarchy

A *Device* can have more than one such interface to the same or to different *Networks*. Different interfaces usually exist for different protocols. Figure 17 shows the *ConnectionPointType* components. It is formally defined in Table 20.

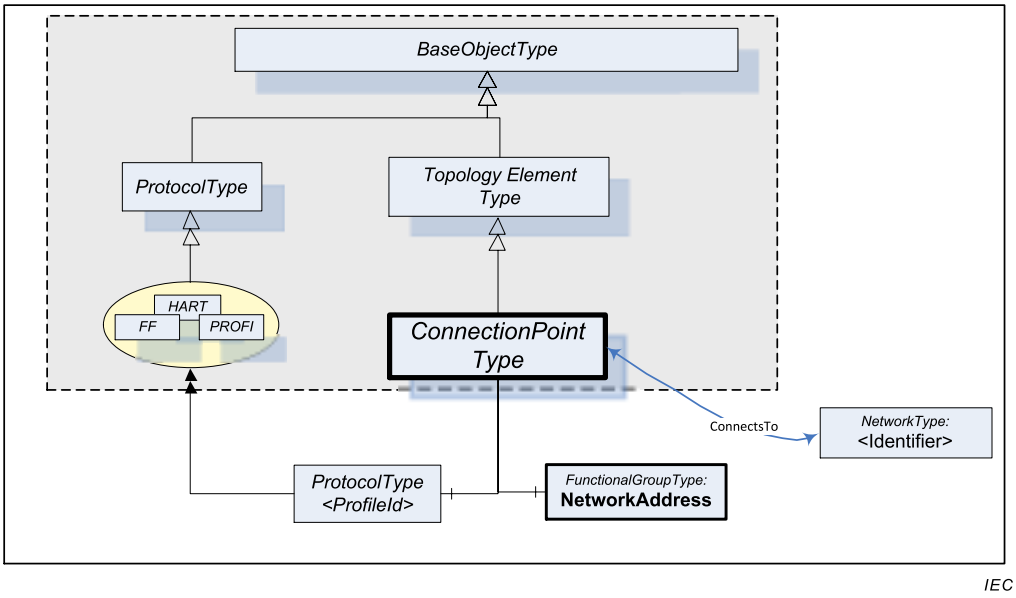


Figure 17 – ConnectionPointType

Table 20 – ConnectionPointType definition

Attribute	**Value**				
BrowseName	ConnectionPointType				
IsAbstract	True				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Inherit the Properties of the TopologyElementType defined in 5.2					
HasComponent	Object	NetworkAddress		FunctionalGroupType	Mandatory
HasComponent	Object	<ProfileIdentifier>		ProtocolType	MandatoryPlaceHolder
ConnectsTo	Object	<NetworkIdentifier>		NetworkType	OptionalPlaceHolder

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

The *NetworkAddress FunctionalGroup* organises all *Parameters* needed to specify the protocol-specific address information of the connected *Device*. These *Parameters* are also components of the *ParameterSet*. The *NetworkAddress Parameters* and their data types will be specified by standard organizations (e.g. Fieldbus organizations).

<ProfileIdentifier> identifies the *Communication Profile* that this *ConnectionPoint* supports. *ProtocolType* and *Communication Profile* are defined in 5.9. 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 18 illustrates some usage models.

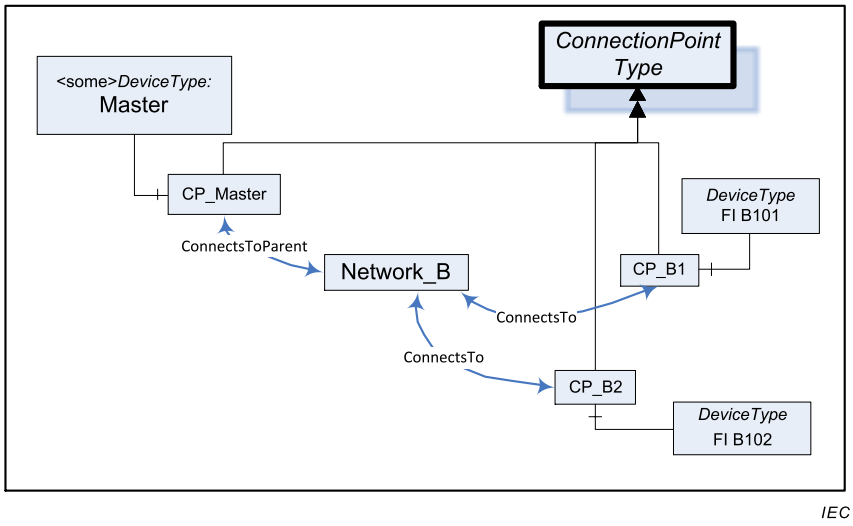


Figure 18 – ConnectionPoint usage

6.4 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 sub-type of The *ConnectsTo ReferenceType*.

The two *ReferenceTypes* are illustrated in Figure 19.

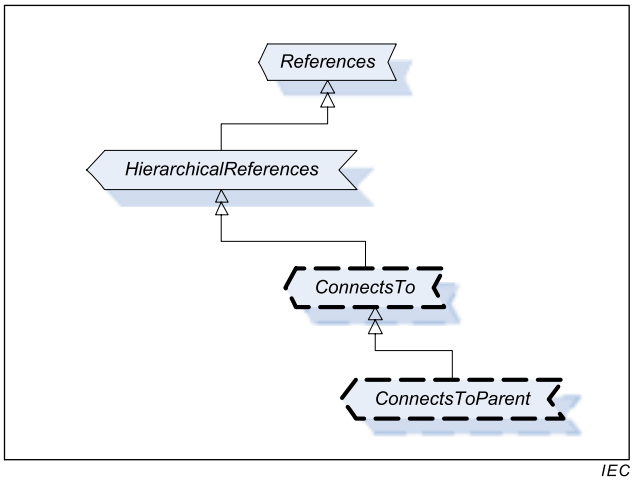


Figure 19 – Type hierarchy for ConnectsTo and ConnectsToParent References

The representation in the *AddressSpace* is specified in Table 21 and Table 22.

Table 21 – ConnectsTo ReferenceType

Attributes	**Value**		
BrowseName	ConnectsTo		
Symmetric	True		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of HierarchicalReferences ReferenceType defined in IEC 62541-5			

Table 22 – ConnectsToParent ReferenceType

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

Figure 20 illustrates how this *Reference* can be used to express topological and parental relationships. In this example two devices are connected; the module DPcomm is the communication *Device* for the *Network*.

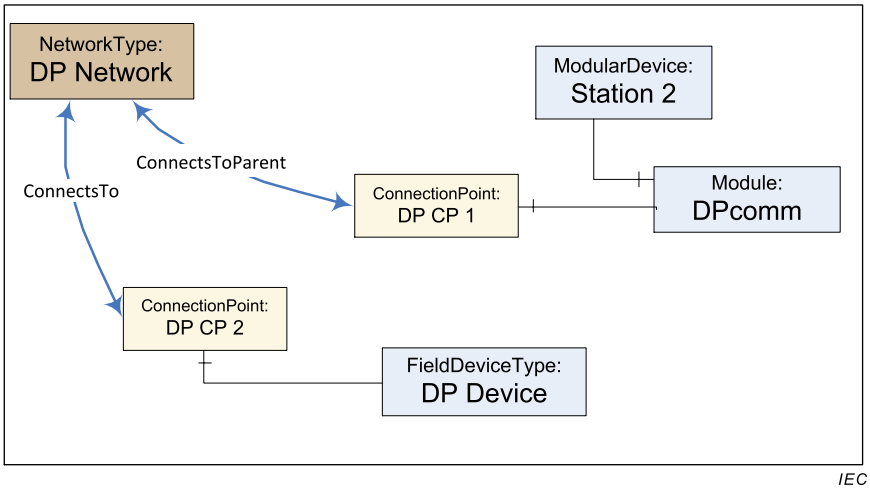


Figure 20 – Example with ConnectsTo and ConnectsToParent References

6.5 NetworkSet Object (mandatory)

All *Networks* are components of the **NetworkSet** *Object*.

The **NetworkSet** *Node* is formally defined in Table 23.

Table 23 – NetworkSet definition

Attribute	**Value**		
BrowseName	NetworkSet		
References	**NodeClass**	**BrowseName**	**TypeDefinition**
OrganizedBy by the Objects Folder defined in IEC 62541-5			
HasTypeDefinition	ObjectType	BaseObjectType	

7 Device integration host model

7.1 General

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

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

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

Figure 21 and Figure 22 illustrate an example configuration and the configured topology as it will appear in the *Server AddressSpace* (details left out).

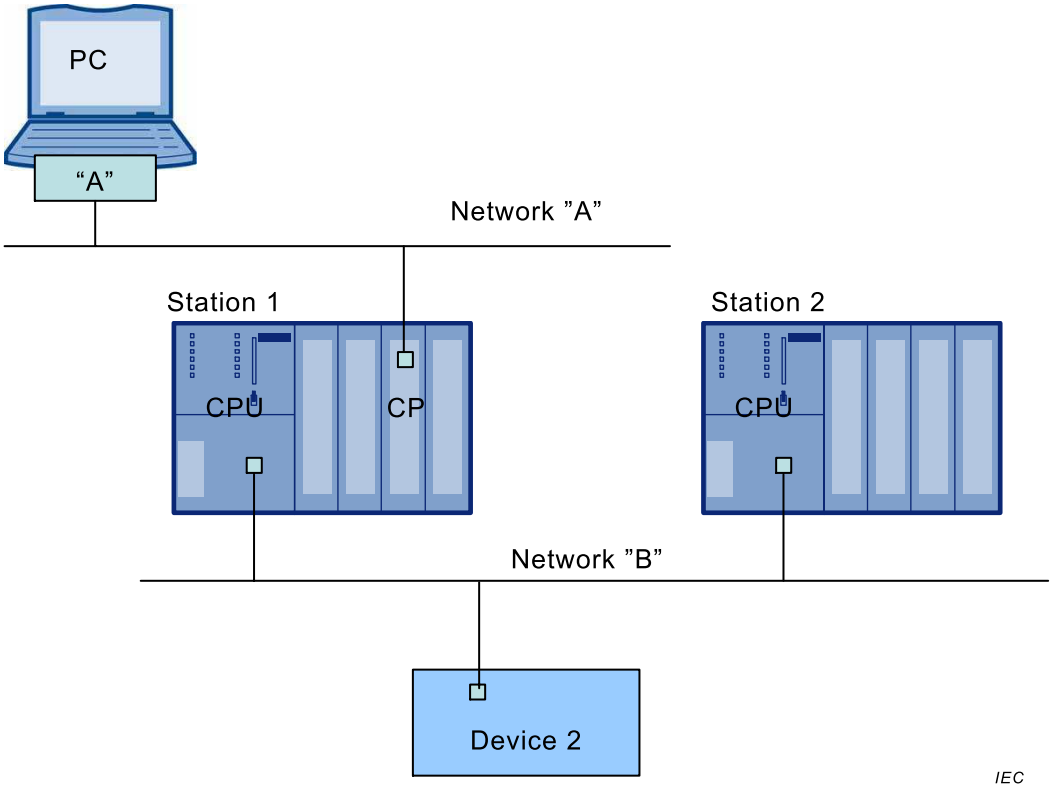


Figure 21 – Example of an automation system

The PC in Figure 21 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.

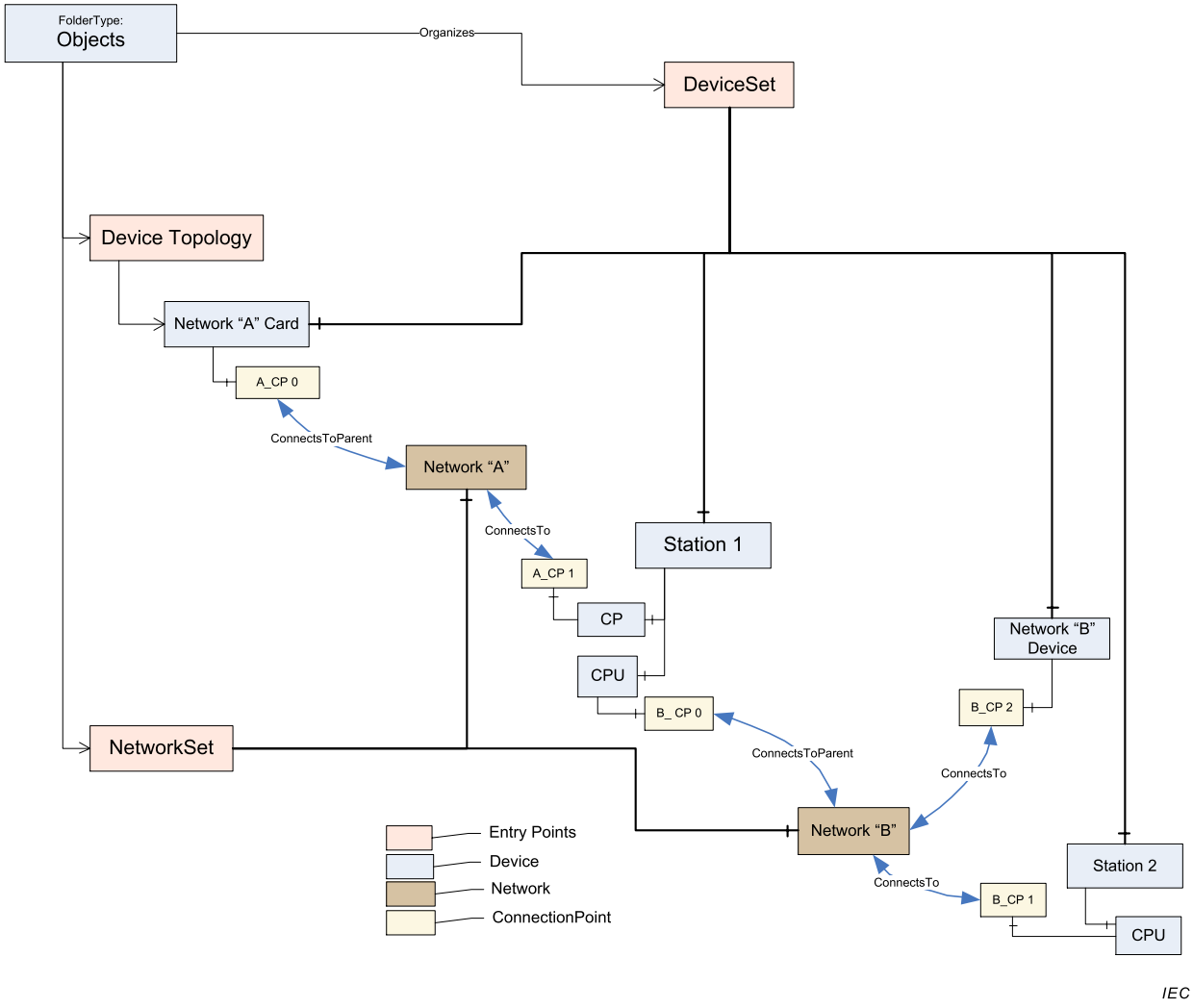


Figure 22 – Example of a Device topology

Coloured boxes are used to recognize the various types of information (see legend).

Entry points assure common behaviour across different implementations:

- **DeviceTopology**: Starting node for the topology configuration (see 7.2).
- **DeviceSet**: See s 5.8.
- **NetworkSet**: See 6.5.

7.2 DeviceTopology Object

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

The *DeviceTopology* node is formally defined in Table 24.

Table 24 – DeviceTopology definition

Attribute	Value			
BrowseName	DeviceTopology			
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**
OrganizedBy by the Objects Folder defined in IEC 62541-5				
HasTypeDefinition	ObjectType	BaseObjectType	Defined in IEC 62541-5.	
HasProperty	Variable	OnlineAccess	Boolean	PropertyType

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

7.3 Online/Offline

7.3.1 General

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

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

This is illustrated in Figure 23.

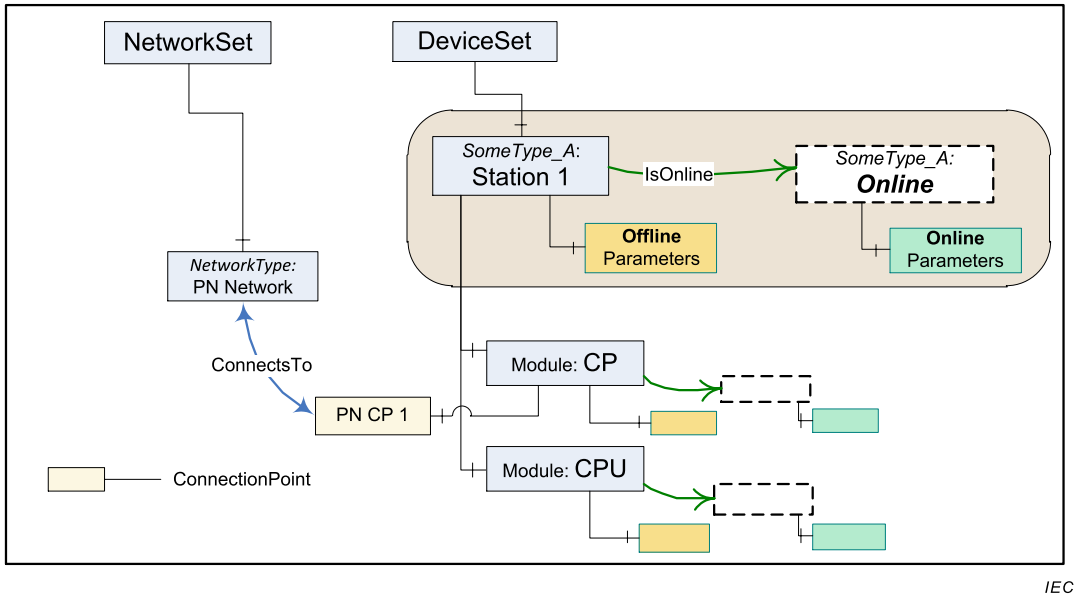


Figure 23 – Online component for access to device data

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. If the instance is a sub-type of *DeviceType* its *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 23. The **Online** *Object* is referenced via an *IsOnline Reference*. It is always of the same *ObjectType* as the offline instance.

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

The transfer of information (*Parameters*) between offline nodes and the physical device in correct order is supported through *TransferToDevice*, *TransferFromDevice* together with *FetchTransferResultData*. These *Methods* are exposed by means of an *AddIn* instance of *TransferServicesType* described in 8.2.

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* (see 5.4) organises *Parameters* that help identify a topology element. *Clients* can compare the values of these *Parameters* in the online and the offline instance to detect mismatches between the configuration data and the currently connected element.

7.3.2 IsOnline ReferenceType

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

The *IsOnline ReferenceType* is illustrated in Figure 24. Its representation in the *AddressSpace* is specified in Table 25.

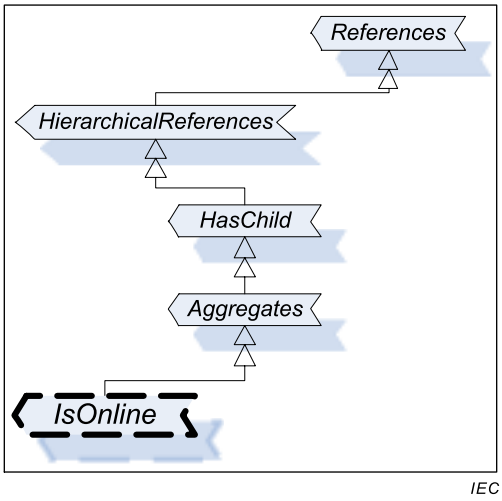


Figure 24 – Type hierarchy for *IsOnline Reference*

Table 25 – IsOnline *ReferenceType*

Attributes	**Value**		
BrowseName	IsOnline		
InverseName	OnlineOf		
Symmetric	False		
IsAbstract	False		
References	**NodeClass**	**BrowseName**	**Comment**
Subtype of Aggregates ReferenceType defined in IEC 62541-5			

8 AddIn Capabilities

8.1 Overview

The *Type* model of OPC UA is targeted towards a single inheritance, i.e., an *ObjectType* or a *VariableType* is assumed to be sub-type of only one super-type. Besides sub-typing, however, OPC UA in general allows extending types or instances with additional components. OPC UA describes a 'Best Practice', called ***AddIn Capability*,** of how to extend *Objects* and *Variables* in OPC UA and avoid multiple inheritances. AddIn Capabilities are comparable to the interface technology found in some programming languages.

Each *AddIn Capability* is modelled as an *ObjectType*, indistinguishable from other *ObjectTypes* used to expose full functionality with one exception: instances of AddIn *ObjectTypes* will always have pre-defined *BrowseNames*.

When calling a *Method* of an *AddIn Object*, the *ObjectId* shall identify the *AddIn Object* that is a component of the *Object* to be locked.

NOTE This can best be illustrated with Figure 28. In this figure the *Object* named "Lock" is the *AddIn Object*. It has several *Methods*, e.g., InitLock and ExitLock. When calling InitLock, the caller has to specify the *NodeId* of the "Lock" instance which is a component of MD002 in the *ObjectId* argument of the *Call Service*.

The following features are based on this *AddIn Capability* concept.

8.2 Offline-Online data transfer

8.2.1 Definition

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

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

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

The *Device* shall have been locked by the *Client* prior to invoking these *Methods* (see 0).

8.2.2 TransferServices Type

The *TransferServicesType* provides the *Methods* needed to transfer data to and from the online *Device*. Figure 25 shows the *TransferServicesType* definition. It is formally defined in Table 26.

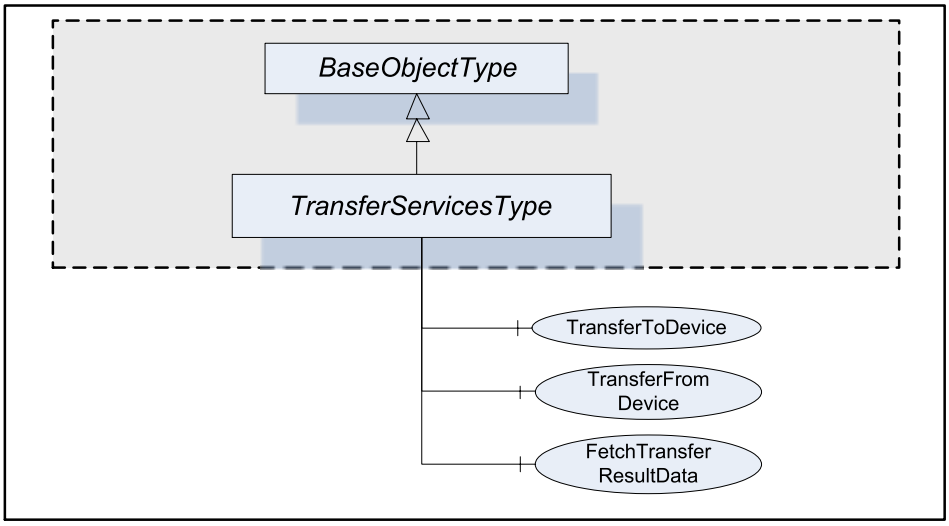


Figure 25 – TransferServicesType

Table 26 – TransferServicesType definition

Attribute	**Value**				
BrowseName	TransferServicesType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseObjectType defined in IEC 62541-5					
HasComponent	Method	TransferToDevice			Mandatory
HasComponent	Method	TransferFromDevice			Mandatory
HasComponent	Method	FetchTransferResultData			Mandatory

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

8.2.3 TransferServices Object

The support of *TransferServices* for an *Object* is declared by aggregating an instance of the *TransferServicesType* as illustrated in Figure 26.

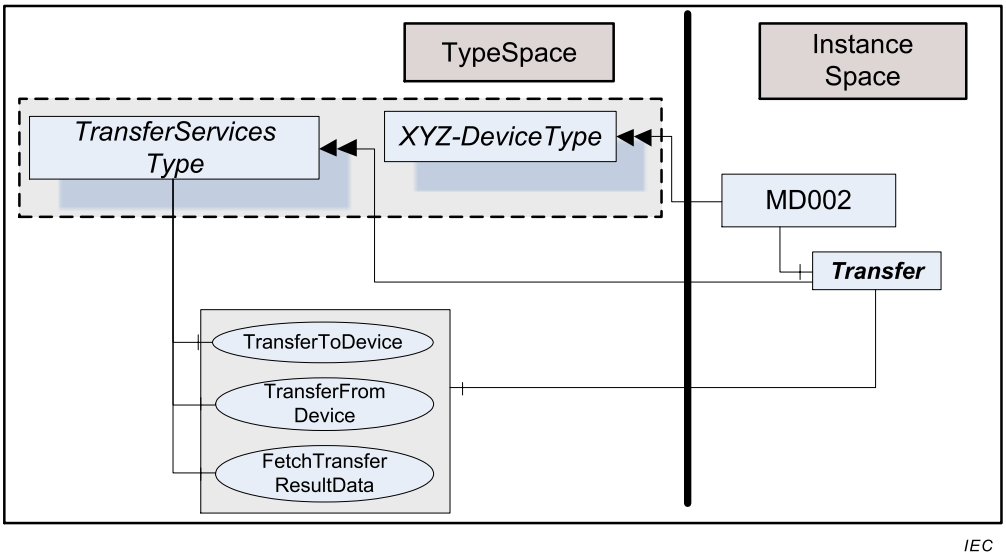


Figure 26 – TransferServices

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

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

8.2.4 TransferToDevice Method

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

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

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

The signature of this *Method* is specified below. Table 27 and Table 28 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 27 – 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 28 – TransferToDevice Method AddressSpace definition

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

8.2.5 TransferFromDevice Method

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

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

The signature of this *Method* is specified below. Table 29 and Table 30 specify the arguments and *AddressSpace* representation, respectively.

Signature

```
TransferFromDevice(

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

Table 29 – 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 / can not be accessed

Table 30 – TransferFromDevice Method AddressSpace definition

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

8.2.6 FetchTransferResultData Method

The *TransferToDevice* and *TransferFromDevice Methods* execute asynchronously after sending a response to the *Client*. Excution 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 31 and Table 32 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 31 – 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 sub-types 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 32 – 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 33.

Table 33 – FetchResultDataType structure

Attribute	Value		
BrowseName	FetchResultDataType		
IsAbstract	True		
Subtype of Structure defined in IEC 62541-3			
References	**NodeClass**	**BrowseName**	**DataType**
HasSubtype	DataType	TransferResultErrorDataType	Defined in Table 34.
HasSubtype	DataType	TransferResultDataDataType	Defined in Table 35.

The *TransferResultErrorDataType* is a subtype of the *FetchResultDataType* and represents an error result. It is defined in Table 34.

Table 34 – 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 IEC 62541-4.

The *TransferResultData DataType* is a subtype of the *FetchResultDataType* and includes parameter-results from the transfer operation. It is defined in Table 35.

Table 35 – TransferResultData DataType structure

Name	Type	Description
TransferResultData DataType	Structure	A set of results from the transfer operation.
sequenceNumber	Int32	The sequence number of this result set.
endOfResults	Boolean	TRUE – all result data have been fetched. Additional *FetchTransferResultData* calls with the same *TransferID* will return a FetchTransferError with status=InvalidTransferId. FALSE – further result data shall be expected.
parameterDefs	structure[]	Specific value for each *Parameter* that has been transferred. If OmitGoodResults is TRUE, parameterDefs will only contain *Parameters* which have not been transferred correctly.
NodePath	QualifiedName[]	List of BrowseNames that represent the relative path from the *Device Object* to the *Parameter* following hierarchical references. The *Client* may use these names for *TranslateBrowsePathsToNodeIds* to retrieve the *Parameter NodeId* for the online or the offline representation.
statusCode	StatusCode	OPC UA *StatusCode* as defined in IEC 62541-4 and in IEC 62541-8.
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 IEC 62541-4.

8.3 Locking

8.3.1 Overview

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

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

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

When locking a *Modular Device* (see 9.3), the lock applies to the complete *Device* (including all modules). Equally, when locking a block-oriented *Device* (see 9.2), the lock applies to the complete *Device* (including all blocks). *Servers* may allow independent locking of sub-*Devices* or blocks, respectively, if no lock is applied to the top-level *Device* (for *Modular Device* or for *Block Device*).

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

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

8.3.2 LockingServices Type

The *LockingServicesType* provides the *Methods* needed to lock or unlock. Figure 27 shows the *LockingServicesType* definition. It is formally defined in Table 36.

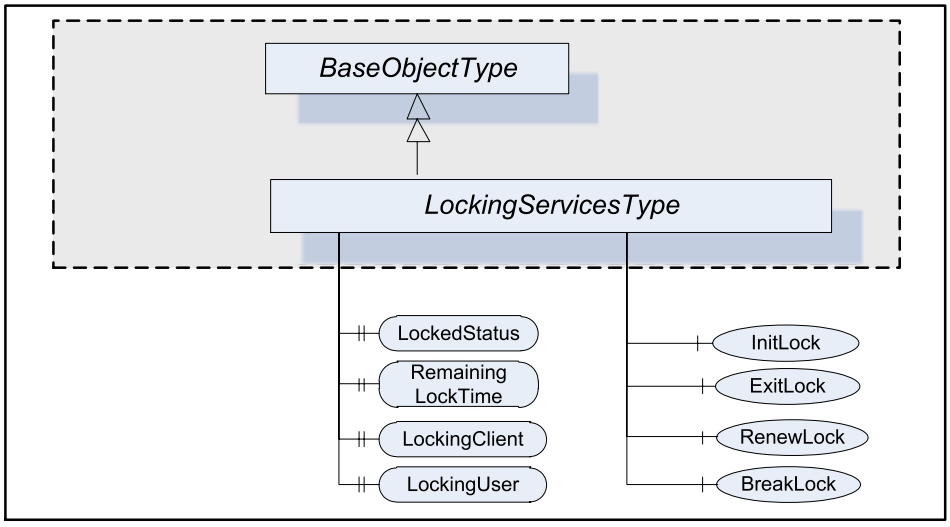


Figure 27 – LockingServicesType

Table 36 – LockingServicesType definition

Attribute	**Value**				
BrowseName	LockingServicesType				
IsAbstract	False				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**ModellingRule**
Subtype of the BaseObjectType defined in IEC 62541-5					
HasComponent	Method	InitLock			Mandatory
HasComponent	Method	RenewLock			Mandatory
HasComponent	Method	ExitLock			Mandatory
HasComponent	Method	BreakLock			Mandatory
HasProperty	Variable	Locked	Boolean	PropertyType	Mandatory
HasProperty	Variable	LockingClient	String	PropertyType	Mandatory
HasProperty	Variable	LockingUser	String	PropertyType	Mandatory
HasProperty	Variable	RemainingLockTime	Duration	PropertyType	Mandatory

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

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

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

LockingClient contains the ApplicationUri of the *Client* as provided in the *CreateSession Service* call (see IEC 62541-4).

LockingUser contains the identity of the user. It is obtained directly or indirectly from the *UserIdentityToken* passed by the *Client* in the *ActivateSession Service* call (see IEC 62541-4).

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

8.3.3 LockingServices Object

The support of *LockingServices* for an *Object* is declared by aggregating an instance of the *LockingServicesType* as illustrated in Figure 28.

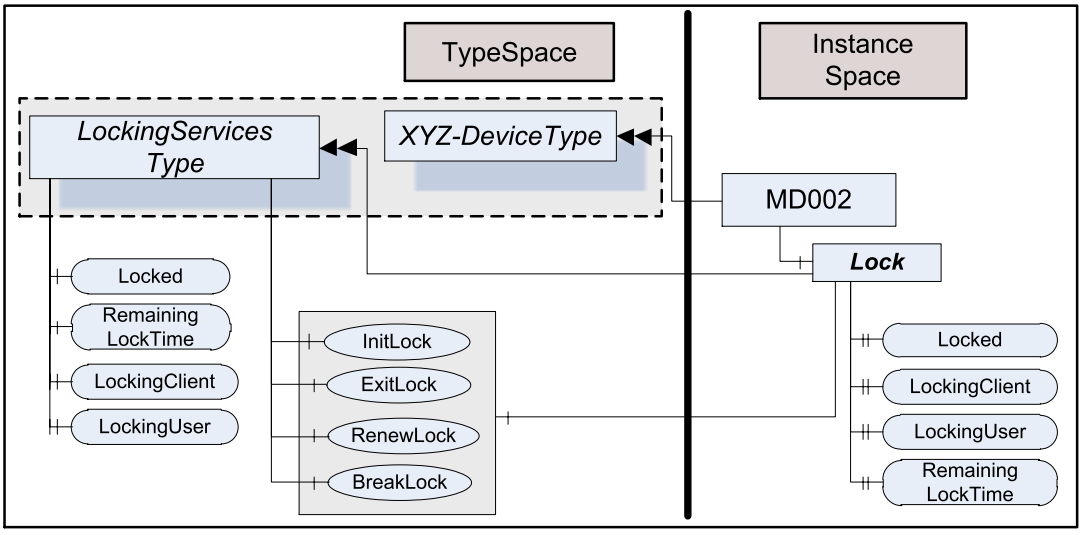


Figure 28 – LockingServices

This *Object* is used as container for the *LockingServices Methods* and *Properties* and shall have the *BrowseName* **Lock**. *HasComponent* is used to reference from a *TopologyElement* (for example, a *Device*) to its "LockingServices" *Object*.

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

8.3.4 MaxInactiveLockTime Property

The *MaxInactiveLockTime Property* shall be added to the *ServerCapabilities Object* (see IEC 62541-5).

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

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

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

The *MaxInactiveLockTime Property* is formally defined in Table 37.

Table 37 – MaxInactiveLockTime Property definition

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

8.3.5 InitLock Method

InitLock restricts access for other *Clients*.

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

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

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

The signature of this *Method* is specified below. Table 38 and Table 39 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 38 – 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 39 – InitLock Method AddressSpace definition

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

8.3.6 ExitLock Method

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

The signature of this *Method* is specified below. Table 40 and Table 41 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 40 – ExitLock Method Arguments

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

Table 41 – ExitLock Method AddressSpace definition

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

8.3.7 RenewLock Method

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

The signature of this *Method* is specified below. Table 42 and Table 43 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 42 – RenewLock Method Arguments

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

Table 43 – RenewLock Method AddressSpace definition

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

8.3.8 BreakLock Method

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

The signature of this *Method* is specified below. Table 44 and Table 45 specify the arguments and *AddressSpace* representation, respectively.

Signature

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

Table 44 – BreakLock Method Arguments

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

Table 45 – BreakLock Method AddressSpace definition

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

9 Specialized topology elements

9.1 General

Devices and other *TopologyElements* can be enhanced using the modeling elements defined in this standard. No derived *ObjectType* is needed.

9.2 Block Devices (BlockOriented DeviceType)

A block-oriented *Device* can be composed using the modelling elements defined in this standard. A block-oriented *Device* includes a configurable set of *Blocks*. Figure 29 shows the general structure of block-oriented *Devices*.

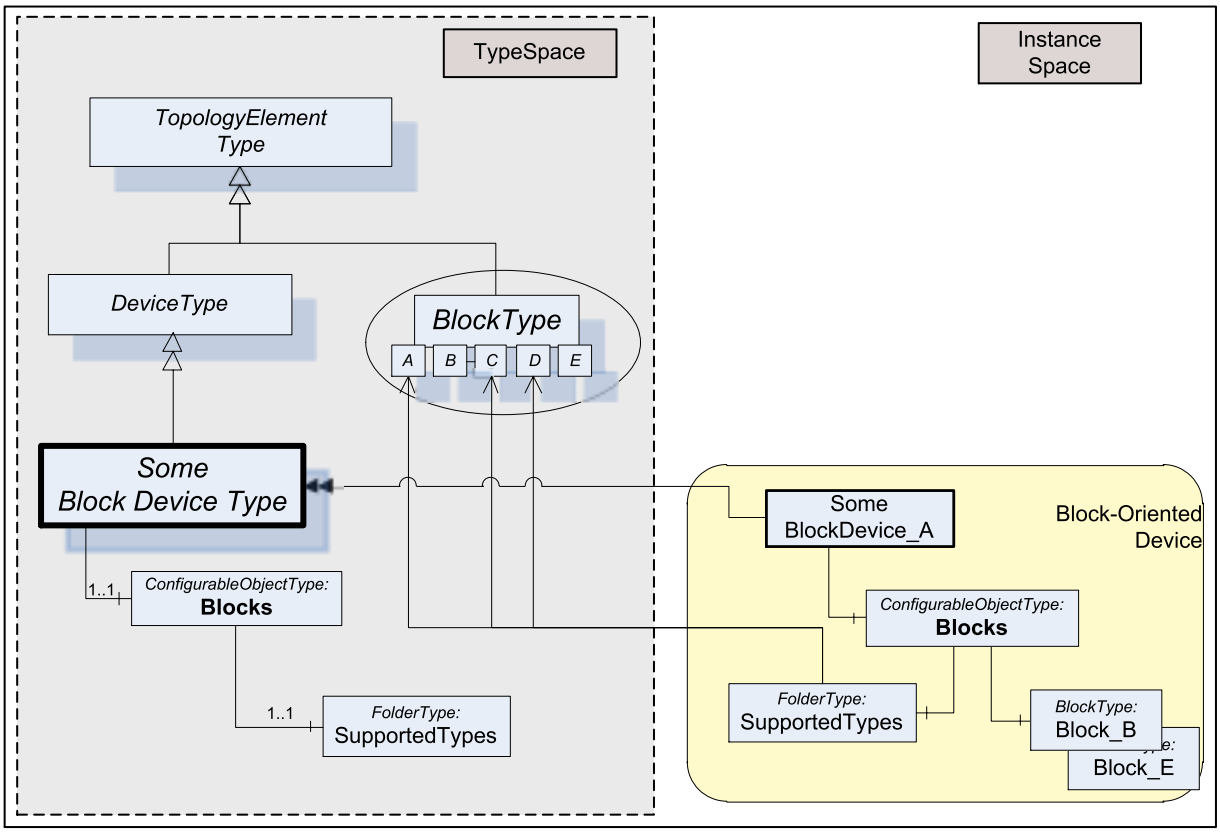


Figure 29 – Block-oriented Device structure example

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 (sub-types of) *BlockTypes* that can be instantiated. The supported *Blocks* may be restricted by the block-oriented *Device*. In Figure 29 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 5.11.1 for the complete definition of the *ConfigurableObjectType*.

9.3 Modular Devices

A *Modular Device* is a (sub-type of) *Device* 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 30 shows the general structure of *Modular Devices*.

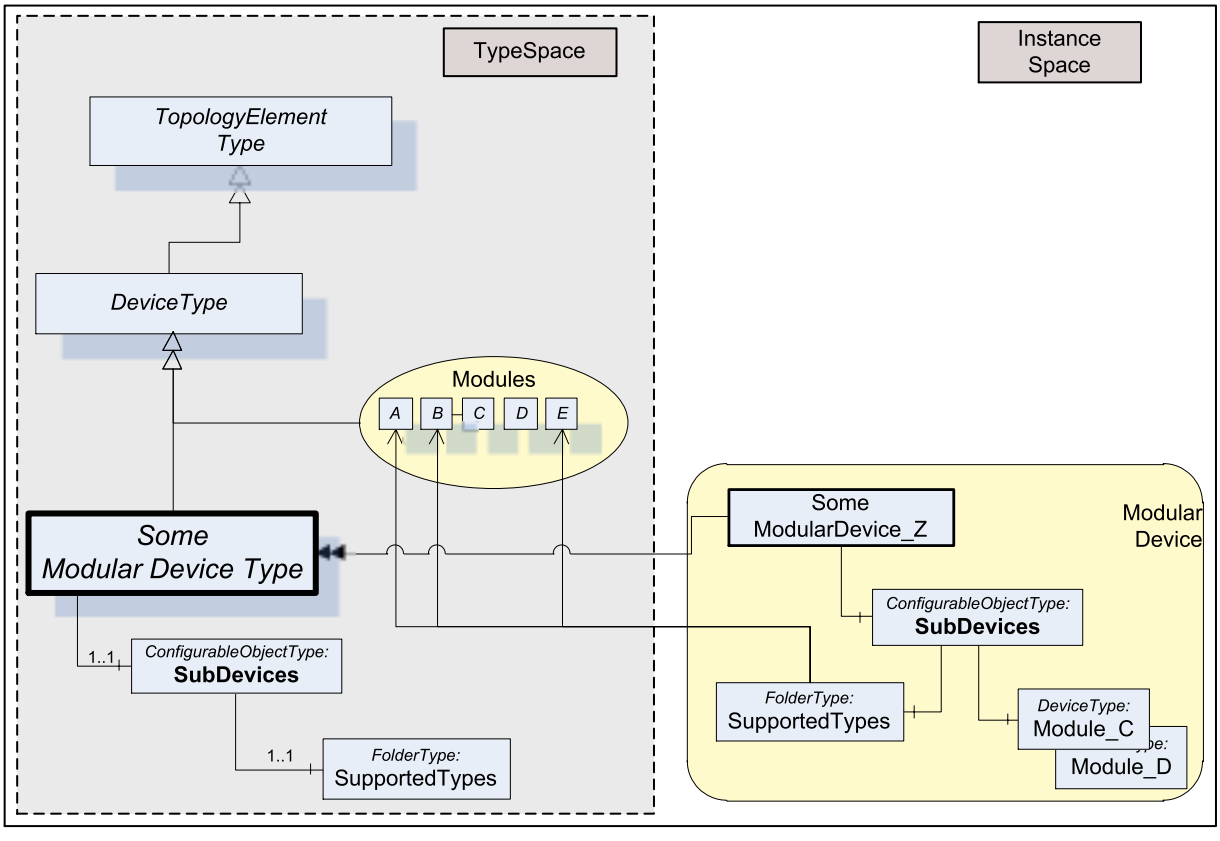


Figure 30 – Modular Device structure example

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 of (sub-types of) *DeviceTypes* that can be added to the *Modular Device*. 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 30 the DeviceTypes 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 5.11.1 for the complete definition of the *ConfigurableObjectType*.

Subdevices may themselves be *Modular Devices*.

10 Profiles

10.1 General

Profiles are named groupings of *ConformanceUnits* as defined in IEC 62541-7. 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 standard defines OPC UA *Profile* facets for *Servers* or *Clients* when they plan to support OPC UA for Devices. They are described in 10.2 and 10.3.

10.2 Device Server Facets

The following tables specify the facets available for *Servers* that implement the *Devices* companion standard. Table 46 describes *Conformance Units* included in the minimum needed facet. It includes the organisation of instantiated *Devices* in the *Server AddressSpace*.

Table 46 – BaseDevice_Server_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI Information Model	Supports *Objects* that conform to the types specified in the *Device* companion standard. This includes in particular *Objects* of *DeviceType* and *FunctionalGroups*.	M
DI DeviceSet	Supports the **DeviceSet** object to aggregate all *Device* instances.	M
DI DeviceHealth	Supports the DeviceHealth Property.	O
DI DeviceSupportInfo	Server provides additional data for its Devices as defined in 5.7.	O

Table 47 defines a facet for the identification *FunctionalGroup* of *Devices*. This includes the option of identifying the *Protocol*(s).

Table 47 – DeviceIdentification_Server_Facet definition

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

Table 48 defines extensions specifically needed for *BlockDevices*.

Table 48 – BlockDevice_Server_Facet definition

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

Table 49 defines a facet for the Locking *AddIn Capability*. This includes the option of breaking a lock.

Table 49 – Locking_Server_Facet definition

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

Table 50 defines a facet for the support of the Device Communication model.

Table 50 – DeviceCommunication_Server_Facet definition

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

Table 51 defines a facet for the support of the Device Integration Host model.

Table 51 – DeviceIntegrationHost_Server_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI DeviceTopology	Supports the DeviceTopology object as starting node for the communication topology of the devices to integrate.	M
DI Offline	Supports offline and online representations of devices including the methods to transfer data from or to the device.	M

10.3 Device Client Facets

The following tables specify the facets available for *Clients* that implement the *Devices* companion standard. Table 52 describes *Conformance Units* included in the minimum needed facet.

Table 52 – BaseDevice_Client_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI Client Information Model	Consumes *Objects* that conform to the types specified in the *Device* companion standard. This includes in particular *Objects* of *DeviceType* and *FunctionalGroups*.	M
DI Client DeviceSet	Uses the **DeviceSet** *Object* to detect available *Devices*.	M
DI Client DeviceHealth	Uses the DeviceHealth Property.	O
DI Client DeviceSupportInfo	Uses available additional data for *Devices* as defined in 5.7.	O

Table 53 defines a facet for the identification *FunctionalGroup* of *Devices*. This includes the option of identifying the *Protocol*(s).

Table 53 – DeviceIdentification_Client_Facet definition

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

Table 54 defines extensions specifically needed for BlockDevices.

Table 54 – BlockDevice_Client_Facet definition

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

Table 55 defines a facet for the Locking *AddIn Capability*. This includes the option of breaking a lock.

Table 55 – Locking_Client_Facet definition

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

Table 56 defines a facet for the support of the Device Communication model.

Table 56 – DeviceCommunication_Client_Facet definition

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

Table 57 defines a facet for the support of the Device Integration Host model.

Table 57 – DeviceIntegrationHost_Client_Facet definition

Conformance Unit	Description	Optional/ Mandatory
DI DeviceTopology	Uses the DeviceTopology object as starting node for the communication topology of the devices to integrate.	M
DI Offline	Uses offline and online representations of devices including the methods to transfer data from or to the device.	M

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 standard 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 **Name** 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/1.1/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 IEC 62541-6.

The Information Model Schema released with this version of the standard can be found at:
http://www.opcfoundation.org/UADevices/1.1/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/1.1/Opc.Ua.Di.NodeSet2.xml

Bibliography

IEC 61131 (all parts), *Programmable controllers*

IEC 61499-1:2012, *Function blocks – Part 1: Architecture*

IEC 61784 (all parts), *Industrial Communication Networks – Profiles*

IEC 62591, *Industrial communication networks – Wireless communication network and communication profiles – WirelessHART™*

IEC 62769 (all parts), *Field Device Integration (FDI)*

UA Companion ADI, *OPC UA Companion Specification for Analyser Devices*

UA Companion PLCopen, *OPC UA Companion Specification for PLCopen*

SOMMAIRE

COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

_____

ARCHITECTURE UNIFIÉE OPC –

Partie 100: Interface d'appareils

AVANT-PROPOS

1) La Commission Électrotechnique Internationale (IEC) est une organisation mondiale de normalisation composée de l'ensemble des comités électrotechniques nationaux (Comités nationaux de l'IEC). L'IEC a pour objet de favoriser la coopération internationale pour toutes les questions de normalisation dans les domaines de l'électricité et de l'électronique. À cet effet, l'IEC – entre autres activités – publie des Normes internationales, des Spécifications techniques, des Rapports techniques, des Spécifications accessibles au public (PAS) et des Guides (ci-après dénommés "Publication(s) de l'IEC"). Leur élaboration est confiée à des comités d'études, aux travaux desquels tout Comité national intéressé par le sujet traité peut participer. Les organisations internationales, gouvernementales et non gouvernementales, en liaison avec l'IEC, participent également aux travaux. L'IEC collabore étroitement avec l'Organisation Internationale de Normalisation (ISO), selon des conditions fixées par accord entre les deux organisations.
2) Les décisions ou accords officiels de l'IEC concernant les questions techniques représentent, dans la mesure du possible, un accord international sur les sujets étudiés, étant donné que les Comités nationaux de l'IEC intéressés sont représentés dans chaque comité d'études.
3) Les Publications de l'IEC se présentent sous la forme de recommandations internationales et sont agréées comme telles par les Comités nationaux de l'IEC. Tous les efforts raisonnables sont entrepris afin que l'IEC s'assure de l'exactitude du contenu technique de ses publications; l'IEC ne peut pas être tenue responsable de l'éventuelle mauvaise utilisation ou interprétation qui en est faite par un quelconque utilisateur final.
4) Dans le but d'encourager l'uniformité internationale, les Comités nationaux de l'IEC s'engagent, dans toute la mesure possible, à appliquer de façon transparente les Publications de l'IEC dans leurs publications nationales et régionales. Toutes divergences entre toutes Publications de l'IEC et toutes publications nationales ou régionales correspondantes doivent être indiquées en termes clairs dans ces dernières.
5) L'IEC elle-même ne fournit aucune attestation de conformité. Des organismes de certification indépendants fournissent des services d'évaluation de conformité et, dans certains secteurs, accèdent aux marques de conformité de l'IEC. L'IEC n'est responsable d'aucun des services effectués par les organismes de certification indépendants.
6) Tous les utilisateurs doivent s'assurer qu'ils sont en possession de la dernière édition de cette publication.
7) Aucune responsabilité ne doit être imputée à l'IEC, à ses administrateurs, employés, auxiliaires ou mandataires, y compris ses experts particuliers et les membres de ses comités d'études et des Comités nationaux de l'IEC, pour tout préjudice causé en cas de dommages corporels et matériels, ou de tout autre dommage de quelque nature que ce soit, directe ou indirecte, ou pour supporter les coûts (y compris les frais de justice) et les dépenses découlant de la publication ou de l'utilisation de cette Publication de l'IEC ou de toute autre Publication de l'IEC, ou au crédit qui lui est accordé.
8) L'attention est attirée sur les références normatives citées dans cette publication. L'utilisation de publications référencées est obligatoire pour une application correcte de la présente publication.
9) L'attention est attirée sur le fait que certains des éléments de la présente Publication de l'IEC peuvent faire l'objet de droits de brevet. L'IEC ne saurait être tenue pour responsable de ne pas avoir identifié de tels droits de brevets et de ne pas avoir signalé leur existence.

La Norme internationale IEC 62541-100 a été établie par le sous-comité 65E: Les dispositifs et leur intégration dans les systèmes de l'entreprise, du comité d'études 65 de l'IEC: Mesure, commande et automation dans les processus industriels.

Le texte de cette norme est issu des documents suivants:

CDV	Rapport de vote
65E/372/CDV	65E/412/RVC

Le rapport de vote indiqué dans le tableau ci-dessus donne toute information sur le vote ayant abouti à l'approbation de cette norme.

Cette publication a été rédigée selon les Directives ISO/IEC, Partie 2.

Une liste de toutes les parties de la série IEC 62541, publiées sous le titre général *Architecture unifiée OPC* , peut être consultée sur le site web de l'IEC.

Le comité a décidé que le contenu de cette publication ne sera pas modifié avant la date de stabilité indiquée sur le site web de l'IEC sous "http://webstore.iec.ch" dans les données relatives à la publication recherchée. À cette date, la publication sera

- reconduite,
- supprimée,
- remplacée par une édition révisée, ou
- amendée.

IMPORTANT – Le logo *"colour inside"* qui se trouve sur la page de couverture de cette publication indique qu'elle contient des couleurs qui sont considérées comme utiles à une bonne compréhension de son contenu. Les utilisateurs devraient, par conséquent, imprimer cette publication en utilisant une imprimante couleur.

ARCHITECTURE UNIFIÉE OPC –

Partie 100: Interface d'appareils

1 Domaine d'application

La présente partie de l'IEC 62541 est une extension de la série globale de normes de l'OPC UA (architecture unifiée OPC) et définit le modèle d'information associé aux *Appareils*. La présente spécification décrit trois modèles qui se construisent les uns sur les autres comme suit:

- le Modèle d'appareils (de base) vise à fournir une vue unifiée des appareils, et ce, indépendamment des protocoles d'appareils sous-jacents;
- le Modèle de communication d'appareils ajoute des éléments informationnels relatifs au réseau et à la connexion qui permettent ainsi de créer des topologies de communication;
- le Modèle d'hôte d'intégration d'appareils enfin ajoute des éléments et des règles complémentaires qui sont requis pour que les systèmes gèrent l'intégration pour un système complet. Il permet de refléter la topologie du système d'automation avec les appareils ainsi que les réseaux de communication qui se connectent.

2 Références normatives

Les documents suivants sont cités en référence de manière normative, en intégralité ou en partie, dans le présent document et sont indispensables pour son application. Pour les références datées, seule l'édition citée s'applique. Pour les références non datées, la dernière édition du document de référence s'applique (y compris les éventuels amendements).

IEC TR 62541-1, *OPC Unified Architecture – Part 1: Overview and Concepts* (disponible en anglais seulement)

IEC 62541-3, *Architecture unifiée OPC – Partie 3: Modèle de l'Espace d'Adressage*

IEC 62541-4, *Architecture unifiée OPC – Partie 4: Services*

IEC 62541-5, *Architecture unifiée OPC – Partie 5: Modèle d'Informations*

IEC 62541-6, *Architecture unifiée OPC – Partie 6: Correspondances*

IEC 62541-7, *Architecture unifiée OPC – Partie 7: Profils*

IEC 62541-8, *Architecture unifiée OPC – Partie 8: Accès aux données*

NAMUR Recommendation NE107: *Self-monitoring and diagnosis of field devices* (disponible en anglais seulement)

3 Termes, définitions, abréviations et types de données utilisés

3.1 Termes et définitions

Pour les besoins du présent document, les termes et définitions donnés dans l'IEC TR 62541-1, l'IEC 62541-3, et l'IEC 62541-8 ainsi que les suivants s'appliquent.

3.1.1
block
bloc
une entité de groupage fonctionnelle de *Paramètres*

Note 1 à l'article: Il peut se mapper à un bloc fonctionnel (voir IEC 62769 (toutes les parties), *Intégration des appareils de terrain (FDI)*

) ou aux paramètres de ressources de l'appareil lui-même.

3.1.2
blockMode
mode de bloc
mode de fonctionnement (mode cible, modes permis, mode réel et mode normal) pour un *Block*

Note 1 à l'article: D'autres détails relatifs aux modes de *Block* sont définis par des organisations de normalisation.

3.1.3
Communication Profile
Profil de communication
ensemble fixe de règles de mapping pour permettre respectivement une interopérabilité sans ambiguïté entre *Appareils* ou Applications

Note 1 à l'article: Des exemples de ces profils sont les “Réseaux de communication et profils de communication sans fil pour WirelessHART” dans l'IEC 62591 et les Mappings de protocoles pour l'OPC UA dans l'IEC 62541-6.

3.1.4
Connection Point
Point de connexion
représentation logique de l'interface entre un *Appareil* et un *Réseau*

3.1.5
device
appareil
entité physique indépendante capable d'accomplir une ou plusieurs fonctions spécifiées dans un contexte particulier et délimitée par ses interfaces

Note 1 à l'article: Voir IEC 61499-1.

Note 2 à l'article: Les *appareils* types fournissent des fonctionnalités de détection, d'actionnement, de communication et/ou de commande. Exemples: transmetteurs, appareils de commande de soupapes, entraînements, appareils de commande de moteur, automates programmables et passerelles de communication.

3.1.6
Device Integration Host
Hôte d'intégration d'appareils
Serveur qui gère l'intégration de plusieurs *Appareils* dans un système d'automation

3.1.7
Device Topology
Topologie d'appareils
agencement de *Réseaux* et d'*Appareils* qui constituent une topologie de communication

3.1.8
fieldbus
bus de terrain
système de communication basé sur le transfert de données en série et utilisé dans des applications d’automatisation industrielle ou de commande de processus

Note 1 à l'article: Voir l’IEC 61784.

Note 2 à l'article: Désigne le bus de communication utilisé par un *Appareil*.

3.1.9
parameter
paramètre
variable de l'*Appareil* qui peut être utilisée pour des besoins de configuration, de surveillance ou de commande

Note 1 à l'article: Dans le modèle d'information, il est synonyme de *DataVariable* (variable de données) de l'OPC UA.

3.1.10
network (réseau)
moyen utilisé pour communiquer avec un protocole spécifique

3.2 Abréviations

ADI Analyser Device Integration (Intégration d'appareils d'analyse)
CP Communication Processor (processeur de communication) (module matériel)
CPU (UCT) Central Processing Unit (Unité centrale de traitement d'un *Appareil*)
DA Data Access (Accès aux données)
DI Device Integration (Intégration d'appareil) (le diminutif de la présente norme)
UA Unified Architecture (Architecture unifiée)
UML Unified Modelling Language (Langage de modélisation unifié)
XML Extensible Mark-up Language (Langage de balisage étendu)

3.3 Types de données utilisés

Le Tableau 1 décrit les *DataTypes* qui sont utilisés tout au long du présent document.

Tableau 1 – DataTypes définis dans l'IEC 62541-3

Type du paramètre
Argument
Boolean
Duration
LocalizedText
String
Int32

4 Principes essentiels

4.1 OPC UA

Le cas d'utilisation principale pour les normes OPC est l'échange de données en ligne entre des dispositifs et des systèmes IHM ou SCADA utilisant une fonctionnalité d'accès aux données. Dans ce cas d'utilisation, les données d'un dispositif sont fournies par un *Serveur* d'OPC et sont consommées par un *Client* OPC intégré dans le système IHM ou SCADA. L'OPC DA fournit la fonctionnalité pour naviguer à travers des espaces de noms hiérarchiques contenant des éléments de données et pour lire, écrire et surveiller ces éléments pour des modifications de données. Les normes OPC classiques sont basées sur la technologie de COM/DCOM de Microsoft pour la communication entre des composants logiciels provenant de différents vendeurs. Par conséquent, le *Serveur* et les *Clients* OPC sont limités aux systèmes d'automation basés sur PC Windows.

L'architecture OPC UA comporte toutes les caractéristiques des normes OPC classiques comme OPC DA, A&E et HDA mais elle définit les mécanismes de communication

indépendants des plateformes et des capacités de modélisation génériques, orientées objet et extensibles pour les informations qu'un système souhaite exposer.

La partie communication de réseau de l'OPC UA définit différents mécanismes optimisés pour différents cas d'utilisation. La version courante de l'OPC UA définit un protocole binaire optimisé pour la communication intranet haute performance ainsi que les services web. Elle permet l'ajout de nouveaux protocoles dans le futur. Les caractéristiques telles que la sécurité, le contrôle d'accès et la fiabilité sont directement intégrées dans les mécanismes de transport. Sur la base de l'indépendance des protocoles vis-à-vis de toute plateforme, les *Serveur* et Clients OPC UA peuvent être directement intégrés dans les appareils et les contrôleurs.

Le *Modèle* d'*information* de l'OPC UA fournit un moyen normalisé à des *Serveurs* pour présenter des *Objets* à des *Clients*. Les *Objets* au sens de l'OPC UA se composent d'autres *Objets*, de *Variables* et de *Méthodes*. L'OPC UA permet aussi l'expression de relations à d'autres *Objets*.

L'ensemble des *Objets* et des relations connexes qu'un *Serveur* de l'OPC UA met à la disposition des *Clients* est appelé son *AddressSpace* (espace d'adresses). Les éléments du modèle d'*Objets* OPC UA sont représentés dans l'*AddressSpace* sous la forme d'un ensemble de *Nodes* (nœuds) décrits par des *Attributs* et interconnectés par des *Références*. L'OPC UA définit huit classes des *Nœuds* pour représenter les composants de l'*AddressSpace*. Les classes sont *Object* (objet), *Variable, Method* (méthode), *ObjectType* (type d'objet), *VariableType* (type de variable), *DataType* (type de données), *ReferenceType* (type de référence) et *View* (vue). Chaque *NodeClass* (classe de nœuds) a un ensemble défini d'*Attributs*.

La présente norme utilise presque toutes les *NodeClass* de l'OPC UA.

Les *Objets* sont utilisés pour représenter les entités du monde réel telles que les *Appareils* et les Réseaux (de communication) ainsi que les entités logicielles telles que les *Blocs*. Un *Objet* est associé à un *ObjectType* correspondant qui fournit des définitions pour l'*Objet* en question.

Les *Variables* sont utilisées pour représenter des valeurs. Deux catégories de *Variables* sont définies, à savoir les *Properties* (propriétés) et les *DataVariables* (variables de données).

Les *Propriétés* sont des caractéristiques définies par un *Serveur* pour les *Objets*, les *DataVariables* et autres *Nœuds*. Les *Propriétés* ne sont pas autorisées à avoir des *Propriétés* qui sont définies pour elles. Exemples de *Propriétés* d'*Objets*: numéro de série d'appareil et marqueur de bloc.

Les *DataVariables* représentent le contenu d'un *Objet*. Les *DataVariables* peuvent avoir des composants *DataVariables*. Elles sont typiquement utilisées par les *Serveurs* pour exposer des éléments individuels de matrices et de structures. La présente norme utilise les *DataVariables* pour représenter les *Paramètres* des *Blocs* et des *Appareils*.

4.2 Conventions utilisées dans le présent document

4.2.1 Conventions pour les descriptions de Nœuds

Les définitions des *Nœuds* sont spécifiées en utilisant des tableaux (voir Tableau 2).

Tableau 2 – Tableau de définitions des types

Attribut	**Valeur**				
Nom d'attribut	Valeur d'attribut. S'il s'agit d'un Attribut facultatif qui n'est pas mis, le signe “--“ sera utilisé.				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Nom du *ReferenceType* (Type de référence)	*NodeClass* (Classe de nœuds) du Target*Node* (nœud cible).	*BrowseName* (Nom d'exploration) du *Nœud* cible. Si la *Référence* doit être instanciée par le serveur, la valeur du Nom d'exploration du Nœud cible est “--“.	Les *Attributs* du *Nœud* référencé, applicables seulement pour les *Variables* et les *Objets*.		*ModellingRule* (Règle de modélisation) référencée de l'*Objet* référencé.
NOTE Notes renvoyant aux notes de bas du contenu du tableau.					

Les *Attributs* sont définis en fournissant le nom d'*Attribut* et une valeur, ou une description de la valeur.

Les *Références* sont définies en fournissant le nom du *ReferenceType* (type de référence), le *Nom d'exploration* du *Nœud cible* et sa *Classe de nœuds*.

- Si le *Nœud cible* est un composant du *Nœud* défini dans le tableau, les *Attributs* du *Nœud* composé sont définis dans la même ligne du tableau.
- Le *DataType* n'est spécifié que pour les *Variables*; “[<nombre>]” indique une matrice à une dimension. Pour les matrices à plusieurs dimensions, l'expression est répétée pour chaque dimension (par exemple: [2][3] pour une matrice à deux dimensions). Pour toutes les matrices, l'attribut *ArrayDimensions* (Dimensions de la matrice) est établi comme identifié par les valeurs de <nombre>. Si aucun <nombre> n'est établi, la dimension correspondante est mise à 0, indiquant une taille inconnue. Si aucun nombre n'est fourni du tout, l'attribut *ArrayDimensions* peut être omis. Si des crochets ne sont pas fournis, cela identifie un *DataType* scalaire et l'attribut *ValueRank* (rang de la valeur) est mis à la valeur correspondante (voir IEC 62541-3). En outre, l'attribut *ArrayDimensions* est mis à "*null*" ou est omis. Si elle peut être *Any* (Tout) ou *ScalarOrOneDimension* (scalaire ou à une dimension), la valeur est placée dans “{<valeur>}”, soit “{*Any*}” ou “{*ScalarOrOneDimension*}”. Dans ce cas, *ValueRank* est mis à la valeur correspondante (voir IEC 62541-3) et *ArrayDimensions* est mis à "null" ou est omis. Des exemples sont donnés dans le Tableau 3.

Tableau 3 – Exemples de DataTypes

Notation	**Data-Type**	**Valeur du rang**	**Dimensions de la matrice**	**Description**
Int32	Int32	-1	omis ou NULL	Un scalaire Int32
Int32[]	Int32	1	omis ou {0}	Matrice à une dimension d'entiers Int32 de taille inconnue
Int32[][]	Int32	2	omis ou {0,0}	Matrice à deux dimensions d'entiers Int32 avec des tailles inconnues pour les deux dimensions
Int32[3][]	Int32	2	{3,0}	Matrice à deux dimensions d'entiers Int32 avec une taille égale à 3 pour la première dimension et une taille inconnue pour la seconde dimension
Int32[5][3]	Int32	2	{5,3}	Matrice à deux dimensions d'entiers Int32 avec une taille égale à 5 pour la première dimension et une taille égale à 3 pour la seconde dimension
Int32{Any}	Int32	-2	omis ou NULL	Entier Int32 où on ne sait pas s'il s'agit d'un scalaire ou d'une matrice ayant un nombre quelconque de dimensions
Int32{ScalarOrOneDimension}	Int32	-3	omis ou NULL	Un entier Int32 où il s'agit soit d'une matrice à une dimension, soit d'un scalaire

- L'attribut TypeDefinition (définition de type) est spécifié pour les *Objets* et les *Variables*.

- La colonne TypeDefinition (définition de type) spécifie un *NodeId* (identificateur de nœud) d'un *TypeDefinitionNode* (Nœud de définition de type), c'est-à-dire que le *Nœud* spécifié pointe avec une *Référence HasTypeDefinition* (Possède une définition de type) vers le *TypeDefinitionNode* correspondant. Le nom symbolique du *NodeId* est utilisé dans le tableau.
- L'attribut *ModellingRule* (Règle de modélisation) du composant référencé est fourni en spécifiant le nom symbolique de la règle dans la colonne *ModellingRule*. Dans l'*AddressSpace* (Espace adresse), le *Nœud* doit utiliser une *Référence HasModellingRule* (Possède une règle de modélisation) pour pointer vers l'*Objet ModellingRule* correspondant.

Si le *NodeId* d'un *DataType* est fourni, le nom symbolique du *Nœud* représentant le *DataType* doit être utilisé.

Si aucun composant n'est fourni, les colonnes *DataType, TypeDefinition* et *ModellingRule* peuvent être omises et seule une colonne *Comment* (Commentaires) est introduite pour pointer vers la définition du *Nœud*.

Les composants des *Nœuds* peuvent être complexes, c'est-à-dire contenir eux-mêmes des composants. Les attributs *TypeDefinition*, *NodeClass*, *DataType* et *ModellingRule* peuvent être dérivés des définitions de types et le nom symbolique peut être créé tel que défini en 4.2.2.1. Par conséquent, ceux qui contiennent des composants ne sont pas spécifiés de façon explicite; ils sont plutôt spécifiés de façon implicite par les définitions de types.

4.2.2 NodeId et BrowseName

4.2.2.1 NodeId

Les *NodeIds* de tous les *Nœuds* décrits dans le présent document sont seulement des noms symboliques. L'Annexe A définit les *NodeIds* réels.

Le nom symbolique de chaque *nœud* défini dans le présent document est son *BrowseName*, ou, lorsqu'il fait partie d'un autre *Nœud*, le *BrowseName* de cet autre *nœud*, un ".", et son propre *BrowseName*. Dans ce cas, "partie de" signifie que le tout a une *Référence HasProperty* (Possède une propriété) ou *HasComponent* (Possède un composant) vers sa partie. Sachant que tous les *Nœuds* qui ne font pas partie d'un autre *Nœud* ont un nom unique dans le présent document, le nom symbolique est unique.

L'espace de noms pour la présente spécification est défini dans l'Annexe A. Le *NamespaceIndex* (Indice d'espace de noms) pour tous les *NodeIds* et *BrowseNames* définis dans la présente spécification est spécifique à un serveur et dépend de la position de l'URI[1] (Identificateur uniforme de ressource) de l'espace de noms dans le tableau d'espaces de noms du serveur.

4.2.2.2 BrowseNames

La partie texte des *BrowseNames* pour tous les *Nœuds* définis dans la présente norme est spécifiée dans les tableaux définissant les *Nœuds*. Le *NamespaceIndex* (Indice d'espace de noms) pour tous les *BrowseNames* définis dans la présente norme est spécifique à un serveur et dépend de la position de l'URI de l'espace de noms dans le tableau d'espaces de noms du serveur.

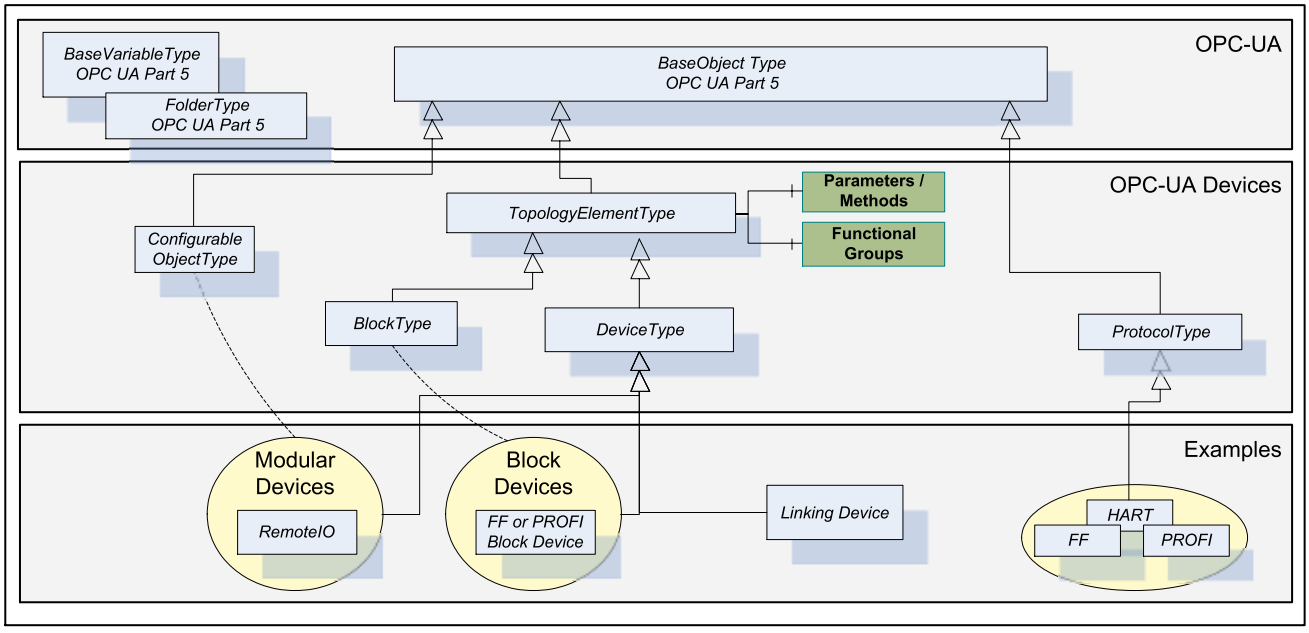
---

1 URI = *Uniform Resource Identifier.*

5 Modèle d'Appareil

5.1 Généralités

La Figure 1 illustre les *ObjectTypes* du modèle d'*Appareil* de base et leur relation. Le dessin ne vise pas à être complet. Par souci de simplicité, seuls quelques composants et relations sont saisis afin de donner une idée grossière de la structure globale.



Anglais	Français
BaseVariableType OPC UA Part 5	BaseVariableType OPC UA Partie 5
FolderType OPC UA Part 5	FolderType OPC UA Partie 5
BaseObject Type OPC UA Part 5	BaseObject Type OPC UA Partie 5
OPC-UA	OPC-UA
Configurable ObjectType	ObjectType configurable
TopologyElementType	TopologyElementType
Parameters /Methods	Paramètres /Méthodes
Functional Groups	Groupes fonctionnels
BlockType	BlockType
DeviceType	DeviceType
ProtocolType	ProtocolType
OPC-UA Devices	Appareils OPC-UA
Modular Devices	Appareils modulaires
RemoteIO	E/S à distance
Block Devices	Appareils à blocs
FF or PROFI Block Device	Appareil à blocs FF ou PROFI
Linking Device	Appareil de liaison
HART FF PROFI	HART FF PROFI
Examples	Exemples

Figure 1 – Vue d'ensemble *des* modèles d'Appareils

Les rectangles dans le dessin montrent les *ObjectTypes* utilisés dans la présente norme, ainsi qu'un certain nombre d'éléments issus d'autres normes qui aident à comprendre certaines décisions de modélisation. Le rectangle gris supérieur montre les *ObjectTypes* de base de l'OPC UA à partir desquels sont dérivés le *TopologyElementType* (Type d'élément de topologie) et le *ProtocolType* (Type de protocole). Le rectangle gris au deuxième niveau montre les principaux *ObjectTypes* qu'introduit la présente norme. Les composants de ces *ObjectTypes* ne sont illustrés que de façon abstraite dans l'image globale.

Le rectangle gris au troisième niveau montre des exemples du monde réel tels qu'ils sont utilisés dans les produits et les usines. En général, ces sous-types sont définis par d'autres organisations.

Le *TopologyElementType* est l'*ObjectType* de base pour les éléments de la topologie d'appareils. Il introduit des *Paramètres* et des *Méthodes*. La présente spécification définit aussi un concept de groupage fonctionnel pour fournir d'autres points de vue.

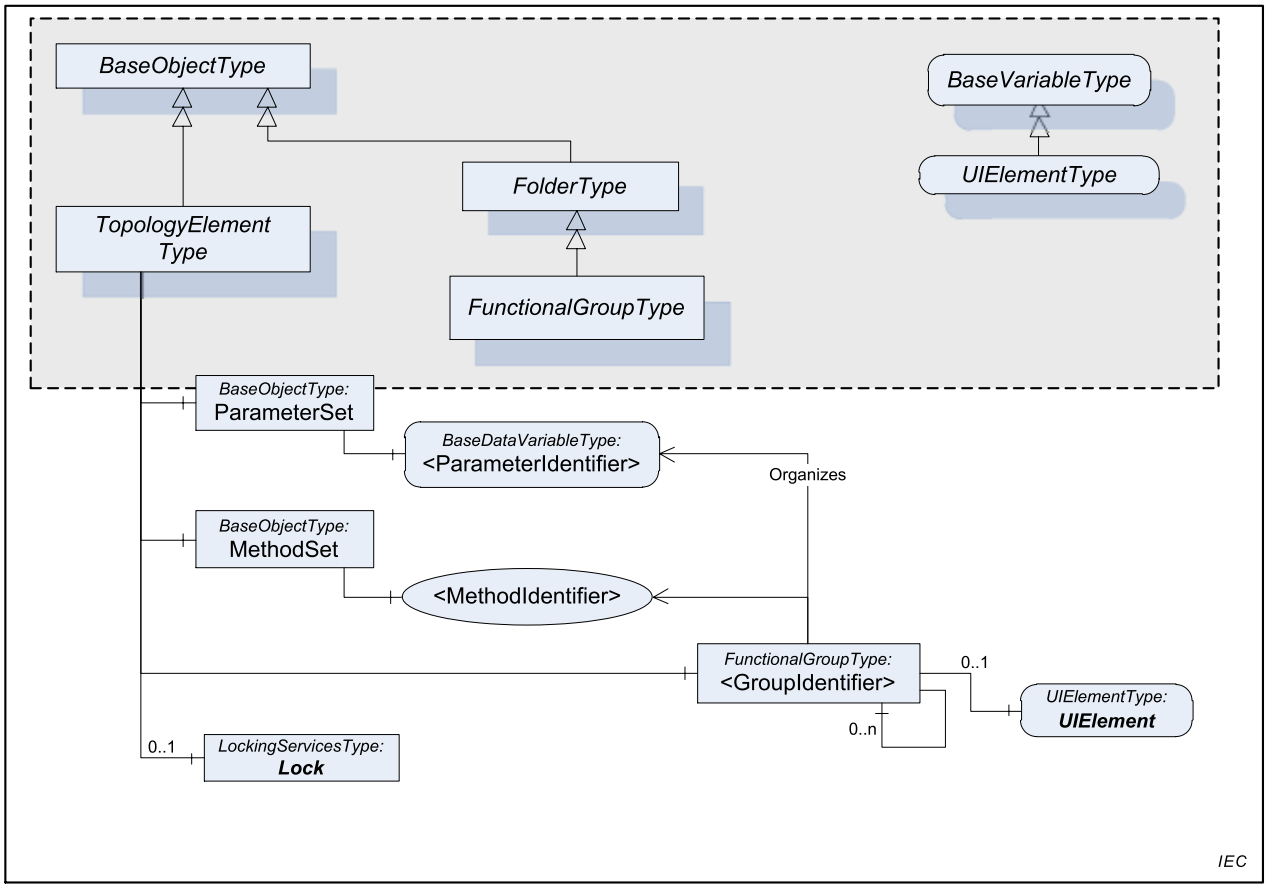
Le *type d'objet DeviceType* fournit une définition de types générale pour n'importe quel *Appareil*. Les *Appareils* – en plus de *Paramètres* et *Méthodes* – peuvent prendre en charge des sous-appareils et peuvent prendre en charge des *Blocks*. Les *Blocks* sont typiquement utilisés comme des moyens d'organiser la fonctionnalité au sein d'un *Appareil*. Les types spécifiques de *Blocks* sont spécifiés par diverses fondations de communication de terrain.

Un *type d'objet ProtocolType* représente un protocole de communication/*FieldBus* (Bus de terrain) spécifique qui est mis en œuvre par un certain *TopologyElement* (Élément de topologie). Des exemples sont *FFBusType*, *HARTBusType* ou PROFIBUS/PROFINETType.

Le *ConfigurableObjectType* (Type d'objet configurable) est utilisé comme un moyen général de création d'unités topologiques modulaires. Si besoin est, une instance de type est ajoutée à l'objet en tête de l'unité modulaire. Des *Appareils* modulaires, par exemple, utilisent cet *ObjectType* pour organiser leurs *Modules*. Les *Appareils* orientés bloc l'utilisent pour exposer et organiser leurs *Blocks* (Blocs).

5.2 TopologyElementType

Cet *ObjectType* définit les composants d'informations de base pour tous les éléments configurables dans une topologie d'appareils. Il introduit *ParameterSet* (Ensemble de paramètres), *MethodSet* (Ensemble de méthodes), et *FunctionalGroups* (Groupes fonctionnels). La Figure 2 montre le *TopologyElementType,* formellement défini dans le Tableau 4.



Anglais	Français
BaseObjectType: ParameterSet	Type d'objet de base: Ensemble de paramètres
BaseObjectType	Type d'objet de base
FolderType	Type de dossier
FunctionalGroupType: <GroupIdentifier>	Type de groupe fonctionnel: <Identificateur de groupe>
TopologyElement Type	Type d'élément de topologie
FunctionalGroupType	Type de groupe fonctionnel
BaseDataVariableType: <ParameterIdentifier>	Type de variable de données de base: <Identificateur de paramètre>
<MethodIdentifier>	<Identificateur de méthode>
BaseObjectType: MethodSet	Type d'objet de base: Ensemble de méthodes
Organizes	Organise
UIElementType: UIElement	Type d'élément d'interface utilisateur: élément d'interface utilisateur
UIElementType	Élément d'interface utilisateur
BaseVariableType	Type de variable de base
LockingServicesType: Lock	Type de services de verrouillage: Verrou

Figure 2 – Composants du TopologyElementType

Tous les éléments dans une topologie peuvent avoir des *Paramètres* et des *Méthodes*. Si un tel élément a des *Paramètres*, ceux-ci sont conservés dans un *Objet* appelé “ParameterSet” (Ensemble de paramètres) sous la forme d'une liste plate de *Paramètres*. S'il a des *Méthodes*, celles-ci sont conservées de la même manière dans un *Objet* appelé “MethodSet” (Ensemble

de méthodes). Des *FunctionalGroups* (Groupes fonctionnels) peuvent être utilisés pour organiser les *Paramètres* et les *Méthodes* pour refléter la structure du *TopologyElement* (Élément de topologie). Le même *Paramètre* ou la même *Méthode* pourrait être référencé(e) à partir de plus d'un *FunctionalGroup*. Pour la fonctionnalité représentée, chaque groupe peut référencer une interface utilisateur facultative. Les *FunctionalGroups* sont spécifiés en 5.3.

- Règle pour toutes les *Méthodes* organisées en *FunctionalGroups*:

  Les *Méthodes* étant des composants de l'objet *MethodSet*, les *Clients* doivent spécifier le *NodeId* de l'instance de *MethodSet* dans l'argument *ObjectId* du *Service* Call (Appel).

Les *Paramètres* sont modélisés avec des nœuds *DataVariable* de l'OPC UA. Un *Paramètre* peut être une instance de *BaseDataVariableType* (défini dans l'IEC 62541-5) ou tout sous-type défini par l'OPC UA ou autres organisations de normalisation. Certains des *VariableTypes* les plus courants (*DataItemType, AnalogItemType,* et *DiscreteItemTypes*) sont définis dans l'IEC 62541-8. Dans la présente norme, le *Paramètre* est utilisé d'une manière générale – quel que soit le *VariableType*.

Le *type d'objet TopologyElement* est formellement défini dans le Tableau 4

Tableau 4 – Définition de TopologyElementType

Attribut	**Valeur**				
Nom d'exploration	TopologyElementType				
IsAbstract	Vrai				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Sous-type du *BaseObjectType* défini dans l'IEC 62541-5					
HasSubtype	ObjectType	DeviceType	Défini en 5.3		
HasSubtype	ObjectType	BlockType	Défini en 5.10		
HasComponent	Objet	ParameterSet		BaseObjectType	Facultative
HasComponent	Objet	MethodSet		BaseObjectType	Facultative
HasComponent	Objet	<GroupIdentifier>		FunctionalGroupType	OptionalPlaceHolder (Réceptacle facultatif)
HasComponent	Objet	Identification		FunctionalGroupType	Facultative
HasComponent	Objet	Lock		LockingServicesType	Facultative

Le *TopologyElementType* est abstrait. Il n'y aura pas d'instances d'un *TopologyElementType* en soi, mais il y aura des instances de sous-types de ce type. Dans la présente norme, le terme *TopologyElement* renvoie génériquement à une instance de n'importe quel *ObjectType* dérivé du type *TopologyElementType*.

ParameterSet rassemble les références à tous les *Paramètres* qui sont exposés au *Client*. Le *ParameterSet* est obligatoire si des *Paramètres* existent pour un *TopologyElement*. L'*Objet* "*ParameterSet*" est formellement défini dans le Tableau 5.

Tableau 5 – Définition de ParameterSet

Attribut	**Valeur**			
Nom d'exploration	ParameterSet			
Références	**Classe de nœuds**	**Nom d'exploration**	**Définition de type**	**Règle de modélisation**
HasTypeDefinition	ObjectType	BaseObjectType		
HasComponent	Variable	<ParameterIdentifier>	BaseDataVariableType	MandatoryPlaceholder (Réceptacle obligatoire)

MethodSet rassemble les références à toutes les *Méthodes* qui sont exposées au *Client*. L'*Objet* "*MethodSet*" est formellement défini dans le Tableau 6.

Tableau 6 – Définition de MethodSet

Attribut	**Valeur**			
Nom d'exploration	MethodSet			
Références	**Classe de nœuds**	**Nom d'exploration**	**Définition de type**	**Règle de modélisation**
HasTypeDefinition	ObjectType	BaseObjectType		
HasComponent	Méthode	<MethodIdentifier>		MandatoryPlaceholder

Des *TopologyElements* peuvent avoir un nombre arbitraire de *FunctionalGroups* pour organiser les *Paramètres* et les *Méthodes* (voir 5.3). Un *FunctionalGroup* spécial appelé **Identification** doit être utilisé pour organiser les *Paramètre*s pour l'identification de ce *TopologyElement* (voir 5.4).

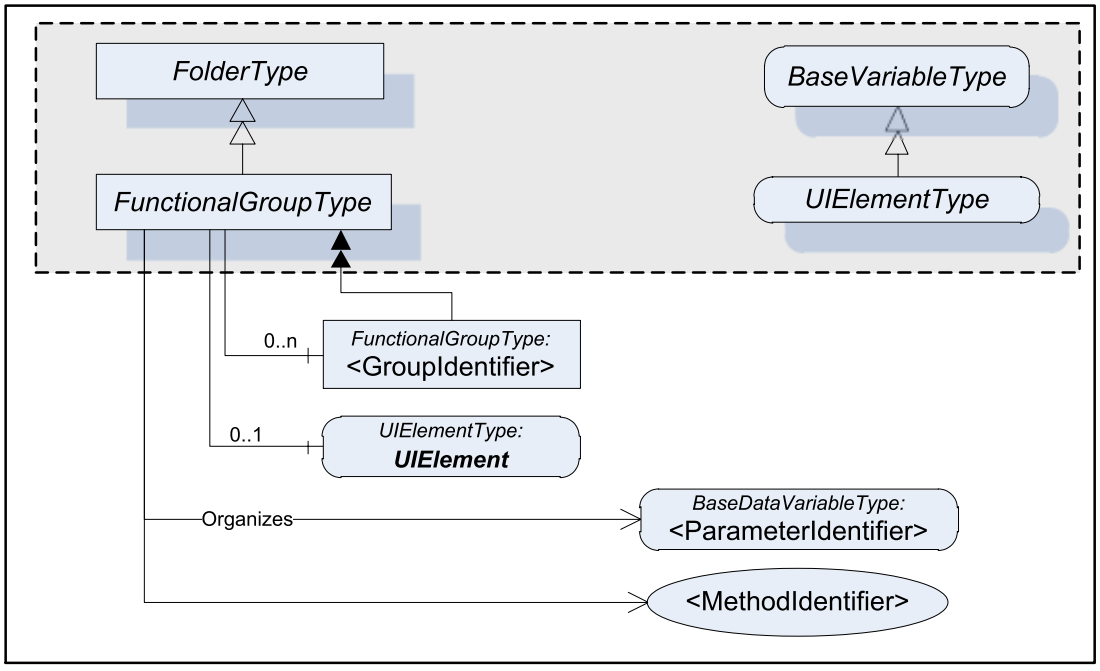
En plus des éléments décrits en 5.2, les TopologyElements peuvent aussi prendre en charge des *LockingServices* (définis en 8.3.3).

5.3 FunctionalGroupType

Ce sous-type du *FolderType* de l'OPC UA est utilisé pour organiser les *Paramètres* et les *Méthodes* issus de l'ensemble complet (*ParameterSet*, *MethodSet*) en fonction de leur application (par exemple: maintenance, diagnostic). Il peut aussi faire référencer une *Propriété* du *TopologyElement* auquel il appartient. Il peut contenir un élément d'interface utilisateur. La même *Propriété*, le même *Paramètre* ou la même *Méthode* pourrait être référencé(e) à partir de plus d'un *FunctionalGroup*.

Des *FunctionalGroups* peuvent être imbriqués.

La Figure 3 montre les composants du *FunctionalGroupType,* formellement défini dans le Tableau 7.



Anglais	Français
FolderType	Type de dossier
FunctionalGroupType	Type de groupe fonctionnel
FunctionalGroupType: <GroupIdentifier>	Type de groupe fonctionnel: <Identificateur de groupe>
UIElementType: UIElement	Type d'élément d'interface utilisateur: élément d'interface utilisateur
Organizes	Organise
BaseDataVariableType: <ParameterIdentifier>	Type de variable de données de base: <Identificateur de paramètre>
<MethodIdentifier>	<Identificateur de méthode>
BaseVariableType	Type de variable de base
UIElementType	Élément d'interface utilisateur

Figure 3 – FunctionalGroupType

Tableau 7 – Définition de FunctionalGroupType

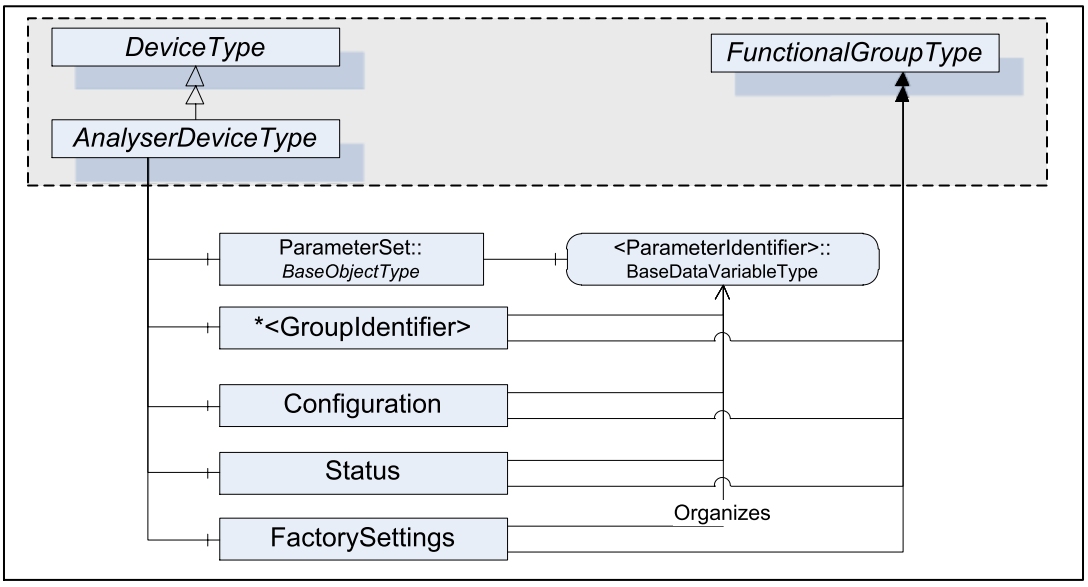
Attribut	**Valeur**				
Nom d'exploration	FunctionalGroupType				
IsAbstract	Faux				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Sous-type du *FolderType* défini dans l'IEC 62541-5					
HasComponent	Objet	<GroupIdentifier>		FunctionalGroupType	OptionalPlaceHolder
Organise	Variable	<ParameterIdentifier>	BaseDataType	BaseDataVariableType	OptionalPlaceHolder
Organise	Méthode	<MethodIdentifier>			OptionalPlaceHolder
HasComponent	Variable	UIElement	BaseDataType	UIElementType	Facultative

Les *Références* "Organizes" (Organise) peuvent être présentes seulement au niveau de l'instance, pas au type. Selon l'état courant du *TopologyElement*, le *Serveur* peut décider de cacher ou de révéler certains *FunctionalGroup*s ou (partie de) leurs *Références*. Si un *FunctionalGroup* peut être caché sur une instance, la définition de type (*TypeDefinition*) doit utiliser une *ModellingRule* appropriée comme “Facultative”.

L'*Attribut Description* est utilisé pour décrire un but prévu du *FunctionalGroup*.

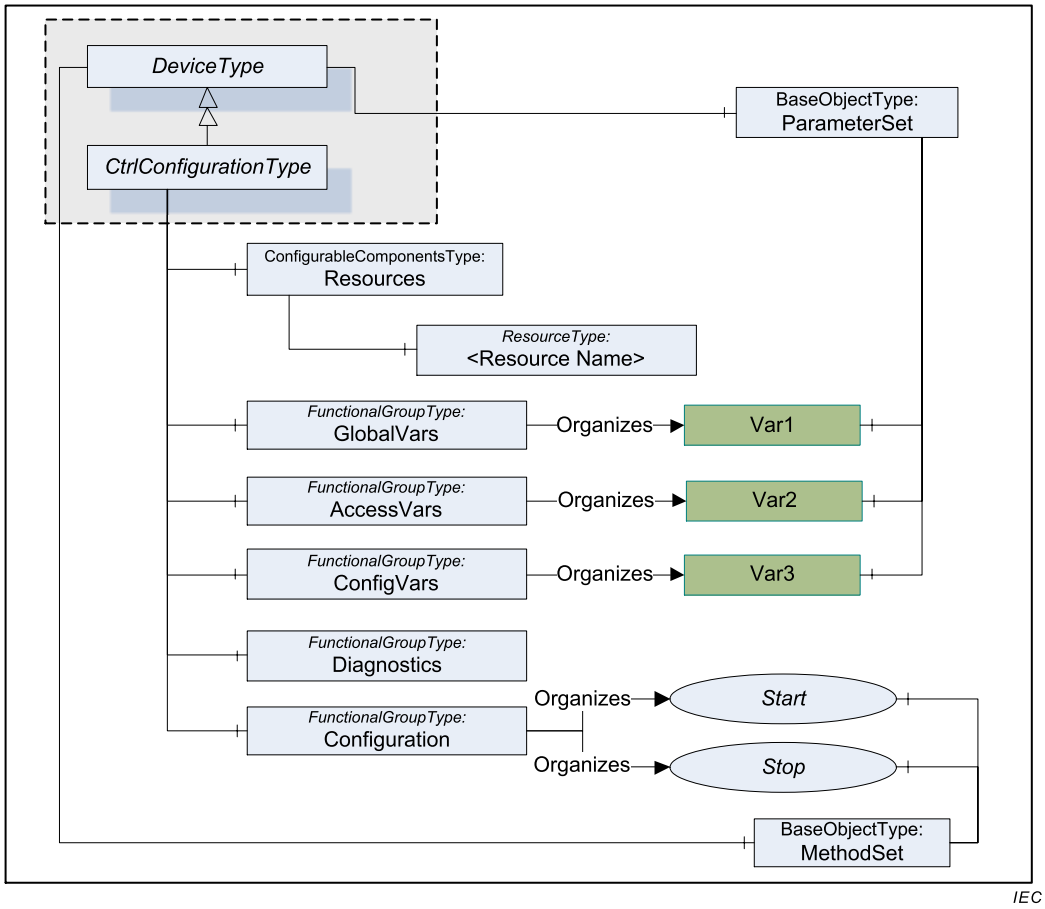
UIElement est l'élément d'interface utilisateur pour ce *FunctionalGroup*. Voir en 5.5 la définition des *UIElements*.

Les exemples à la Figure 4 et à la Figure 5 illustrent l'utilisation des *FunctionalGroups:*



Anglais	Français
ParameterSet::BaseObjectType	ParameterSet::BaseObjectType
DeviceType	DeviceType
Configuration	Configuration
AnalyserDeviceType	AnalyserDeviceType
FunctionalGroupType	FunctionalGroupType
Status	Statut
FactorySettings	FactorySettings
Organizes	Organise
<ParameterIdentifier>::BaseDataVariableType	<Identificateur de paramètre>::BaseDataVariableType
*<GroupIdentifier>	*<Identificateur de groupe>

Figure 4 – Utilisation d'appareil analyseur pour des FunctionalGroups (UA Companion ADI)



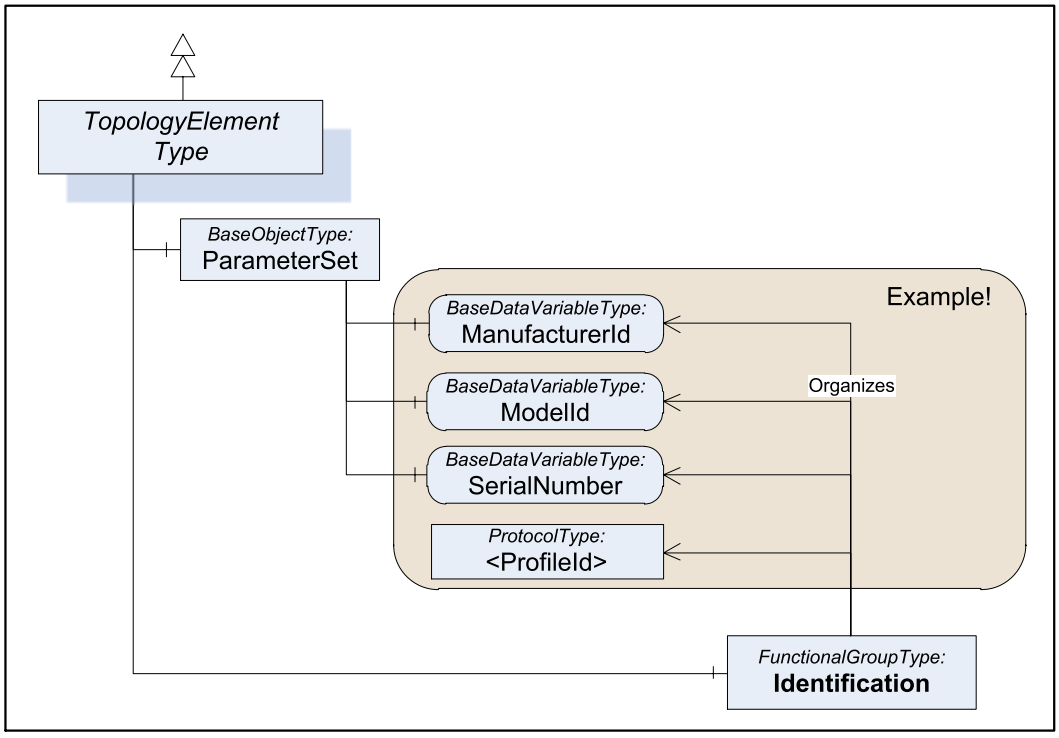
Anglais	Français
DeviceType	DeviceType
CtrlConfigurationType	CtrlConfigurationType
BaseObjectType:ParameterSet	BaseObjectType:ParameterSet
ConfigurableComponentsType:Resources	ConfigurableComponentsType:Resources
ResourceType:<Resource Name>	ResourceType:<Nom de ressource>
FunctionalGroupType:GlobalVars	FunctionalGroupType:GlobalVars
BaseObjectType:MethodSet	BaseObjectType:MethodSet
Organizes	Organise
Var1	Var1
FunctionalGroupType:AccessVars	FunctionalGroupType:AccessVars
Organizes	Organise
Var2	Var2
FunctionalGroupType:ConfigVars	FunctionalGroupType:ConfigVars
Organizes	Organise
Var3	Var3
FunctionalGroupType:Diagnostics	FunctionalGroupType:Diagnostics
FunctionalGroupType:Configuration	FunctionalGroupType:Configuration
Organizes	Organise
Start	Démarrer
Organizes	Organise
Stop	Arrêter

Figure 5 – Utilisation de PLCopen pour des FunctionalGroups (UA Companion PLCopen)

5.4 FunctionalGroup Identification

Les *Paramètres* pour l'identification d'un *TopologyElement* doivent être organisés dans un *FunctionalGroup* appelé **Identification**. À titre d'exemple, des *Clients* peuvent utiliser les valeurs de ces *Paramètres* pour trouver certains éléments ou détecter des discordances entre les données de configuration et l'élément actuellement connecté. La Figure 6 illustre le *FunctionalGroup* **Identification** par un exemple.

NOTE Les organismes de normalisation sont censés définir le contenu d'Identification pour des protocoles de bus spécifiques.



Anglais	Français
TopologyElementType	TopologyElementType
BaseObjectType:ParameterSet	BaseObjectType:ParameterSet
BaseDataVariableType:ManufacturerId	BaseDataVariableType:ManufacturerId
BaseDataVariableType:ModelId	BaseDataVariableType:ModelId
BaseDataVariableType:SerialNumber	BaseDataVariableType:SerialNumber
ProtocolType:ProfileId	ProtocolType:ProfileId
FunctionalGroupType:Identification	FunctionalGroupType:Identification
Organizes	Organise
Example	Exemple

Figure 6 – Exemple d'un FunctionalGroup Identification

5.5 Type UIElement

Des *Serveurs* peuvent exposer des *UIElements* fournissant des interfaces utilisateur dans le contexte de leur conteneur *FunctionalGroup*. Les *Clients* peuvent charger une telle interface utilisateur et l'afficher du côté client. La hiérarchie des *FunctionalGroups* représente l'arborescence des éléments d'interface utilisateur.

L'*UIElementType* est abstrait et sert principalement de filtre pour l'exploration d'un *FunctionalGroup*. Seuls des sous-types peuvent être utilisés pour des instances. Aucun

UIElement concret n'est défini dans la présente spécification. La FDI[2] (Intégration des appareils de terrain, voir l'IEC 62769) spécifie deux sous-types concrets:

- UID[3] (Descriptions d'interface utilisateur), éléments descriptifs d'interface utilisateur, et
- UIPs[4] (Plugiciels d'interface utilisateur), éléments programmés d'interface utilisateur.

L'*UIElementType* est spécifié dans le Tableau 8.

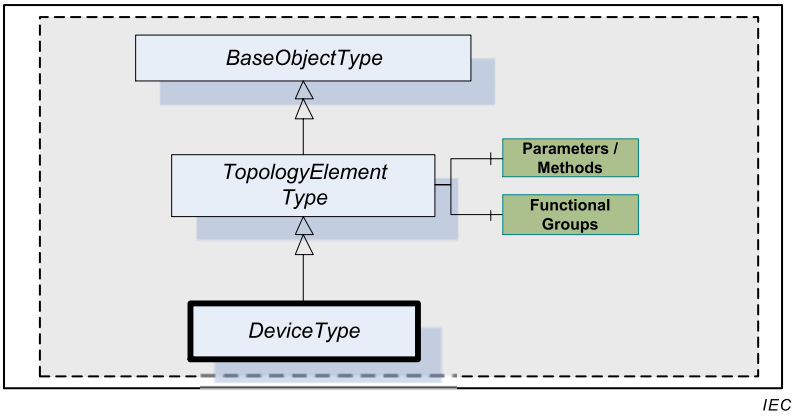
Tableau 8 – Définition de l'UIElementType

Attribut	**Valeur**				
Nom d'exploration	UIElementType				
IsAbstract	Vrai				
Type de données	BaseDataType				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Sous-type de BaseVariableType défini dans l'IEC 62541-5					

L'attribut *Value* (Valeur) de l'*UIElement* contient l'élément d'interface utilisateur. Les sous-types doivent définir le *DataType* (par exemple: *XmlElement* ou *ByteString*).

5.6 DeviceType

Cet *ObjectType* définit la structure d'un *Device* (Appareil). La Figure 7 montre le *DeviceType,* formellement défini dans le Tableau 9.



Anglais	**Français**
Parameters / Methods	Paramètres / Méthodes
Functional Groups	Groupes fonctionnels

Figure 7 – DeviceType

2 FDI = *Field Device Integration*.

3 UID = *UI Description*.

4 UIPs = *UI Plug-ins*.

Tableau 9 – Définition de DeviceType

Attribut	**Valeur**				
Nom d'exploration	DeviceType				
IsAbstract	Vrai				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Hériter des *Propriétés* du *TopologyElementType* défini en 5.2					
HasProperty	Variable	SerialNumber	String	PropertyType	Obligatoire
HasProperty	Variable	RevisionCounter	Int32	PropertyType	Obligatoire
HasProperty	Variable	Manufacturer (fabricant)	LocalizedText	PropertyType	Obligatoire
HasProperty	Variable	Modèle	LocalizedText	PropertyType	Obligatoire
HasProperty	Variable	DeviceManual	String	PropertyType	Obligatoire
HasProperty	Variable	DeviceRevision	String	PropertyType	Obligatoire
HasProperty	Variable	SoftwareRevision	String	PropertyType	Obligatoire
HasProperty	Variable	HardwareRevision	String	PropertyType	Obligatoire
HasProperty	Variable	DeviceClass	String	PropertyType	Facultative
HasComponent	Variable	DeviceHealth	DeviceHealth	BaseDataVariableType	Facultative
HasComponent	Objet	DeviceTypeImage		FolderType	Facultative
HasComponent	Objet	Documentation		FolderType	Facultative
HasComponent	Objet	ProtocolSupport		FolderType	Facultative
HasComponent	Objet	ImageSet		FolderType	Facultative
HasComponent	Objet	<CPIdentifier>		ConnectionPointType	OptionalPlaceHolder

Le *type d'objet DeviceType* est abstrait. Il n'y aura pas d'instances d'un *DeviceType* en soi, mais il y aura des instances de sous-types de ce type. Dans la présente norme, le terme *Device* (Appareil) renvoie génériquement à une instance de n'importe quel *ObjectType* dérivé du type *DeviceType*.

Les *Appareils* peuvent avoir des *Paramètres*, des *Méthodes*, et des *FunctionalGroup*s (groupes fonctionnels) tels que définis pour le *TopologyElementType*.

Les *Propriétés* ci-après fournissent un moyen permettant à un *Client* d'obtenir des informations communes relatives à un *Appareil*. Il ne s'agit pas nécessairement d'un remplacement des informations apparaissant dans le *ParameterSet* et/ou dans les *FunctionalGroups* de l'*Appareil*. Noter que la présente norme ne fait aucune autre hypothèse que les hypothèses ci-après qui sont relatives à la sémantique. Les organisations de normalisation peuvent spécifier le contenu d'une manière plus approfondie.

Bien que la plupart des *Propriétés* soient obligatoires pour toutes les instances de *DeviceType*, certaines peuvent ne pas être prises en charge par certains types d'appareils. Dans ce cas, les vendeurs doivent fournir les valeurs par défaut suivantes:

- *Propriétés* avec le *type de données* String: **chaîne vide**
- *Propriétés* avec le *type de données* LocalizedText: **champ textuel vide**
- *Propriété RevisionCounter*: **- 1**

La *Propriété SerialNumber* (Numéro de série) est un numéro de production unique du fabricant du *Device* (Appareil). Elle est souvent estampillée sur l'extérieur de l'*Appareil* et peut être utilisée aux fins de traçabilité et de garantie.

La *Propriété RevisionCounter* (Compteur de révisions) est un compteur incrémentiel indiquant le nombre de fois que les données statiques dans l'*Appareil* ont été modifiées.

La *Propriété Manufacturer* (Fabricant) fournit le nom de l'entreprise qui a fabriqué l'*Appareil*.

La *Propriété Model* (Modèle) fournit le nom du modèle de l'*Appareil*.

La *Propriété DeviceManual* (Manuel d'appareil) permet de spécifier une adresse du manuel d'utilisateur pour l'*Appareil*. Il peut s'agir du nom de chemin dans le système de fichiers ou d'un URL, localisateur uniforme de ressource (adresse web).

La *propriété DeviceRevision* (Révision d'appareil) fournit le niveau de révision globale de l'*Appareil*.

La *propriété SoftwareRevision* (Révision logicielle) fournit le niveau de révision du logiciel/micrologiciel (firmware) de l'*Appareil*.

La *propriété HardwareRevision* (Révision matérielle) fournit le niveau de révision du matériel de l'*Appareil*.

La *propriété DeviceClass* (Classe d'appareils) indique le domaine ou le but dans lequel un certain *Appareil* est utilisé. Des exemples sont “ProgrammableController” (automate programmable), “RemoteIO” (E/S à distance), et “TemperatureSensor” (Capteur de température). La présente norme ne prédéfinit aucun nom de *DeviceClass*. Des normes plus spécifiques qui utilisent le *DeviceType* introduiront vraisemblablement ces classifications (par exemple IEC 62769, UA Companion PLCopen, ou UA Companion ADI).

L'attribut *Description* de l'*Objet Appareil* fournit des informations qui servent à davantage identifier, gérer, localiser et/ou expliquer l'appareil dont le contenu est défini par l'utilisateur.

D'autres organisations peuvent spécifier une sémantique complémentaire pour le contenu de ces *Propriétés*.

Les *Paramètres* comme *ManufacturerId* (Identificateur numérique de l'entreprise), *ModelId* et *SubModelId* (Identificateurs numériques du modèle) ne sont pas fournis comme *Propriétés*. Au contraire, ces *Paramètres* doivent être inclus dans le *FunctionalGroup* **Identification** défini en 5.4.

La propriété *DeviceHealth* (Santé d'appareil) indique le statut de l'appareil tel que défini dans la NAMUR Recommendation NE107. Les *Clients* peuvent lire ou surveiller cette *Variable* pour déterminer l'état de l'appareil.

Le *DeviceHealth DataType* est une énumération qui définit l'état de l'appareil. Ses valeurs sont définies dans le Tableau 10.

Tableau 10 – Valeurs de DeviceHealth

Valeur	Description
NORMAL_0	L'appareil fonctionne normalement.
FAILURE_1	Mauvais fonctionnement de l'appareil ou d'un de ses périphériques. Cause typiquement interne à l'appareil ou liée à un processus.
CHECK_FUNCTION_2	Des vérifications fonctionnelles sont actuellement effectuées. Exemples: Changement de configuration, fonctionnement local, valeur de substitution saisie.
OFF_SPEC_3	"Off-spec" (hors spécification) signifie que l'appareil est exploité hors de sa plage spécifiée (par exemple: plage de mesure ou de température) ou que des diagnostics internes indiquent des écarts par rapport à des valeurs mesurées ou à des valeurs de consigne en raison de problèmes internes dans les caractéristiques de l'appareil ou du processus.
MAINTENANCE_REQUIRED_4	Bien que le signal de sortie soit valide, la réserve d'usure est presque épuisée ou une fonction sera bientôt limitée en raison des conditions de fonctionnement, par exemple accumulation de dépôts.

Les *ConnectionPoints* (Points de connexion, voir 6.3) représentent l'interface (carte d'interface) d'un *Appareil* à un *Réseau*. Un *Appareil* peut avoir plusieurs *ConnectionPoints* s'il prend en charge plusieurs protocoles et/ou plusieurs *Profils de communication*.

5.7 Informations de prise en charge d'appareil

5.7.1 Généralités

Chaque *DeviceType* peut avoir un ensemble de données complémentaires. Il s'agit principalement d'images, de documents ou d'images spécifiques à un protocole. Les divers types d'informations sont organisés dans des dossiers différents. Chaque élément d'information est représenté par une *Variable* en lecture seule. Les informations peuvent être récupérées en lisant la valeur de la *Variable*. Les *Variables* appartiennent au *DeviceType*, c'est-à-dire, toutes les instances d'*Appareil* partagent typiquement le même *Variable Node* (Nœud de variable) et, lorsqu'elles sont lues, retournent la même valeur.

La Figure 8 illustre comment des données complémentaires sont intégrées dans un *DeviceType*.

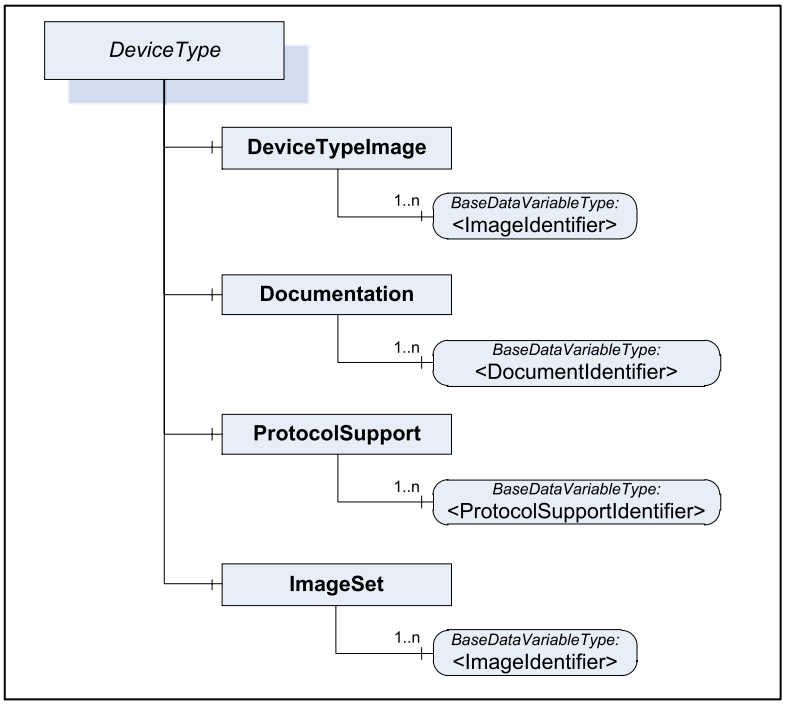


Figure 8 – Intégration des informations de prise en charge dans un DeviceType

Les *Clients* ont besoin d'être conscients que le contenu représenté par ces *Variables* peut être volumineux. Lire de grandes valeurs par une seule opération Read (Lire) peut ne pas être possible en raison de limites configurées soit dans le *Client*, soit dans la pile du *Serveur*. La taille minimale par défaut pour une matrice d'octets est 1 Mo (mégaoctet). Il convient que les *Clients* utilisent l'IndexRange (Plage d'indices) dans le service Read de l'OPC UA (voir l'IEC 62541-4) pour lire ces *Variables* par fragments (par exemple, des fragments de 1 Mo). Il appartient au *Client* de décider de démarrer sans indice et répéter avec une IndexRange seulement après une erreur ou de toujours utiliser une IndexRange.

Les différents types d'informations de prise en charge sont spécifiés dans les 5.7.2 à 5.7.5.

5.7.2 Image de type d'appareil

Des images d'un *Appareil* peuvent être exposées sous forme de *Variables* organisées dans le dossier *DeviceTypeImage*. Il peut y avoir plusieurs images à des résolutions différentes. Chaque image est une *Variable* distincte. Le *dossier* “*DeviceTypeImage*” est formellement défini dans le Tableau 11.

Tableau 11 – Définition de DeviceTypeImage

Attribut	Valeur				
Nom d'exploration	DeviceTypeImage				
Références	**Classe de nœuds**	**Nom d'exploration**	**Définition de type**	**Type de données**	**Règle de modélisation**
HasTypeDefinition	ObjectType	FolderType (défini dans l'IEC 62541-5)			
HasComponent	Variable	<ImageIdentifier>	BaseDataVariableType	Image	MandatoryPlaceholder

Toutes les images sont transférées comme une chaîne d'octets *ByteString*. Le *DataType* de la *Variable* spécifie le format d'image. L'OPC UA définit les formats BMP, GIF, JPG et PNG (voir IEC 62541-3).

5.7.3 Documentation

Les documents fournis pour un *Appareil* sont exposés sous forme de *Variables* organisées dans le dossier *Documentation*. Dans la plupart des cas, ils représentent un manuel de produit, qui peut exister sous la forme d'un ensemble de documents individuels. Le *dossier* "*Documentation*" est formellement défini dans le Tableau 12.

Tableau 12 – Définition de Documentation

Attribut	Valeur				
Nom d'exploration	Documentation				
Références	**Classe de nœuds**	**Nom d'exploration**	**Définition de type**	**Type de données**	**Règle de modélisation**
HasTypeDefinition	ObjectType	FolderType (défini dans l'IEC 62541-5)			
HasComponent	Variable	<DocumentIdentifier>	BaseDataVariableTyp e	ByteString	MandatoryPlaceholder

Tous les documents sont transférés comme une chaîne d'octets *ByteString*. Le *BrowseName* de chaque *Variable* est constitué du nom de fichier (filename) comprenant l'extension qui peut être utilisée pour identifier le type du document. Des extensions types sont ".pdf" ou ".txt".

5.7.4 Fichiers de prise en charge de protocole

Les fichiers de prise en charge de protocole fournis pour un *Appareil* sont exposés sous forme de *Variables* organisées dans le dossier *ProtocolSupport*. Ils peuvent représenter divers types d'informations tels que définis par un protocole. Des exemples sont un fichier GSD ou un fichier CFF. Le *dossier* "*ProtocolSupport*" est formellement défini dans le Tableau 13.

Tableau 13 – Définition de ProtocolSupport

Attribut	Valeur				
Nom d'exploration	ProtocolSupport				
Références	**Classe de nœuds**	**Nom d'exploration**	**Définition de type**	**Type de données**	**Règle de modélisation**
HasTypeDefinition	ObjectType	FolderType (défini dans l'IEC 62541-5)			
HasComponent	Variable	<ProtocolSupportIdentifier>	BaseDataVariableT ype	ByteString	MandatoryPlaceholder

Tous les fichiers de prise en charge de protocole sont transférés sous la forme d'une chaîne d'octets *ByteString*. Le *BrowseName* de chaque *Variable* doit être constitué du nom de fichier (filename) complet comprenant l'extension qui peut être utilisée pour identifier le type d'information.

5.7.5 Images

Les images qui sont utilisées au sein des *UIElements* sont exposées sous la forme de *Variables* séparées, plutôt que d'être intégrées dans l'élément. Toutes les *Variables* images seront agrégées par le dossier *ImageSet*. L'*UIElement* doit spécifier une image par son nom qui est aussi le *BrowseName* de la *Variable* image. Les *Clients* peuvent placer des images en cache et, ainsi, celles-ci ne doivent pas être transférées plus d'une fois. Le *dossier* "*ImageSet*" est formellement défini dans le Tableau 14.

Tableau 14 – Définition d'ImageSet

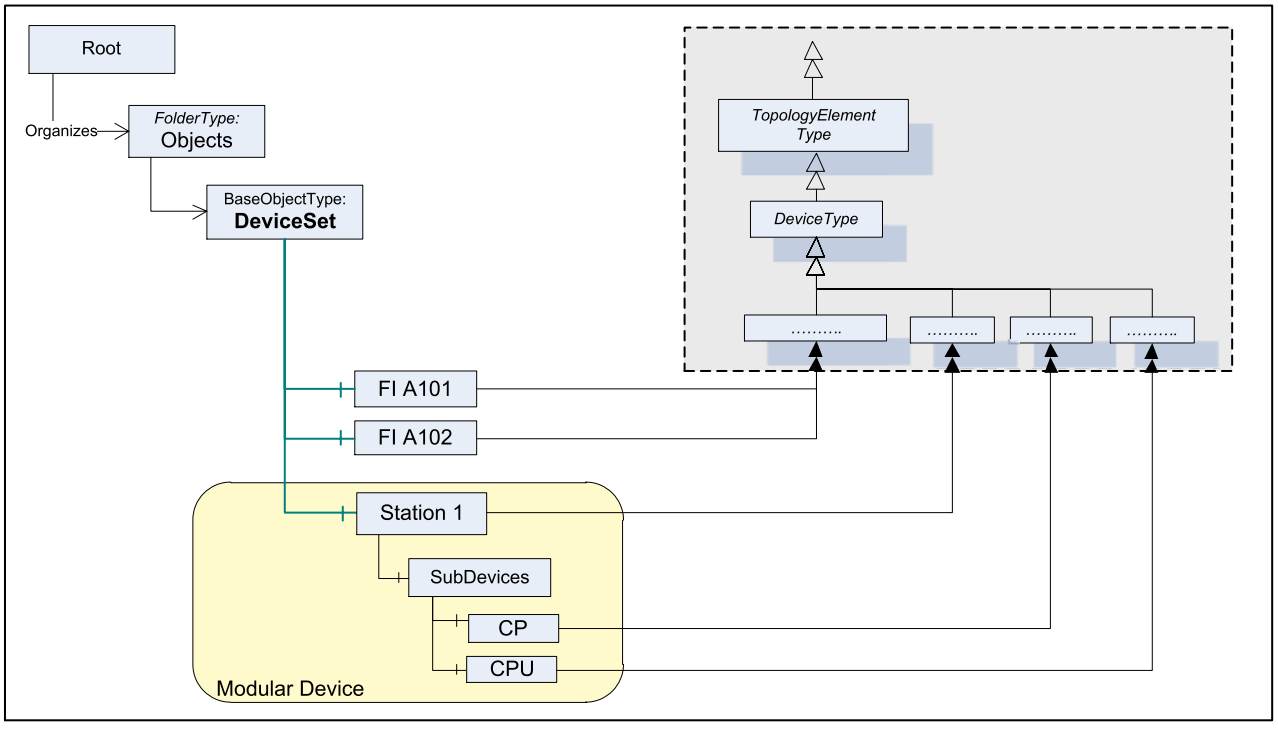
Attribut	**Valeur**				
Nom d'exploration	ImageSet				
Références	**Classe de nœuds**	**Nom d'exploration**	**Définition de type**	**Type de données**	**Règle de modélisation**
HasTypeDefinition	ObjectType	FolderType (défini dans l'IEC 62541-5)			
HasComponent	Variable	<ImageIdentifier>	BaseDataVariableType	Image	MandatoryPlaceholder

Le *DataType* de la *Variable* spécifie le format d'image. L'OPC UA définit les formats BMP, GIF, JPG et PNG (voir IEC 62541-3).

5.8 Point d'entrée de DeviceSet

Pour favoriser l'interopérabilité des *Client*s et des *Serveur*s, tous les *Appareils* instanciés doivent être agrégés dans un *Objet* appelé "**DeviceSet**" (Ensemble d'appareils). Pour les *Appareils* complexes qui sont composés de divers composants qui sont également des *Appareils*, seul l'*Appareil* de niveau supérieur doit être référencé à partir de l'*objet DeviceSet*. Les *Appareils* composants doivent être localisables en suivant les *Références HasComponent* partant de l'*Appareil* de niveau supérieur. Un exemple est le *Modular Device* (Appareil modulaire) défini en 9.3 et également illustré à la Figure 9.

La Figure 9 montre l'organisation des *AddressSpaces* (Espaces d'adresses) avec ce point d'entrée normalisé.



Anglais	Français
Root	Racine
Organizes	Organise
Modular Device	Appareil modulaire

Figure 9 – Point d'entrée normalisé pour Appareils

Le nœud DeviceSet est formellement défini dans le Tableau 15.

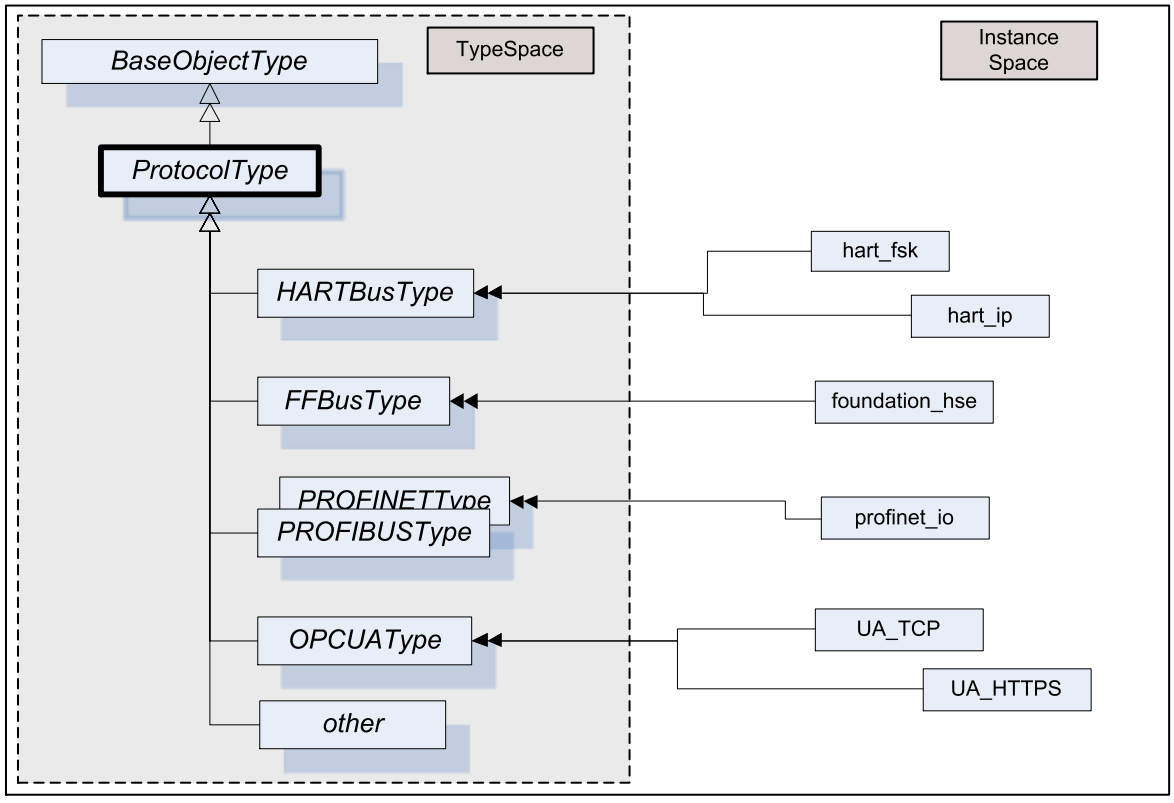
Tableau 15 – Définition de DeviceSet

Attribut	**Valeur**		
Nom d'exploration	DeviceSet		
Références	**Classe de nœuds**	**Nom d'exploration**	**Définition de type**
Organisée par le dossier d'Objets défini dans l'IEC 62541-5			
HasTypeDefinition	ObjectType	BaseObjectType	

5.9 ProtocolType

Le *type d'objet ProtocolType* (Type de protocole) et ses sous-types sont utilisés pour spécifier un protocole de communication/*FieldBus* spécifique qui est pris en charge par un *Appareil* ou un *Réseau*. Le ***BrowseName*** de chaque instance d'un *ProtocolType* doit définir le *Profil de communication* (voir Figure 10).

La Figure 10 montre le *ProtocolType* comprenant certains types et instances spécifiques qui représentent les *Profils de communication* du type en question. Il est formellement défini dans le Tableau 16.



Anglais	Français
Type Space	Espace de types
Instance Space	Espace d'instances
Other	Autres

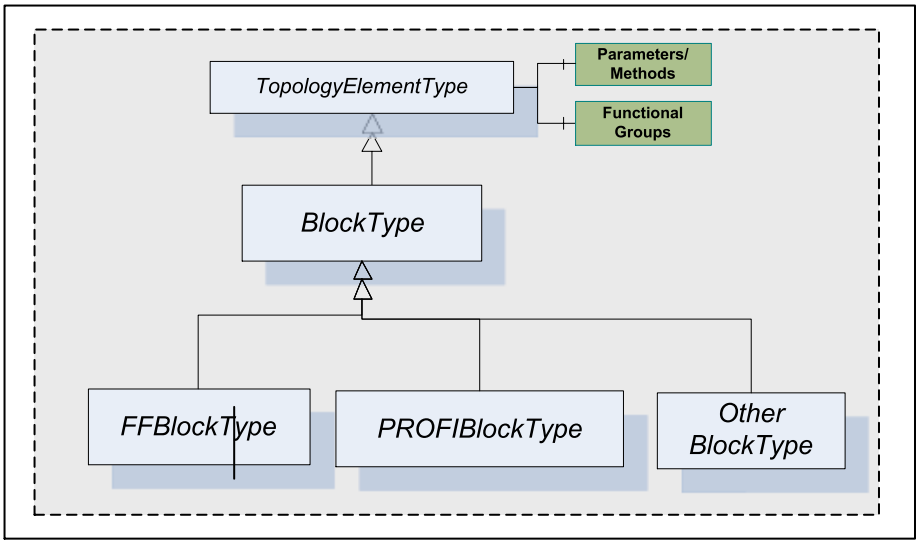
Figure 10 – Exemple de hiérarchie de ProtocolType avec des instances qui représentent des profils de communication spécifiques

Tableau 16 – Définition de ProtocolType

Attribut	**Valeur**				
Nom d'exploration	ProtocolType				
IsAbstract	Faux				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Sous-type du *BaseObjectType* défini dans l'IEC 62541-5					

5.10 BlockType

Cet *ObjectType* définit la structure d'un *Block Object* (Objet Bloc). La Figure 11 montre la hiérarchie de *BlockType*, formellement défini dans le Tableau 17.



Anglais	Français
Parameters / Methode	Paramètres / Méthodes
Functional Groups	Groupes fonctionnels

Figure 11 – Hiérarchie de BlockType

FFBlockType et PROFIBlockType sont des exemples. Ils ne sont pas décrits davantage dans la présente norme. Il est prévu que des groupes de l'industrie normalisent des *BlockTypes* d'usage général.

Tableau 17 – Définition de BlockType

Attribut	**Valeur**				
Nom d'exploration	BlockType				
IsAbstract	Vrai				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Hériter des *Propriétés* du *TopologyElementType* défini en 5.2					
HasProperty	Variable	RevisionCounter	Int32	PropertyType	Facultative
HasProperty	Variable	ActualMode	LocalizedText	PropertyType	Facultative
HasProperty	Variable	PermittedMode	LocalizedText[]	PropertyType	Facultative
HasProperty	Variable	NormalMode	LocalizedText[]	PropertyType	Facultative
HasProperty	Variable	TargetMode	LocalizedText[]	PropertyType	Facultative

BlockType est un sous-type de *TopologyElement* et hérite des éléments pour *Paramètres*, *Méthodes* et *FunctionalGroups*.

Le *BlockType* est abstrait. Il n'y aura pas d'instances d'un *BlockType* en soi, mais il y aura des instances de sous-types de ce *Type*. Dans la présente norme, le terme *Block* (Bloc) fait génériquement référence à une instance d'un sous-type quelconque du *BlockType*.

Le *RevisionCounter* (Compteur de révisions) est un compteur incrémentiel indiquant le nombre de fois que les données statiques dans le *Bloc* ont été modifiées. Une valeur de -1 indique qu'aucune information relative aux révisions n'est disponible.

Les *Propriétés* ci-après se réfèrent au *Mode* de *Bloc* (par exemple: “Manual”, c'est-à-dire “manuel”, “Out of Service”, c'est-à-dire “hors service”).

La *propriété ActualMode* (Mode réel) reflète le mode courant de fonctionnement.

PermittedMode (Mode permis) définit les modes de fonctionnement qui sont autorisés pour le *Bloc* selon les exigences de l'application.

NormalMode (Mode normal) est le mode dans lequel il convient de placer le *Bloc* dans les conditions normales de fonctionnement. Plusieurs modes peuvent exister selon la configuration du *Bloc*.

TargetMode (Mode cible) indique le mode de fonctionnement qui est souhaité pour le *Bloc*. Plusieurs modes peuvent exister selon la configuration du *Bloc*.

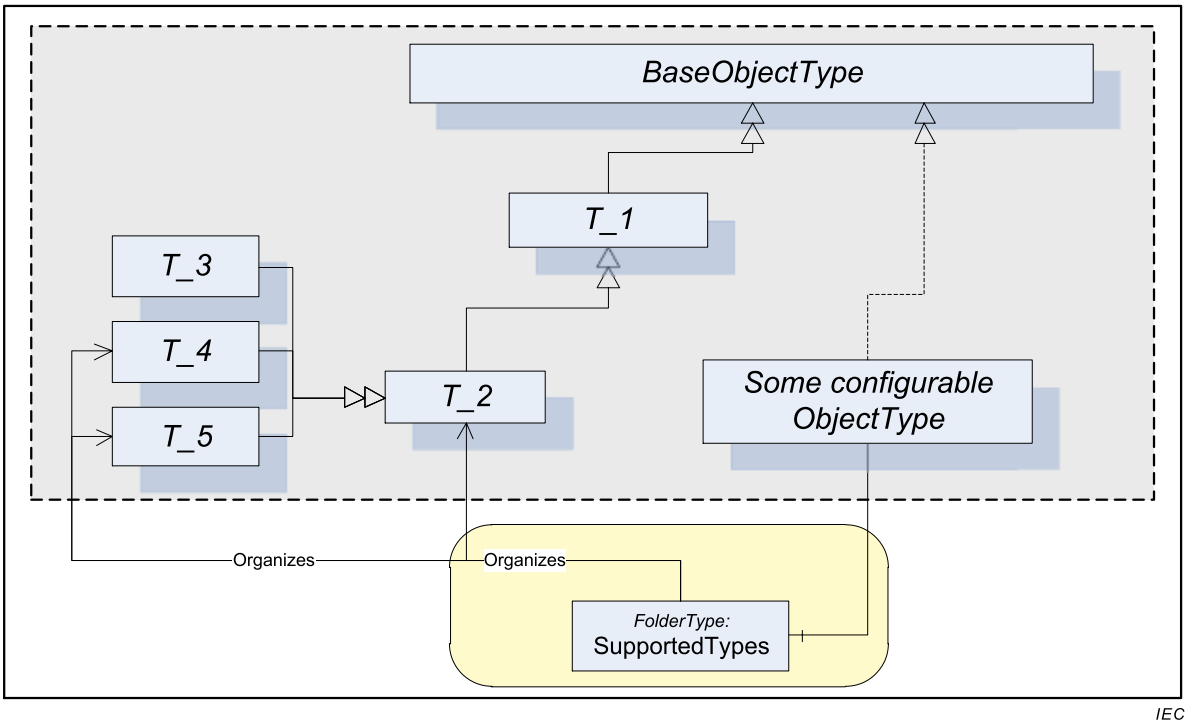
5.11 Composants configurables

5.11.1 Modèle général

Le présent paragraphe définit un modèle générique pour exposer et configurer les composants. Il définit les principes suivants:

- Un *Objet* configurable doit contenir un dossier appelé *SupportedTypes* (Types pris en charge) qui référence la liste des *Types* disponibles pour configurer les composants en utilisant les *Références Organizes* (organise). Des sous-dossiers peuvent être utilisés pour structurer davantage l'ensemble. Les noms de ces sous-dossiers sont spécifiques à un vendeur.
- Les instances configurées doivent être des composants de l'*Objet* configurable.

La Figure 12 illustre ces principes.



Anglais	Français
Some configurable ObjectType	Un certain ObjectType configurable
Organizes	Organise

Figure 12 – Modèle de composants configurables

Dans certains cas, le dossier *SupportedTypes* sur l'instance peut différer de celui sur le *Type* et peut contenir seulement un sous-ensemble. Il se peut, par exemple, qu'une seule instance de chaque *Type* puisse être configurée. Dans ce cas, la liste des *Types* pris en charge rétrécira avec chaque composant configuré. Si la liste des *Types* pris en charge est autorisée à rétrécir sur une instance, la *TypeDefinition* doit utiliser la *ModellingRule* appropriée comme "*Facultative*".

5.11.2 ConfigurableObjectType

Cet *ObjectType* met en œuvre le modèle de composants configurables et il est utilisé lorsqu'une déclaration d'*Objet* ou d'instance n'a besoin de rien d'autre qu'une capacité de configuration. La Figure 13 illustre le *ConfigurableObjectType*, formellement défini dans le Tableau 18. Un exemple concret est présenté à l'Article 9.

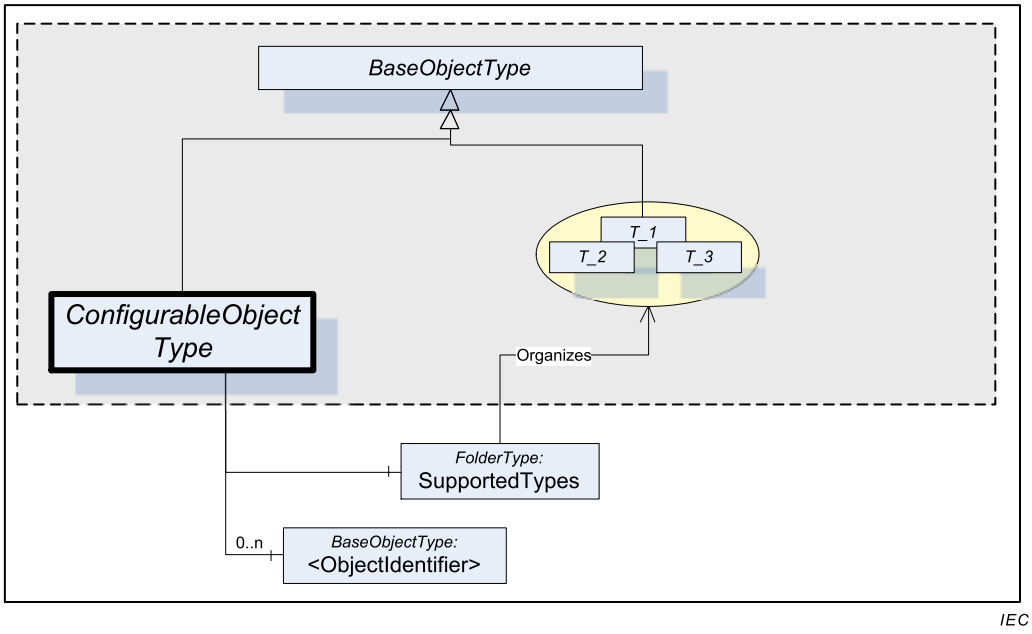


Figure 13 – ConfigurableObjectType

Tableau 18 – Définition de ConfigurableObjectType

Attribut	**Valeur**				
Nom d'exploration	ConfigurableObjectType				
IsAbstract	Faux				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Sous-type du *BaseObjectType* défini dans l'IEC 62541-5					
HasComponent	Objet	SupportedTypes		FolderType	Obligatoire
HasComponent	Objet	<ObjectIdentifier>		BaseObjectType	OptionalPlaceholder

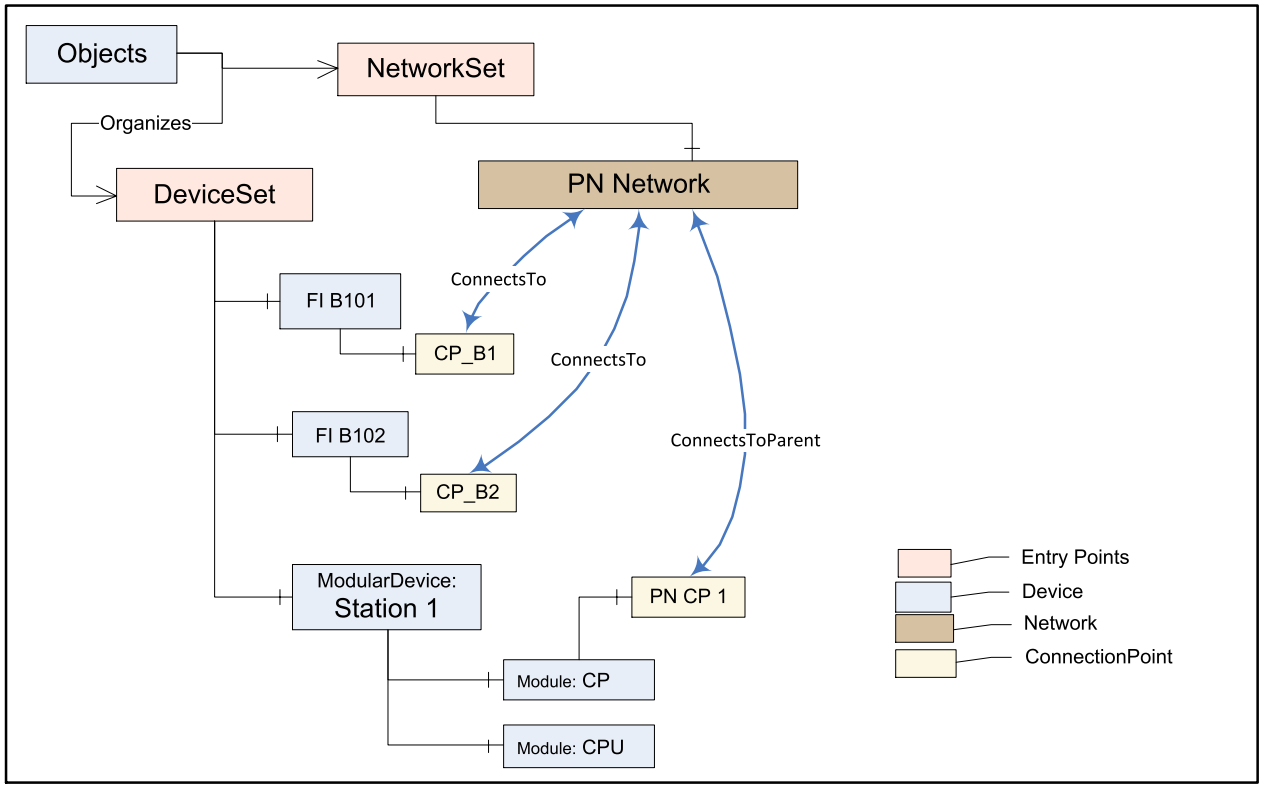
Le dossier *SupportedTypes* est utilisé pour maintenir l'ensemble des (sous-types des) *BaseObjectTypes* qui peuvent être instanciés dans cet *Objet* configurable (le plan d'action pour instancier des composants ne relève pas du domaine d'application de la présente norme).

Les instances configurées doivent être des composants de l'objet *ConfigurableObject*.

6 Modèle de communication d'appareil

6.1 Généralités

L'Article 6 introduit les *Références* et les *TopologyElementTypes* de base qui sont nécessaires pour créer une topologie de communication. La Figure 14 présent un aperçu initial des relations topologiques. Des exemples plus spécifiques suivront.



Anglais	Français
Objects	Objets
Organizes	Organise
Module: CPU	Module: CPU/UCT
Entry Points	Points d'entrée
Device	Appareil
Network	Réseau
ConnectionPoint	Point de connexion

Figure 14 – Exemple initial d'une topologie de communication

6.2 Réseau

Un *Réseau* représente le moyen de communication pour les *Appareils* qui lui sont connectés. Il inclut des technologies filaires et sans fil. Une instance de *Réseau* est qualifiée par le protocole qu'il référence.

La Figure 15 montre la hiérarchie des types et les composants de *NetworkType*, formellement défini dans le Tableau 19.

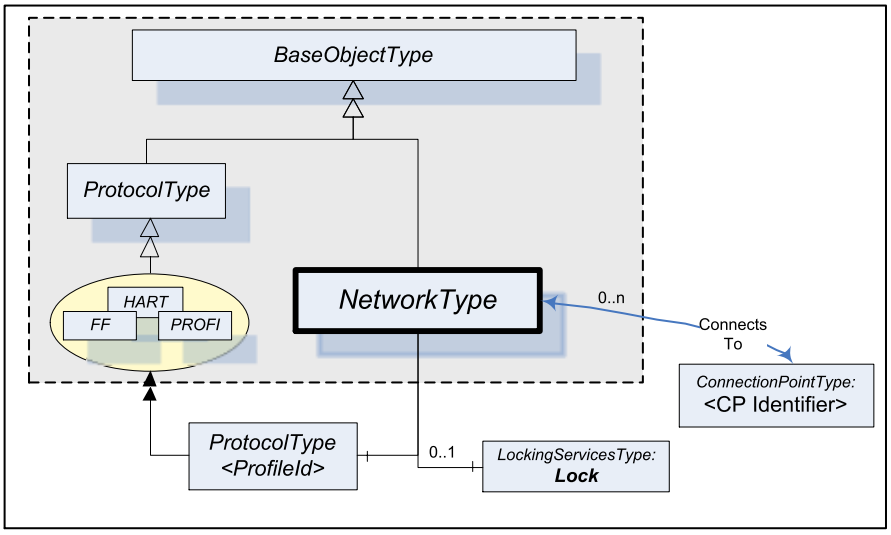


Figure 15 – NetworkType

Tableau 19 – Définition de NetworkType

Attribut	**Valeur**				
Nom d'exploration	NetworkType				
IsAbstract	Faux				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Sous-type du BaseObjectType défini dans l'IEC 62541-5					
HasComponent	Objet	<ProfileIdentifier>		ProtocolType	MandatoryPlaceholder
ConnectsTo	Objet	<CPIdentifier>		ConnectionPointType	OptionalPlaceholder
HasComponent	Objet	Lock		LockingServicesType	Facultative

Le <ProfileIdentifier> (Identificateur de profil) spécifie le *Protocole* et le *Profil de communication* pour lesquels ce *Réseau* est utilisé.

<CPIdentifier> (référencé par la *Référence ConnectsTo*) référence le(s) *ConnectionPoint*(s) qui a/ont été configuré(s) par ce *Réseau*. Tous les *ConnectionPoint*s doivent adhérer au même *Protocole* que le *Réseau*. Voir aussi la Figure 18 pour un exemple d'utilisation. Ils représentent les points d'accès, spécifiques à un protocole, pour les *Appareils* connectés.

De plus, les *Réseaux* peuvent aussi prendre en charge des *LockingServices* (définis en 8.3.3).

6.3 ConnectionPoint

Cet *ObjectType* représente l'interface (carte d'interface) d'un *Appareil* à un *Réseau*. Un sous-type spécifique doit être défini pour chaque protocole. La Figure 16 montre le *ConnectionPointType*, y compris certains types spécifiques.

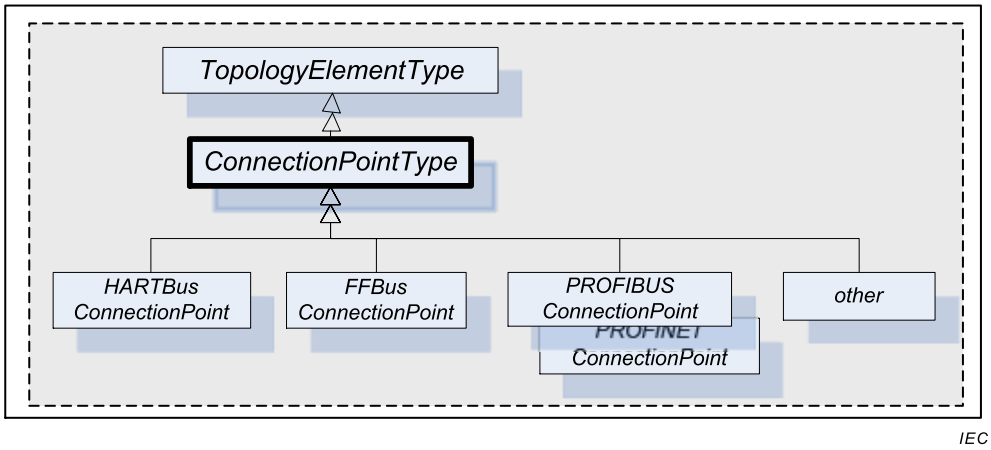


Figure 16 – Exemple de hiérarchie de ConnectionPointType

Un *Appareil* peut avoir plus d'une de ces interfaces aux mêmes *Réseaux* ou à des *Réseaux* différents. Habituellement, il existe des interfaces différentes pour des protocoles différents. La Figure 17 montre les composants du *ConnectionPointType*, formellement défini dans le Tableau 20.

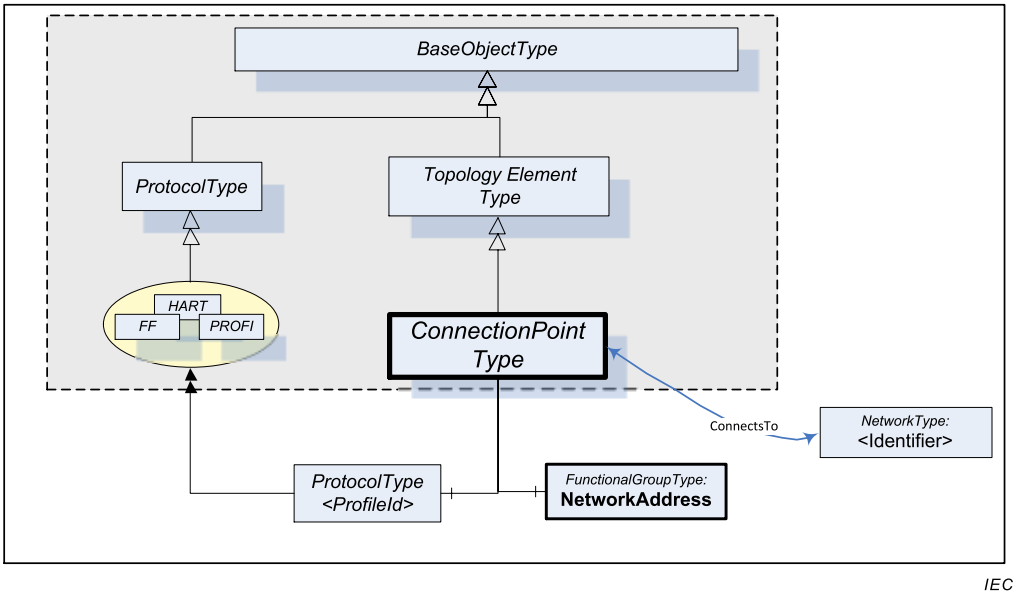


Figure 17 – ConnectionPointType

Tableau 20 – Définition de ConnectionPointType

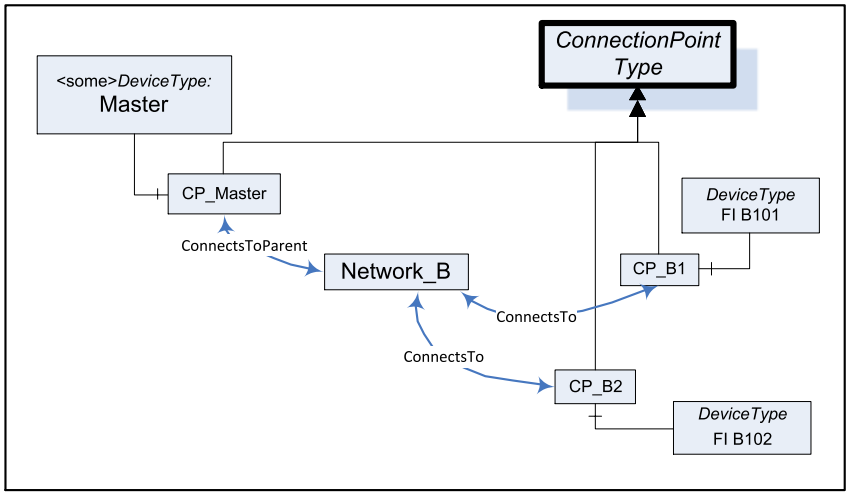
Attribut	**Valeur**				
Nom d'exploration	ConnectionPointType				
IsAbstract	Vrai				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Hérite des Propriétés du TopologyElementType défini en 5.2					
HasComponent	Objet	NetworkAddress		FunctionalGroupType	Obligatoire
HasComponent	Objet	<ProfileIdentifier>		ProtocolType	MandatoryPlaceHolder
ConnectsTo	Objet	<NetworkIdentifier>		NetworkType	OptionalPlaceHolder

Les *ConnectionPoints* ont des *Propriétés* et d'autres composants dont ils héritent du *TopologyElementType*.

Le *groupe fonctionnel NetworkAddress* organise tous les *Paramètres* nécessaires pour spécifier les informations spécifiques à un protocole qui sont relatives à l'adresse de l'*Appareil* connecté. Ces *Paramètres* sont aussi des composants du *ParameterSet*. Les *paramètres NetworkAddress* et leurs types de données seront spécifiés par les organisations de normalisation (les organisations Fieldbus, par exemple).

<ProfileIdentifier> identifie le *Profil de communication* que ce *ConnectionPoint* prend en charge. Le type de protocole *ProtocolType* et le *Profil de communication* sont définis en 5.9. Cela implique que ce *ConnectionPoint* peut être utilisé pour connecter des *Réseaux* et des *Appareils* du même *Profil de communication*.

Les *ConnectionPoints* sont entre un *Réseau* et un *Appareil*. L'emplacement dans la topologie est configuré au moyen du *ReferenceType ConnectsTo*. La Figure 18 illustre un certain nombre de modèles d'utilisation.



Anglais	Français
<some>DeviceType:Master	<un certain>Type d'appareils: Maître

Figure 18 – Utilisation des ConnectionPoints

6.4 Types de référence (ReferenceTypes) ConnectsTo et ConnectsToParent

Le *type de référence ConnectsTo* est un *ReferenceType* concret qui est utilisé pour indiquer que les Nœuds source et cible ont une connexion topologique. Il est à la fois hiérarchique et symétrique, car cela est naturel pour cette *Référence*. La *Référence ConnectsTo* existe entre un *réseau* et les *Appareils* connectés (ou leur*ConnectionPoint*). L'exploration d'un *Réseau* retourne les *Appareils* connectés; l'exploration à partir d'un *Appareil* permet de suivre la *Référence ConnectsTo* allant du *ConnectionPoint* de l'*Appareil* au *Réseau*.

Le *type de référence ConnectsToParent* est un *ReferenceType* concret qui est utilisé pour définir le parent (c'est-à-dire, l'*Appareil* de communication) d'un *Réseau*. Il est un sous-type du *ReferenceType ConnectsTo*.

Les deux *ReferenceTypes* sont illustrés à la Figure 19.

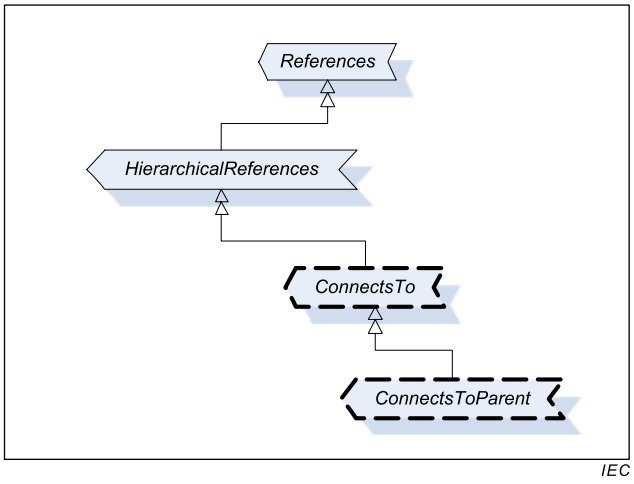


Figure 19 – Hiérarchies de types pour les Références ConnectsTo et ConnectsToParent

La représentation dans l'*AddressSpace* est spécifiée dans le Tableau 21 et dans le Tableau 22.

Tableau 21 – ReferenceType ConnectsTo

Attributs	**Valeur**		
Nom d'exploration	ConnectsTo		
Symmetric	Vrai		
IsAbstract	Faux		
Références	**Classe de nœuds**	**Nom d'exploration**	**Commentaire**
Sous-type du ReferenceType HierarchicalReferences défini dans l'IEC 62541-5			

Tableau 22 – ReferenceType ConnectsToParent

Attributs	**Valeur**		
Nom d'exploration	ConnectsToParent		
Symmetric	True		
IsAbstract	False		
Références	**Classe de nœuds**	**Nom d'exploration**	**Commentaire**
Sous-type du ReferenceType ConnectsTo			

La Figure 20 illustre comment cette *Référence* peut être utilisée pour exprimer des relations topologiques et parentales. Dans cet exemple, deux appareils sont connectés, à savoir le module DPcomm et l'*Appareil* de communication pour le *Réseau*.

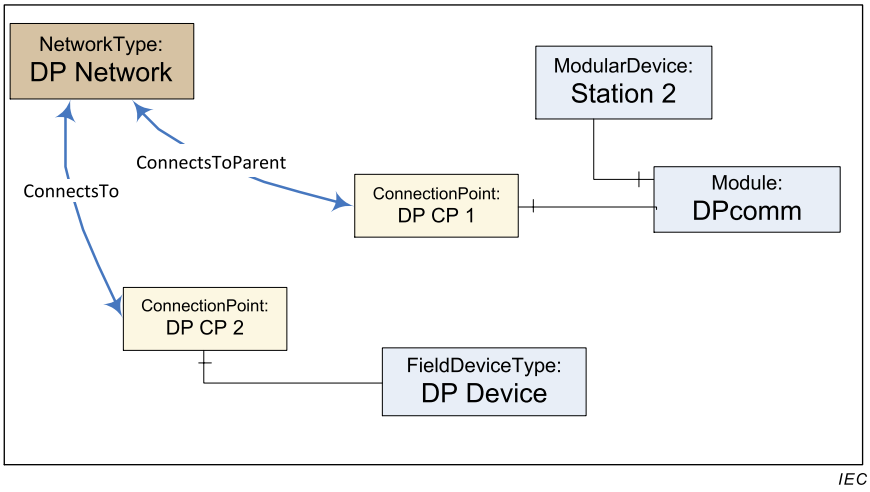


Figure 20 – Exemple avec les Références ConnectsTo et ConnectsToParent

6.5 Objet NetworkSet (obligatoire)

Tous les *Réseaux* sont des composants de l'objet *NetworkSet* (ensemble de réseaux)

Le *Nœud* NetworkSet est formellement défini dans le Tableau 23.

Tableau 23 – Définition de NetworkSet

Attribut	**Valeur**		
Nom d'exploration	NetworkSet		
Références	**Classe de nœuds**	**Nom d'exploration**	**Définition de type**
Organisée par le dossier d'Objets défini dans l'IEC 62541-5			
HasTypeDefinition	ObjectType	BaseObjectType	

7 Modèle d'hôte d'intégration d'appareils

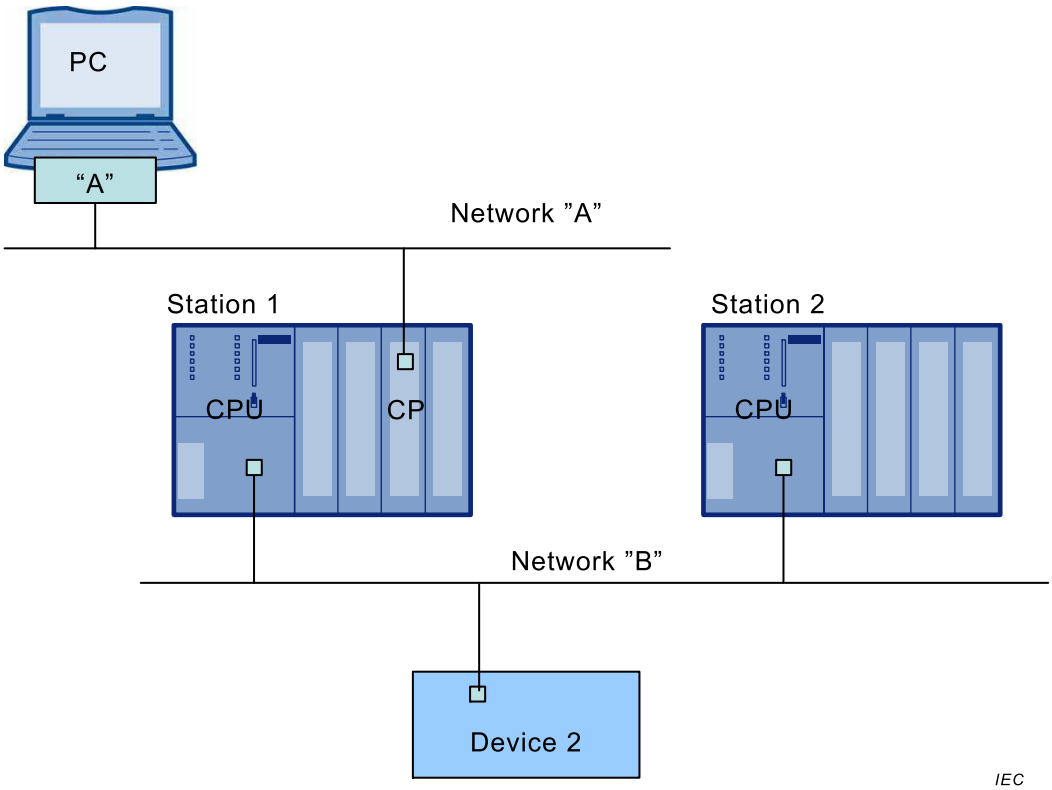
7.1 Généralités

Un *Device Integration Host* (Hôte d'intégration d'appareils) est un *Serveur* qui gère l'intégration de plusieurs appareils dans un système d'automation et fournit aux *Clients* un accès à des informations relatives à des *Appareils*, quel que soit l'emplacement où les informations sont stockées, par exemple, dans l'*Appareil* lui-même ou dans un magasin de données. La communication des appareils est interne à l'Hôte et peut être basée sur des protocoles spécifiques à un champ.

Le *Modèle d'information* spécifie les entités qui peuvent être accessibles dans un *Hôte d'intégration d'appareils*. La présente norme ne définit pas comment ces éléments sont instanciés. L'Hôte peut utiliser les services de balayage réseau, les *Services de gestion de nœuds* de l'OPC UA ou des outils de configuration propriétaires.

Une des principales tâches du *Modèle d'information* est de refléter la topologie du système d'automation. Par conséquent, il représente les *Appareils* du système d'automation ainsi que les réseaux de communication qui se connectent, y compris leurs propriétés, relations, et les opérations qui peuvent être effectuées sur eux.

La Figure 21 et la Figure 22 illustrent un exemple de configuration et la topologie configurée telle qu'elle apparaîtra dans l'*AddressSpace* (Espace d'adresse) du *Serveur* (les détails sont omis).

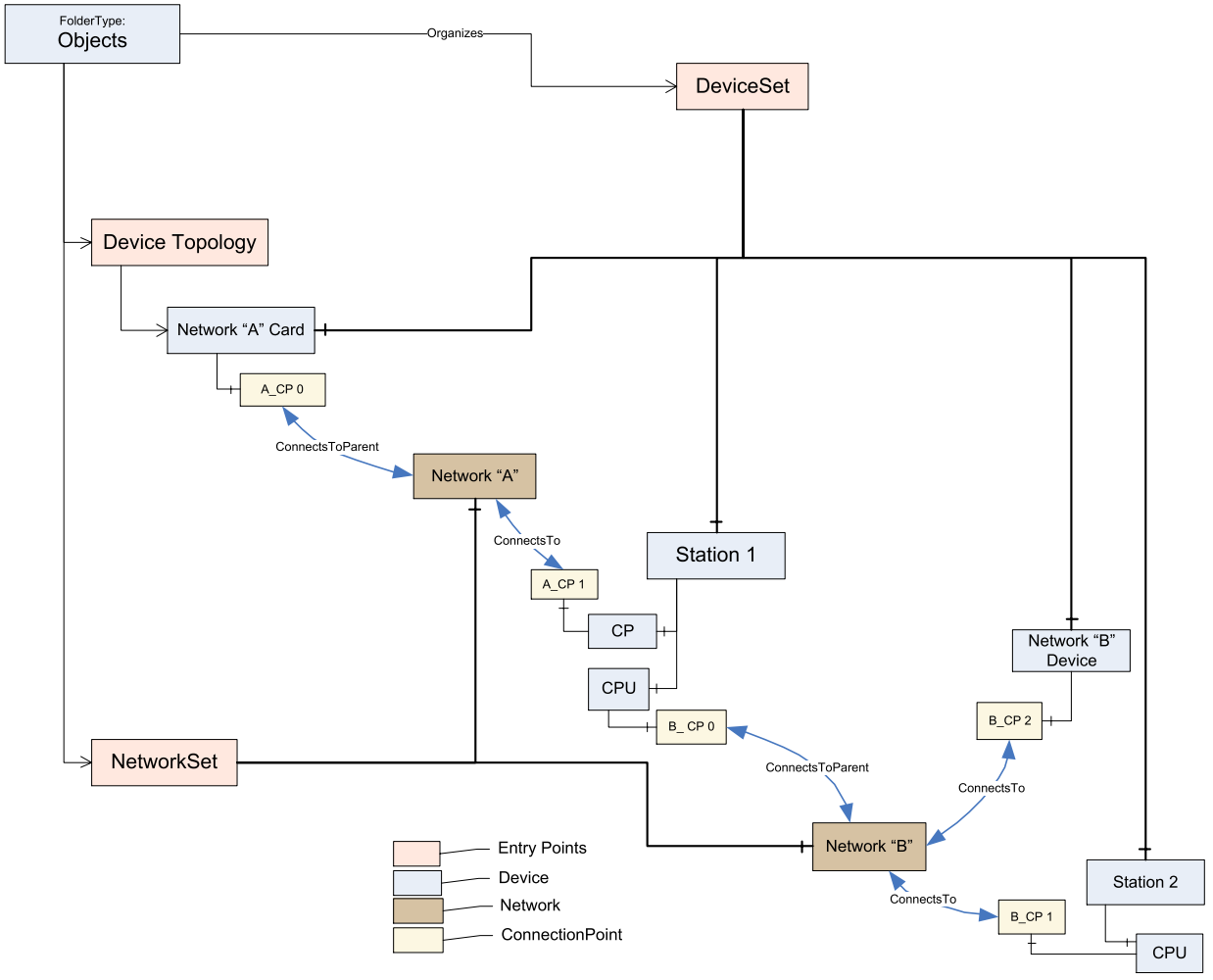


Légende

Anglais	Français
PC "A"	PC/ordinateur personnel "A"
Network A	Réseau "A"
Station 1	Station 1
Station 2	Station 2
CPU	CPU/UCT
Network "B"	Réseau "B"
Device 2	Appareil 2

Figure 21 – Exemple de système d'automation

Le PC à la Figure 21 représente le *Serveur* (l'*Hôte d'intégration d'appareils*). Le *Serveur* communique avec des appareils connectés au *Réseau* "A" par une communication native et communique avec des appareils connectés au *Réseau* "B" par une communication imbriquée.



Anglais	Français
Network A card	Carte réseau "A"
Device Topology	Topologie d'appareil
Network A	Réseau "A"
CPU	CPU/UCT
Network B Device	Appareil du Réseau "B"
Network "B"	Réseau "b"
Entry Points	Points d'entrée
Device	Appareil
Network	Réseau
ConnexionPoint	Point de connexion

Figure 22 – Exemple de topologie d'appareils

Des rectangles colorés sont utilisés pour reconnaître les divers types d'informations (voir la légende).

Des points d'entrée assurent le comportement commun parmi les mises en œuvre différentes:

- **DeviceTopology**: Nœud de départ pour la configuration de la topologie (voir 7.2).
- **DeviceSet**: Voir 5.8.
- **NetworkSet:** Voir 6.5.

7.2 Objet DeviceTopology

La *Device Topology* (Topologie d'appareils) reflète la topologie de communication des *Appareils*. Elle inclut des *Appareils* et les *Réseaux*. Le point d'entrée **DeviceTopology** est le point de départ dans l'*AddressSpace* et il est utilisé pour organiser les *Appareils* de communication pour les *Réseaux* de niveau supérieur qui fournissent un accès à toutes les instances qui constituent la *Topologie* d'*appareils* ((sous-)réseaux, appareils et éléments de communication).

Le nœud *DeviceTopology* est formellement défini dans le Tableau 24.

Tableau 24 – Définition de DeviceTopology

Attribut	**Valeur**			
Nom d'exploration	DeviceTopology			
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**
Organisée par le dossier d'Objets défini dans l'IEC 62541-5				
HasTypeDefinition	ObjectType	BaseObjectType	Défini dans l'IEC 62541-5.	
HasProperty	Variable	OnlineAccess	Boolean	PropertyType

OnlineAccess (Accès en ligne) donne une idée permettant de savoir si le *Serveur* est actuellement capable de communiquer avec des *Appareils* dans la topologie. “False” (Faux) signifie qu'aucune communication n'est disponible.

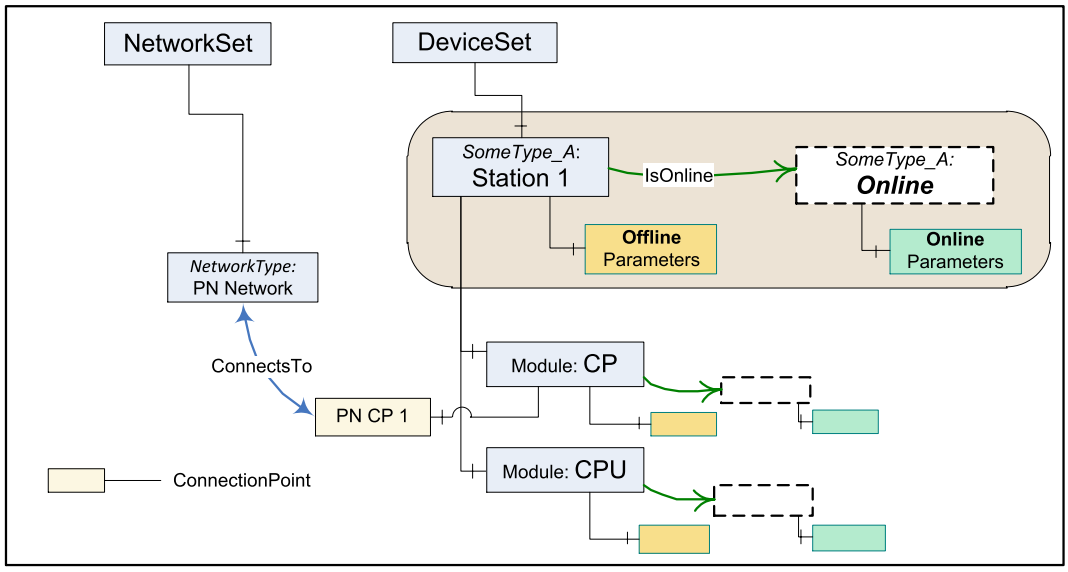
7.3 En ligne/Hors ligne

7.3.1 Généralités

La gestion de la *Topologie d'appareils* est une tâche de configuration, c'est-à-dire que les éléments dans la topologie (*Appareils*, *Réseaux*, et *Points de connexion*) sont habituellement configurés "offline" (hors ligne) et, ultérieurement, seront validés par rapport à leur représentant physique dans le réseau réel.

Pour prendre en charge l'accès explicite aux informations soit en ligne, soit hors ligne, chaque élément peut être représenté par deux instances qui sont schématiquement identiques, c'est-à-dire qu'il existe un *ParameterSet*, des *FunctionalGroups*, et ainsi de suite. Une *Référence* connecte les représentations en ligne et hors ligne et permet de naviguer entre elles.

Cela est illustré à la Figure 23.



Anglais	Français
Offline Parameters	Paramètres hors ligne
Online Parameters	Paramètres en ligne
Module: CPU	Module: CPU/UCT

Figure 23 – Composant en ligne pour l'accès aux données d'appareil

Si la configuration En ligne/Hors ligne est prise en charge, l'instance principale (directrice) représente les informations hors ligne. Sa *Référence HasTypeDefinition* pointe vers l'*ObjectType* concret qui est configuré ou identifié. Tous les *Paramètres* de cette instance représentent des points de données hors ligne. Leur lecture et leur écriture conduiront typiquement à un accès à la base de données de configuration. Si l'instance est un sous-type de *DeviceType*, ses *Propriétés* représenteront aussi des informations hors ligne.

Un *Appareil* peut être conçu à travers l'instance hors ligne sans accès en ligne

Les données en ligne pour un élément de topologie sont conservées dans un *Objet* associé avec le *Nom d'exploration* **Online** conformément à la Figure 23. L'*objet* **Online** (en ligne) est référencé par l'intermédiaire de la *Référence IsOnline*. Il est toujours du même type *ObjectType* que l'instance hors ligne

Les *Nœuds Paramètres* en ligne reflètent des valeurs dans un élément physique (typiquement un *Appareil*), c'est-à-dire que leur lecture ou leur écriture dans une valeur *Paramètre* conduira à une demande de communication à cet élément. Lorsque des éléments ne sont pas connectés, la lecture ou l'écriture dans le *Paramètre* en ligne retournera un code de statut correct (Bad_NotConnected).

Le transfert d'informations (*Paramètres*) entre les nœuds hors ligne et l'appareil physique dans l'ordre correct est pris en charge par l'intermédiaire de *TransferToDevice* (transfert vers l'appareil), *TransferFromDevice* (transfert à partir de l'appareil) conjointement avec *FetchTransferResultData* (rechercher les données de résultat de transfert). Ces *Méthodes* sont exposées au moyen d'une instance *AddIn* du type *TransferServicesType* décrit en 8.2.

Les éléments hors ligne et en ligne sont tous deux créés et pilotés par le même *ObjectType*. Selon leur aptitude à l'utilisation, certains composants (*Paramètres*, *Méthodes*, et *FunctionalGroups*) peuvent exister soit dans l'élément en ligne seulement, soit dans l'élément hors ligne seulement.

Un *Paramètre* dans le *ParameterSet* hors ligne et ses contreparties correspondantes dans le *ParameterSet* en ligne doivent avoir le *BrowseName*. Toutefois, leurs *NodeIds* (identificateurs de nœuds) ont besoin d'être différents, car il s'agit de l'identificateur transmis par le *Client* dans ses demandes de lecture/écriture.

Le *FunctionalGroup* **Identification** (voir 5.4) organise les *Paramètres* qui aident à identifier un élément de topologie. Les *Clients* peuvent comparer les valeurs de ces *Paramètres* dans l'instance en ligne et dans l'instance hors ligne pour détecter des discordances entre les données de configuration et l'élément actuellement connecté.

7.3.2 ReferenceType IsOnline

Le *type de référence IsOnline* est un *ReferenceType* concret utilisé pour lier la représentation hors ligne d'un *Appareil* à la représentation en ligne. Les *Nœuds* source et cible des *Références* de ce type doivent avoir une instance du même sous-type d'un *DeviceType*. Chaque *Appareil* doit être la source d'au plus une *Référence* de type *IsOnline*.

Le *type de référence IsOnline* est illustré à la Figure 24. Sa représentation dans l'*AddressSpace* est spécifiée dans le Tableau 25.

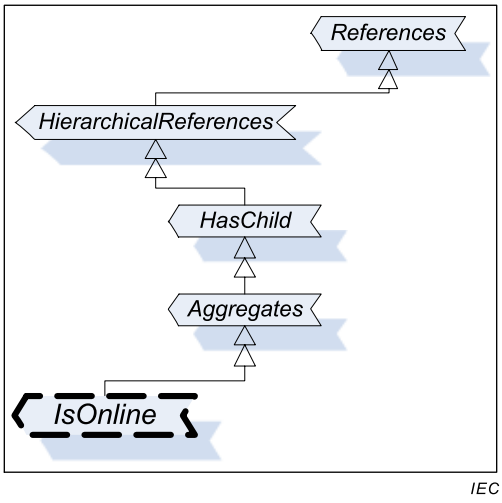


Figure 24 – Hiérarchie des types pour la Référence IsOnline

Tableau 25 – ReferenceType IsOnline

Attributs	**Valeur**		
Nom d'exploration	IsOnline		
InverseName	OnlineOf		
Symmetric	Faux		
IsAbstract	Faux		
Références	**Classe de nœuds**	**Nom d'exploration**	**Commentaire**
Sous-type du ReferenceType Aggregates défini dans l'IEC 62541-5			

8 Capacités d'AddIn

8.1 Vue d'ensemble

Le modèle de *Types* OPC UA est ciblé vers un héritage unique, à savoir, un *ObjectType* ou un *VariableType* est censé être un sous-type d'un seul supertype. Toutefois, outre le sous-typage, l'OPC UA autorise en général d'étendre les types ou les instances avec des composants supplémentaires. L'OPC UA décrit une bonne pratique – appelée ***AddIn Capability*** (Capacité d'ajout d'options) – relative à la façon d'étendre les *Objets* et les

Variables dans l'OPC UA et d'éviter les héritages. Les *Capacités* d'AddIn sont comparables à la technologie des interfaces utilisée dans certains langages de programmation.

Chaque *Capacité* d'*AddIn* est modélisée sous la forme d'un *ObjectType*, indiscernable des autres *ObjectTypes* utilisés pour exposer la fonctionnalité complète, avec une exception: les instances des *ObjectTypes* AddIn auront toujours des *BrowseNames* prédéfinis.

Pour appeler une *Méthode* d'un *Objet AddIn*, l'*ObjectId* doit identifier l'*Objet AddIn* qui est un composant de l'*Objet* qui doit être verrouillé.

NOTE Cela peut être mieux illustré par la Figure 28, où l'*Objet* nommé “Lock” (Verrou) est l'*objet AddIn*. Il possède plusieurs *Méthodes*, par exemple: InitLock et ExitLock. Pour appeler InitLock, l'appelant doit spécifier le *NodeId* de l'instance de “Lock” qui est un composant de MD002 dans l'argument objectId du *service Call* (Appel).

Les caractéristiques ci-après sont basées sur le concept de *Capacité* d'AddIn.

8.2 Transfert de données hors ligne – en ligne

8.2.1 Définition

Le transfert d'informations (*Paramètres*) entre des nœuds hors ligne et l'appareil physique est pris en charge par l'intermédiaire de *Méthodes* de l'OPC UA. Ces *Méthodes* sont construites sur des connaissances et des fonctionnalités spécifiques à un appareil.

Il est habituellement mis fin au transfert si une erreur se produit pour l'un quelconque des *Paramètres*. Aucune répétition de tentative automatique ne sera conduite par le *Serveur*. Cependant, chaque fois que cela est possible après une défaillance, il convient que le *Serveur* remette l'*Appareil* à son état fonctionnel. Le *Client* doit répéter la tentative en appelant de nouveau la *Méthode* de transfert.

Le transfert peut impliquer des milliers de *Paramètres*, ce qui peut allonger sa durée (de l'ordre de plusieurs minutes) et conduire à un résultat trop volumineux pour une seule et même réponse. Par conséquent, le déclenchement du transfert et la collecte des données obtenues sont accomplis par des *Méthodes* séparées.

L'*Appareil* doit avoir été verrouillé par le *Client* avant l'invocation de ces *Méthodes* (voir 8.3).

8.2.2 Type TransferServices

Le type *TransferServicesType* fournit les *Méthodes* nécessaires au transfert de données en provenance et à destination de l'*Appareil* en ligne. La Figure 25 montre la définition de *TransferServicesType*, formellement défini dans le Tableau 26.

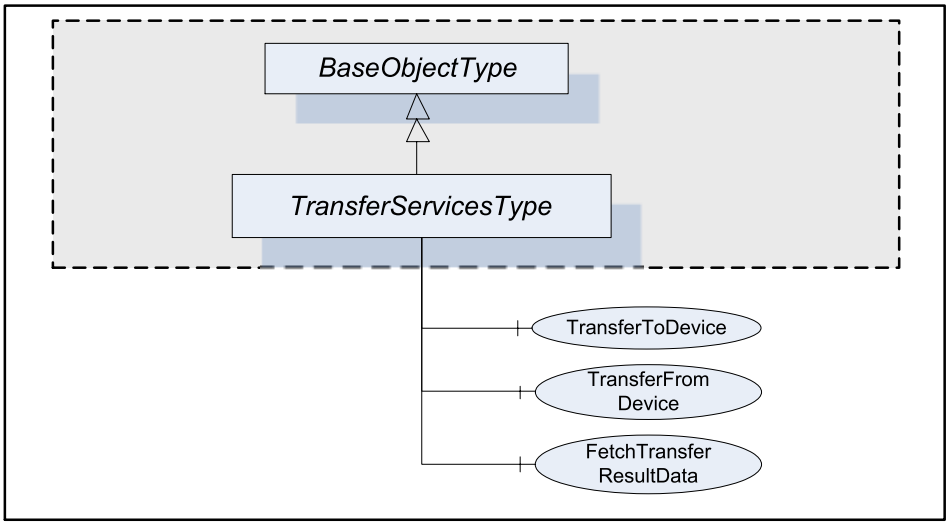


Figure 25 – TransferServicesType

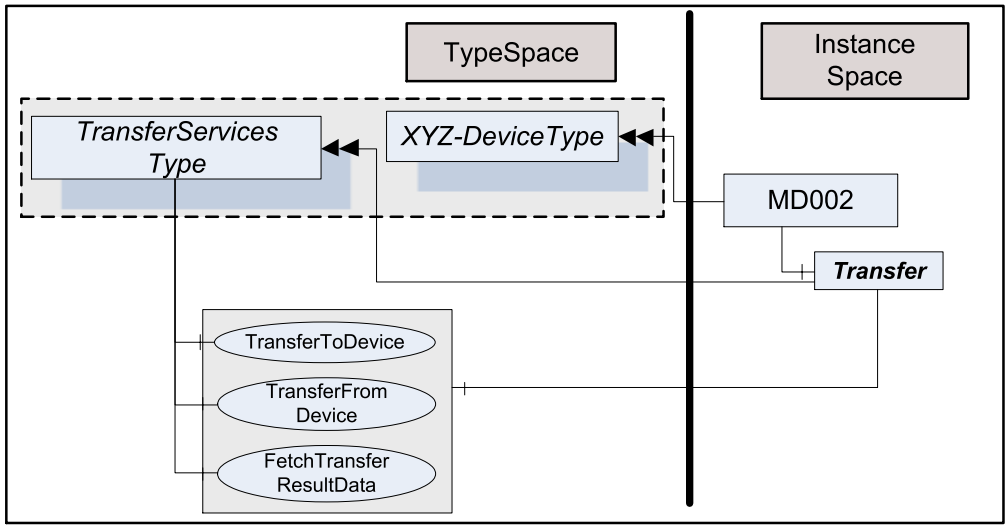
Tableau 26 – Définition de TransferServicesType

Attribut	**Valeur**				
Nom d'exploration	TransferServicesType				
IsAbstract	Faux				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Sous-type du BaseObjectType défini dans l'IEC 62541-5					
HasComponent	Méthode	TransferToDevice			Obligatoire
HasComponent	Méthode	TransferFromDevice			Obligatoire
HasComponent	Méthode	FetchTransferResultData			Obligatoire

Le *StatusCode* (code de statut) *Bad_MethodInvalid* ("Mauvais, car méthode non valide") doit être retourné par le *Service* Call pour les *Objets* où le verrouillage n'est pas pris en charge. *Bad_UserAccessDenied* ("Mauvais, car accès utilisateur refusé") doit être retourné si le*Client Utilisateur* n'a pas la permission d'appeler les *Méthodes*.

8.2.3 Objet TransferServices

La prise en charge des *TransferServices* pour un *Objet* est déclarée en agrégeant une instance du *TransferServicesType* conformément à la Figure 26.



Anglais	Français
TypeSpace	Espace de types
InstanceSpace	Espace d'instances
Transfer	Transfert

Figure 26 – TransferServices

Cet *Objet* est utilisé comme conteneur pour les *Méthodes TransferServices et doit avoir le Nom d'exploration* **Transfer**. *HasComponent* est utilisé pour faire une référence allant d'un *Appareil* à son *Objet* “TransferServices”.

Le *TransferServiceType* et chaque instance peuvent partager les mêmes *Méthodes*.

8.2.4 Méthode TransferToDevice

TransferToDevice déclenche le transfert de données configurées hors ligne (*Paramètres*) à l'appareil physique. Cette *Méthode* n'a pas d'arguments d'entrée. La question de savoir quels *Paramètres* sont transférés dépend des connaissances internes au serveur.

Le *Serveur* doit s'assurer de l'intégrité des données avant de démarrer le transfert. Une fois que le démarrage du transfert a réussi, la *Méthode* retourne immédiatement avec InitTransferStatus = 0. Toutes les informations de statut relatives au transfert lui-même doivent être recueillies à l'aide de la *Méthode FetchTransferResultData* (Rechercher les données de résultat de transfert).

Le *Serveur* réinitialisera toute valeur placée en cache pour les *Nœuds* dans l'instance en ligne représentant les *Paramètres* affectés par le transfert. De cette manière, le cache sera repeuplé à partir de l'*Appareil* la prochaine fois qu'on les demande.

La signature de cette *Méthode* est spécifiée ci-dessous. Le Tableau 27 et le Tableau 28 spécifient respectivement les arguments et la représentation de l'*AddressSpace*.

Signature

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

Tableau 27 – Arguments de la méthode TransferToDevice

Argument	Description
TransferID	Identificateur de transfert. Cet ID doit être utilisé pour appeler *FetchTransferResultData.*
InitTransferStatus	Spécifie si le transfert a été lancé. 0 – OK -1 – E_NotLocked – L'*Appareil* n'est pas verrouillé par le *Client* appelant. -2 – E_NotOnline – L'*Appareil* n'est pas en ligne / n'est pas accessible.

Tableau 28 – Définition de l'AdressSpace de la méthode TransferToDevice

Attribut	Valeur				
Nom d'exploration	TransferToDevice				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Obligatoire

8.2.5 Méthode TransferFromDevice

TransferFromDevice lance le transfert de valeurs de l'appareil physique vers les *Paramètres* correspondants dans la représentation hors ligne de l'*Appareil*. Cette *Méthode* n'a pas d'arguments d'entrée. La question de savoir quels *Paramètres* sont transférés dépend des connaissances internes au serveur.

Une fois que le démarrage du transfert a réussi, la *Méthode* retourne immédiatement avec InitTransferStatus = 0. Toutes les informations de statut relatives au transfert lui-même doivent être recueillies à l'aide de la *Méthode FetchTransferResultData.*

La signature de cette *Méthode* est spécifiée ci-dessous. Le Tableau 29 et le Tableau 30 spécifient respectivement les arguments et la représentation de l'*AddressSpace*.

Signature

```
TransferFromDevice(

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

Tableau 29 – Arguments de la méthode TransferFromDevice

Argument	Description
TransferID	Identificateur de transfert. Cet ID doit être utilisé pour appeler *FetchTransferResultData*.
InitTransferStatus	Spécifie si le transfert a été lancé. 0 – OK -1 – E_NotLocked – L'*Appareil* n'est pas verrouillé par le *Client* appelant. -2 – E_NotOnline – L'*Appareil* n'est pas en ligne / n'est pas accessible.

Tableau 30 – Définition de l'AdressSpace de la méthode TransferFromDevice

Attribut	Valeur				
Nom d'exploration	TransferFromDevice				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Obligatoire

8.2.6 Méthode FetchTransferResultData

Les *Méthodes TransferToDevice* et *TransferFromDevice* s'exécutent de façon asynchrone après l'envoi d'une réponse au *Client*. Le statut d'exécution et les résultats de l'exécution sont recueillis au cours de l'exécution et peuvent être récupérés à l'aide de la *Méthode FetchTransferResultData*. Le *TransferId* est utilisé comme identificateur pour récupérer les données.

Le *Client* est supposé rechercher les données de résultat en temps utile. Cependant, en raison de l'exécution asynchrone et de la possible perte de données à cause d'erreurs de transmission au *Client*, le *Serveur* doit attendre un certain temps (quelques minutes) avant d'effacer les données qui n'ont pas été acquittées. Il convient que cela aille même au-delà de la fin de la *Session*, autrement dit, les *Clients* qui doivent rétablir une *Session* après une erreur peuvent essayer de récupérer les données de résultat manquantes.

Les données de résultat seront effacées avec chaque nouvelle demande de transfert pour le même *Appareil*.

La méthode *FetchTransferResultData* est utilisée pour demander le statut de l'exécution et un ensemble de données de résultat. Si elle est appelée avant que le transfert ne soit terminé, elle ne retournera que des données partielles. La quantité des données retournées peut être limitée davantage si elle est susceptible d'être trop grande. “Trop grande” dans ce contexte signifie que le *Serveur* n'est pas capable de retourner une réponse plus volumineuse ou que le nombre de résultats à retourner dépasse le nombre maximal de résultats qui avait été spécifié par le *Client* lors de l'appel de cette *Méthode*.

Chaque résultat retourné au *Client* reçoit un numéro de séquence assigné. Le *Client* acquitte la réception du résultat en transmettant le numéro de séquence dans le nouvel appel de cette *Méthode*. Le *Serveur* peut supprimer le résultat acquitté et retournera le prochain ensemble de résultats avec un nouveau numéro de séquence.

Les *Clients* ne doivent pas appeler la *Méthode* avant que la précédente n'ait retourné un résultat. Si elle retourne une erreur (par exemple: Bad_Timeout), le *Client* peut appeler *FetchTransferResultData* avec le numéro de séquence 0. Dans ce cas, le *Serveur* renvoie le dernier ensemble de résultats.

Le Serveur retournera Bad_NothingToDo ("mauvais, car rien à faire") dans le *StatusCode* spécifique à une *Méthode* du *Service Call* si le transfert est terminé et aucune autre donnée de résultat n'est disponible.

La signature de cette *Méthode* est spécifiée ci-dessous. Le Tableau 31 et le Tableau 32 spécifient respectivement les arguments et la représentation de l'*AddressSpace*.

Signature

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

Tableau 31 – Arguments de la méthode FetchTransferResultData

Argument	Description
TransferID	Identificateur de transfert retourné à partir de *TransferToDevice* ou *TransferFromDevice*.
SequenceNumber	Le numéro de séquence en cours d'acquittement. Le *Serveur* peut supprimer l'ensemble de résultats avec ce numéro de séquence. “0” est utilisé dans le premier appel après initialisation d'un transfert et également si le précédent appel de *FetchTransferResultData* a échoué.
MaxParameterResultsToReturn	Le nombre de *Paramètres* dans *TransferResult.ParameterDefs* que le *Client* souhaite que le *Serveur* retourne dans la réponse. Le *Serveur* est autorisé à limiter davantage la réponse, mais il ne doit pas dépasser cette limite. Une valeur de 0 indique que le *Client* n'impose aucune limitation.
OmitGoodResults	Si TRUE (VRAI), le *Serveur* omettra des données pour les *Paramètres* qui ont été correctement transférés. Noter que cela entraîne l'émission de tous les bons résultats.
FetchResultData	Deux sous-types sont possibles: • le type TransferResultError est retourné si le transfert a complètement échoué • le type TransferResultData est retourné si le transfert a été parachevé. Des informations de statut sont retournées pour chaque *Paramètre* transféré.

Tableau 32 – Définition de l'AddressSpace de la méthode FetchTransferResultData

Attribut	Valeur				
Nom d’exploration	FetchTransferResultData				
Références	**Classe de nœuds**	**Nom d’exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Obligatoire
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Obligatoire

Le *FetchResultDataType* est un type abstrait. Il est le *DataType* de base pour les types de résultats concrets de la méthode *FetchTransferResultData*. Ses éléments sont définis dans le Tableau 33.

Tableau 33 – FetchResultDataType structure

Attribut	Valeur		
Nom d’exploration	FetchResultDataType		
IsAbstract	Vrai		
Sous-type de Structure définie dans l'IEC 62541-3			
Références	**Classe de nœuds**	**Nom d’exploration**	**Type de données**
HasSubtype	Type de données	TransferResultErrorDataType	Défini dans le Tableau 34.
HasSubtype	Type de données	TransferResultDataDataType	Défini dans le Tableau 35.

Le *DataType* TransferResultError est un sous-type de *FetchResultDataType* et représente un résultat erroné. Il est défini dans le Tableau 34.

Tableau 34 – TransferResultError DataType Structure

Dénomination	Type	Description
TransferResultError Type de données	Structure	Cette structure est retournée en cas d'erreurs. Aucune donnée de résultat n'est retournée. D'autres appels avec le même TransferId ne sont pas possibles.
status	Int32	-1 – Invalid TransferId (TransferId non valide): L'Id est inconnu. Cause possible: tous les résultats ont été recherchés et rapportés ou le résultat peut avoir été supprimé. -2 – Transfer aborted (Transfert abandonné): Le transfert a été abandonné; aucun résultat n'existe. -3 – DeviceError (Erreur appareil): Une erreur dans l'appareil ou de communication vers l'appareil s'est produite. “diagnostics” peut contenir des informations d'erreur spécifiques à un appareil ou spécifiques à un protocole. -4 – UnknownFailure (Défaillance inconnue): Le transfert a échoué. “diagnostics” peut contenir des informations d'erreur spécifiques à un appareil ou spécifiques à un protocole.
diagnostics	DiagnosticInfo	Informations de diagnostic. Ce paramètre est vide si des informations de diagnostic n'étaient pas demandées dans l'en-tête de demande ou si aucune information de diagnostic n'était vue au cours du traitement de la demande. Le type *DiagnosticInfo* est défini dans l'IEC 62541-4.

Le *DataType* TransferResultData est un sous-type de *FetchResultDataType* et comporte des résultats de paramètres obtenus par l'opération de transfert. Il est défini dans le Tableau 35.

Tableau 35 – TransferResultData DataType Structure

Dénomination	Type	Description
TransferResultData Type de données	Structure	Un ensemble de résultats obtenus dans l'opération de transfert.
sequenceNumber	Int32	Le numéro de séquence de cet ensemble de résultats
endOfResults	Boolean	TRUE (VRAI) – toutes les données de résultats ont été rapportées. Les appels supplémentaires de *FetchTransferResultData* avec le même *TransferID* retourneront une FetchTransferError avec status=InvalidTransferId. FALSE (FAUX) – d'autres données de résultats doivent être attendues
parameterDefs	structure[]	Valeur spécifique pour chaque *Paramètre* qui a été transféré. Si OmitGoodResults est TRUE, parameterDefs contiendra seulement les *Paramètres* qui n'ont pas été correctement transférés
NodePath	QualifiedName[]	Liste des BrowseNames qui représentent le chemin relatif allant de l'*Objet Appareil* au *Paramètre* en suivant les références hiérarchiques. Le *Client* peut utiliser ces noms pour *TranslateBrowsePathsToNodeIds* afin de récupérer le *Paramètre NodeId* pour la représentation en ligne ou hors ligne.
statusCode	StatusCode	*StatusCode* de l'OPC UA tel que défini dans l'IEC 62541-4 et dans l'IEC 62541-8.
diagnostics	DiagnosticInfo	Informations de diagnostic. Ce paramètre est vide si des informations de diagnostic n'étaient pas demandées dans l'en-tête de demande ou si aucune information de diagnostic n'était vue au cours du traitement de la demande. Le type *DiagnosticInfo* est défini dans l'IEC 62541-4.

8.3 Locking (Verrouillage)

8.3.1 Vue d’ensemble

Le verrouillage est le moyen d'éviter les modifications simultanées apportées à un *Appareil* ou à un *Réseau* et à leurs composants. Les *Clients* doivent utiliser les services de verrouillage s'ils ont besoin d'apporter un ensemble de modifications (par exemple, plusieurs opérations d'*Écriture* et invocation de *Méthodes*) et lorsqu'un état cohérent n'est disponible qu'après l'exécution de toutes les modifications. Le but principal du verrouillage d'un *Appareil* est d'éviter les modifications simultanées. Le but principal du verrouillage d'un *Réseau* est d'éviter les modifications simultanées de la topologie.

Un lock (verrou) fourni par un *Client* permet habituellement à d'autres *Clients* de visualiser (naviguer/lire) l'élément verrouillé. Les *Serveurs* peuvent choisir de mettre en œuvre un verrouillage exclusif dans lequel les autres *Clients* n'ont aucun accès du tout (par exemple, dans les cas où même les opérations de lecture exigent certains réglages dans un *Appareil*).

Pour verrouiller un *Appareil*, le verrou s'applique à l'*appareil* complet (y compris tous les composants comme les blocs ou les modules).

Pour verrouiller un *Appareil modulaire (voir* 9.3), le verrou s'applique à l'*Appareil* complet (y compris tous les modules). De même, pour verrouiller un *Appareil* orienté bloc (voir 9.2), le verrou s'applique à l'*Appareil* complet (y compris tous les blocs). Les *Serveurs* peuvent autoriser un verrouillage indépendant pour les sous-*Appareils* ou les blocs, si aucun verrou n'est appliqué à l'*Appareil* de niveau supérieur (pour l'*Appareil modulaire* ou pour l'*Appareil* à *Blocs*).

Si le modèle En ligne / Hors ligne est pris en charge (voir 7.3), le verrou s'applique toujours tant à la version en ligne qu'à la version hors ligne.

Pour verrouiller un *Réseau*, le verrou s'applique au *Réseau* et à tous les *Appareils* connectés. Si l'un des *Appareils* connectés fournit l'accès à un *Réseau* subordonné (comme une passerelle), le *Réseau* subordonné et ses *Appareils* connectés sont verrouillés aussi.

8.3.2 Type LockingServices

Le *LockingServicesType* fournit les *Méthodes* nécessaires pour le verrouillage ou le déverrouillage. La Figure 27 montre la définition de *LockingServicesType*, formellement défini dans le Tableau 36.

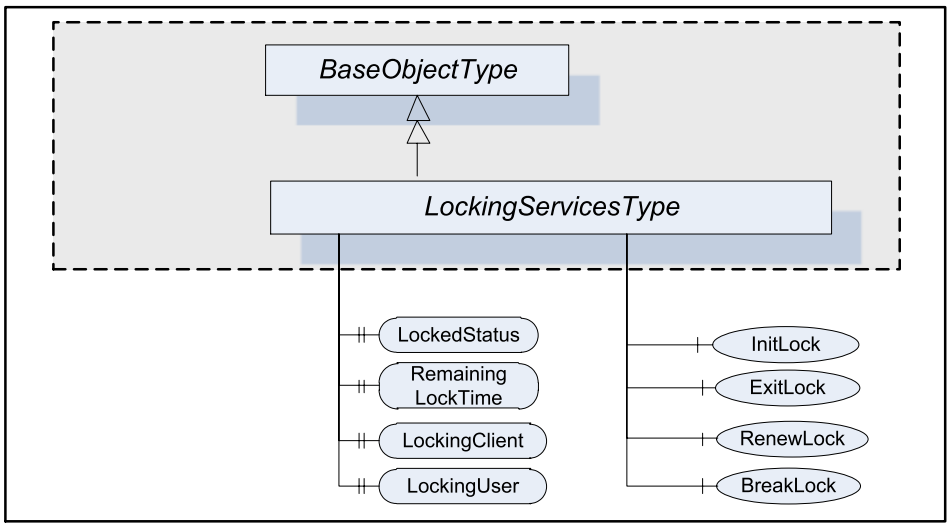


Figure 27 – LockingServicesType

Tableau 36 – Définition de LockingServicesType

Attribut	Valeur				
Nom d'exploration	LockingServicesType				
IsAbstract	Faux				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
Sous-type du BaseObjectType défini dans l'IEC 62541-5					
HasComponent	Méthode	InitLock			Obligatoire
HasComponent	Méthode	RenewLock			Obligatoire
HasComponent	Méthode	ExitLock			Obligatoire
HasComponent	Méthode	BreakLock			Obligatoire
HasProperty	Variable	Locked	Boolean	PropertyType	Obligatoire
HasProperty	Variable	LockingClient	String	PropertyType	Obligatoire
HasProperty	Variable	LockingUser	String	PropertyType	Obligatoire
HasProperty	Variable	RemainingLockTime	Duration	PropertyType	Obligatoire

Le *StatusCode* (Code de statut) *Bad_MethodInvalid* (Mauvais car méthode non valide) doit être retourné par le *Service* Call pour les *Objets* où le verrouillage n'est pas pris en charge. *Bad_UserAccessDenied* (mauvais, car accès utilisateur refusé) doit être retourné si le *Client Utilisateur* n'a pas la permission d'appeler les *Méthodes*.

Les *Propriétés* suivantes des *LockingServices* proposent des informations relatives au statut du verrou.

Locked (Verrouillé) lorsqu'il est True indique que cet élément a été verrouillé par un certain *Client* et que l'accès dont disposent les autres *Clients* est nul ou juste limité.

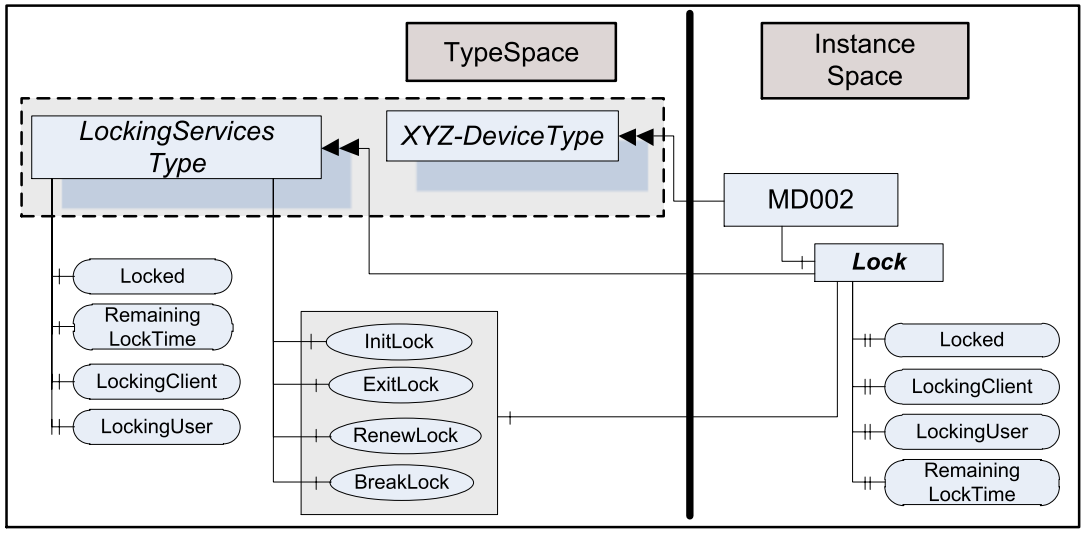
LockingClient contient l'ApplicationUri du *Client* tel que fourni dans l'appel de *Service CreateSession* (voir l'IEC 62541-4).

LockingUser contient l'identité de l'utilisateur. Il est obtenu directement ou indirectement à partir de l'*UserIdentityToken* (Jeton d'identité d'utilisateur) transmis par le *Client* dans l'appel de *Service ActivateSession* (voir IEC 62541-4).

RemainingLockTime désigne la durée restante, en millisecondes, au bout de laquelle le verrou sera automatiquement temporisé par le *Serveur*. Cette durée est basée sur *MaxInactiveLockTime* (voir 8.3.4).

8.3.3 Objet LockingServices

La prise en charge des *LockingServices* pour un *Objet* est déclarée en agrégeant une instance du *LockingServicesType* conformément à la Figure 28.



Anglais	Français
TypeSpace	Espace de types
Instance Space	Espace d'instances

Figure 28 – Services de verrouillage (LockingServices)

Cet *Objet* est utilisé comme conteneur pour les *Méthodes* et les *Propriétés* de *LockingServices* et doit avoir le *BrowseName* **Lock**. *HasComponent* est utilisé pour une référence allant d'un *TopologyElement* (par exemple, un *Appareil*) à son *Objet* "LockingServices".

Le *LockingServiceType* et chaque instance peuvent partager les mêmes *Méthodes*. Toutes les *Propriétés* sont distinctes.

8.3.4 Propriété MaxInactiveLockTime

La *Propriété MaxInactiveLockTime* doit être ajoutée à l'*objet ServerCapabilities* (voir IEC 62541-5).

Elle contient une durée spécifique à un serveur, en millisecondes, jusqu'à laquelle le *Serveur* révoquera le verrou. Le *Serveur* déclenchera un temporisateur comme partie intégrante du traitement de la demande *InitLock*. L'appel de la *Méthode RenewLock* ainsi que d'autres appels de *Service* issus du *Client* pour l'élément verrouillé doit réinitialiser le temporisateur. Le verrou ne sera jamais désactivé au cours de l'exécution d'un *Service* qui exige un verrou.

L'inactivité pendant *MaxInactiveLockTime* déclenchera la temporisation. En conséquence, le *Serveur* libérera le verrou.

Une Timeout (temporisation) ne doit annuler aucun *Service* déjà en exécution comme Write.

La *Propriété* "*MaxInactiveLockTime*" est formellement définie dans le Tableau 37.

Tableau 37 – Définition de la Propriété MaxInactiveLockTime

Références	Classe de nœuds	Nom d'exploration	Type de données	Définition de type	Règle de modélisation
HasProperty	Variable	MaxInactiveLockTime	Duration	PropertyType	Obligatoire

8.3.5 Méthode InitLock

InitLock limite l'accès pour les autres *Clients*.

Un appel de cette *Méthode* pour un élément qui est déjà verrouillé sera rejeté. Cela peut également être dû à un verrou implicite créé par le *Serveur*. Si la méthode *InitLock* est demandée pour un *Réseau*, elle sera rejetée si l'un quelconque des *Appareils* connectés à ce *Réseau* ou à tout autre *Réseau* subordonné, y compris leurs *Appareils,* est déjà verrouillé.

Alors qu'il est verrouillé, les demandes issues d'autres *Clients* de modifier l'élément verrouillé (par exemple, écriture dans des *Paramètres*, modification de la topologie, ou invocation de *Méthodes*) seront rejetées. Cependant, les demandes de lecture ou de navigation fonctionneront typiquement. Les *Serveurs* peuvent choisir de mettre en œuvre un verrouillage exclusif dans lequel les autres *Clients* n'ont aucun accès du tout (par exemple, dans les cas où même les opérations de lecture exigent certains réglages dans un *Appareil*).

Le verrou est retiré lorsque *ExitLock* est appelé. Il est automatiquement retiré lorsque la *Session* se termine. Tel est typiquement le cas lorsque la connexion au *Client* rompt et la *Session* expire. Les *Serveurs* doivent aussi maintenir un déverrouillage automatique si les *Clients* n'accèdent pas à l'élément verrouillé pendant une certaine durée (voir 8.3.4).

La signature de cette *Méthode* est spécifiée ci-dessous. Le Tableau 38 et le Tableau 39 spécifient respectivement les arguments et la représentation de l'*AddressSpace*.

Signature

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

Tableau 38 – Arguments de la méthode InitLock

Argument	Description
Context	Une chaîne de caractères utilisée pour fournir des informations de contexte relatives à l'activité en cours dans le *Client*.
InitLockStatus	0 – OK -1 – E_AlreadyLocked – l'élément est déjà verrouillé. -2 – E_Invalid – l'élément ne peut pas être verrouillé.

Tableau 39 – Définition de l'AdressSpace de la méthode InitLock

Attribut	Valeur				
Nom d'exploration	InitLock				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	Obligatoire
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Obligatoire

8.3.6 Méthode ExitLock

ExitLock retire le verrou. Cette *Méthode* ne peut être appelée qu'à partir de la même *Session* à partir de laquelle la méthode *InitLock* a été appelée.

La signature de cette *Méthode* est spécifiée ci-dessous. Le Tableau 40 et le Tableau 41 spécifient respectivement les arguments et la représentation de l'*AddressSpace*.

Signature

```
ExitLock(
```

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

Tableau 40 – Arguments de la méthode ExitLock

Argument	Description
ExitLockStatus	0 – OK -1 – E_NotLocked – l'*Objet* n'est pas verrouillé

Tableau 41 – Définition de l'AdressSpace de la méthode ExitLock

Attribut	Valeur				
Nom d'exploration	ExitLock				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Obligatoire

8.3.7 Méthode RenewLock

Le temporisateur du verrou est renouvelé automatiquement chaque fois que le *Client* lance une demande pour l'élément verrouillé ou lorsque des *Nœuds* de l'élément verrouillé sont l'objet d'un abonnement. La méthode *RenewLock* est utilisée pour réinitialiser le temporisateur du verrou à la valeur de la *Propriété MaxInactiveLockTime* et empêcher le *Serveur* d'abandonner automatiquement le verrou. Cette *Méthode* ne peut être appelée qu'à partir de la même *Session* à partir de laquelle la méthode *InitLock* a été appelée.

La signature de cette *Méthode* est spécifiée ci-dessous. Le Tableau 42 et le Tableau 43 spécifient respectivement les arguments et la représentation de l'*AddressSpace*.

Signature

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

Tableau 42 – Arguments de la méthode RenewLock

Argument	Description
RenewLockStatus	0 – OK -1 – E_NotLocked – l'Objet n'est pas verrouillé

Tableau 43 – Définition de l'AdressSpace de la méthode RenewLock

Attribut	Valeur				
Nom d'exploration	RenewLock				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Obligatoire

8.3.8 Méthode BreakLock

BreakLock permet à un *Client* (ayant des droits d'utilisateur suffisamment élevés) de casser le verrou détenu par un autre *Client*. Cette *Méthode* est typiquement disponible pour les seuls utilisateurs ayant des privilèges d'administrateurs. Il convient d'utiliser *BreakLock* avec précaution, car l'élément verrouillé peut être dans un état incohérent.

La signature de cette *Méthode* est spécifiée ci-dessous. Le Tableau 44 et le Tableau 45 spécifient respectivement les arguments et la représentation de l'*AddressSpace*.

Signature

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

Tableau 44 – Arguments de la méthode BreakLock

Argument	Description
BreakLockStatus	0 – OK -1 – E_NotLocked – l'Objet n'est pas verrouillé

Tableau 45 – Définition de l'AdressSpace de la méthode BreakLock

Attribut	Valeur				
Nom d'exploration	BreakLock				
Références	**Classe de nœuds**	**Nom d'exploration**	**Type de données**	**Définition de type**	**Règle de modélisation**
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	Obligatoire

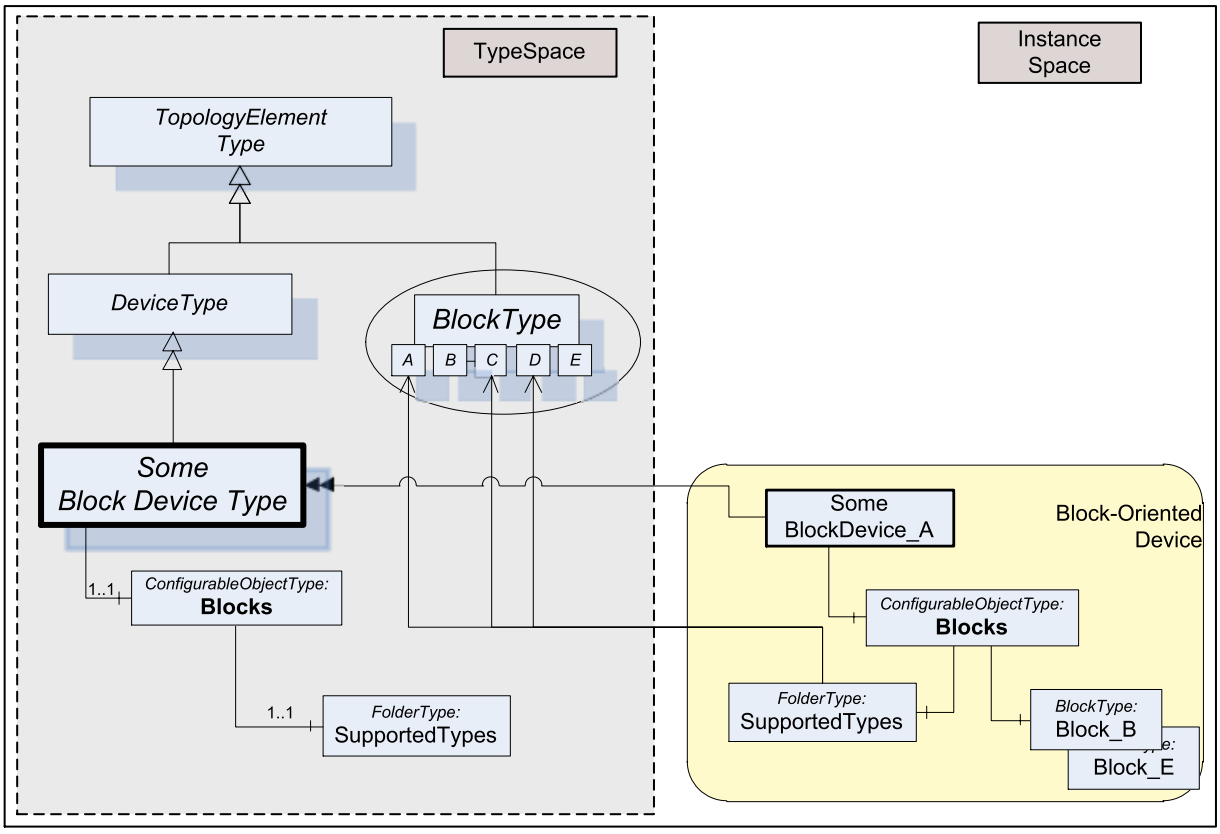
9 Éléments de topologie spécialisés

9.1 Généralités

Les *Appareils* et autres *TopologyElements* peuvent être renforcés en utilisant les éléments de modélisation définis dans la présente norme. Aucun *ObjectType* dérivé n'est nécessaire.

9.2 Appareils à Blocs (Type d'Appareils orientés bloc)

Un *Appareil* orienté bloc peut être constitué en utilisant les éléments de modélisation définis dans la présente norme. Un *Appareil* orienté bloc comprend un ensemble configurable de *Blocs*. La Figure 29 montre la structure générale des *Appareils* orientés bloc.



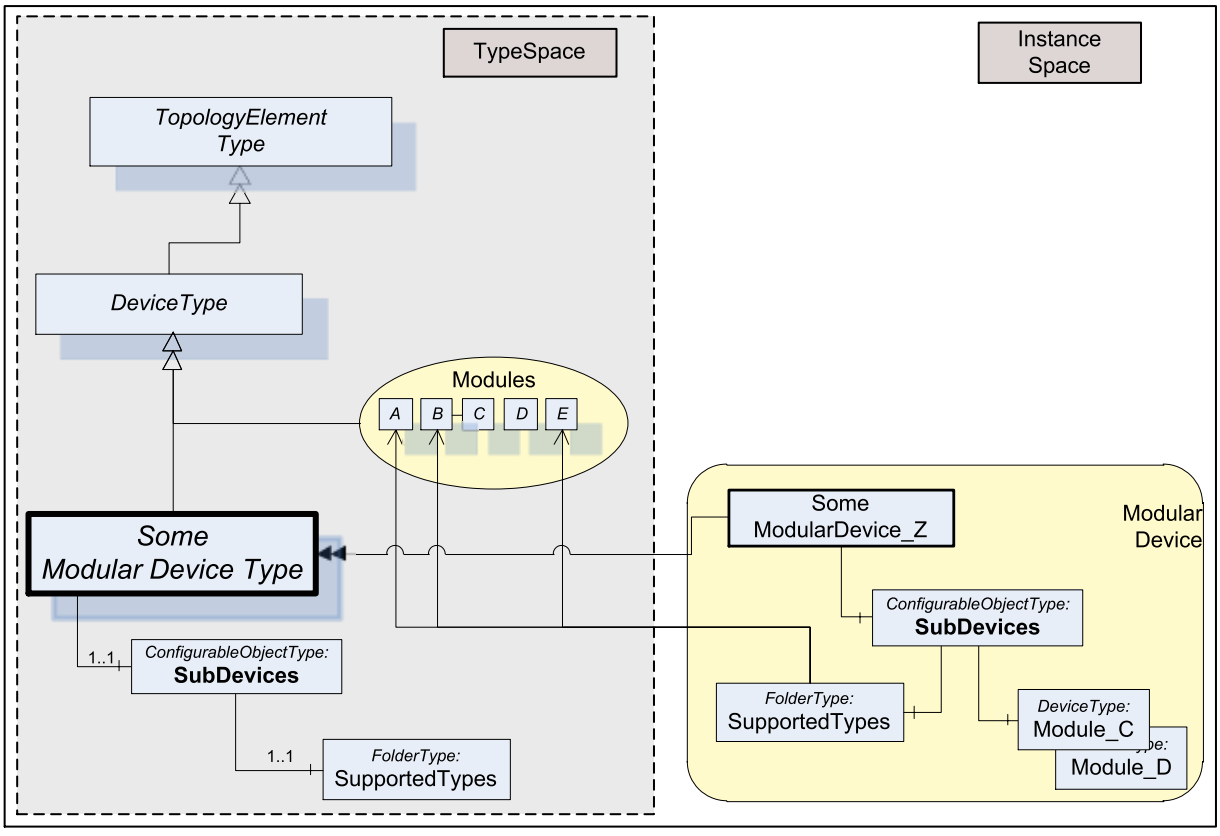
Anglais	Français
TypeSpace	Espace de types
Instance Space	Espace d'instances
Bloc-Oriented Device	Appareil orienté bloc

Figure 29 – Exemple de structure d'Appareils orientés bloc

Un *Objet* appelé ***Bloc*** est utilisé comme conteneur pour les instances réelles de *BlockType*. Il est du type *ConfigurableObjectType* qui inclut le dossier *SupportedTypes*. Le dossier *SupportedTypes* pour les ***Blocs*** est utilisé pour maintenir l'ensemble de (sous-types de) *BlockTypes* qui peuvent être instanciés. Les *Blocs* pris en charge peuvent être limités par l'*Appareil* orienté bloc. À la Figure 29, les *BlockTypes* B et E ont déjà été instanciés. Dans cet exemple, une seule instance de ces types est autorisée et le dossier *SupportedTypes* ne référence donc plus ces types. Voir en 5.11.1 la définition complète du *ConfigurableObjectType*.

9.3 Appareils modulaires

Un *Appareil modulaire* est un (sous-type d')*Appareil* qui est composé d'un *Appareil* supérieur et d'un ensemble de sous-appareils (modules). L'*Appareil* supérieur est souvent le module de tête avec la logique du programme, mais une grande partie de la fonctionnalité dépend des sous-appareils utilisés. Les *sous-appareils* pris en charge peuvent être limités par l'*Appareil modulaire*. La Figure 30 montre la structure générale des *Appareils modulaires*.



Anglais	Français
TypeSpace	Espace de types
Instance Space	Espace d'instances
Modular Device	Appareil modulaire

Figure 30 – Exemple de structure des Appareils modulaires

Les modules (sous-appareils) des *appareils modulaires* sont agrégés dans l'*Objet* **SubDevices** (Sous-appareils). Il est du type *ConfigurableObjectType* qui inclut le dossier *SupportedTypes*. Le dossier *SupportedTypes* pour les **SubDevices** est utilisé pour maintenir l'ensemble des (sous-types des) *DeviceTypes* qui peuvent être ajoutés à l'*Appareil modulaire*. Selon la configuration réelle, des instances d'*Appareil modulaire* pourraient déjà avoir un ensemble de sous-appareils préconfigurés. De surcroît, le dossier *SupportedTypes* pourrait seulement se référer à un sous-ensemble de tous les sous-appareils possibles pour l'*Appareil modulaire*. À la Figure 30, les DeviceTypes C et D ont déjà été instanciés. Dans cet exemple, une seule instance de ces types est autorisée et le dossier *SupportedTypes* ne référence donc plus ces types. Voir en 5.11.1 la définition complète du *ConfigurableObjectType*.

Les sous-appareils peuvent eux-mêmes être des *Appareils modulaires*.

10 Profils

10.1 Généralités

Les *Profils* sont des regroupements nommés de *ConformanceUnits* (Unités de conformité), tels que définis dans l'IEC 62541-7. Le terme Facet (Facette) dans le titre d'un *Profil* indique que ce *Profil* est censé être partie intégrante d'un autre *Profil* plus grand ou se rapporte à un aspect spécifique de l'OPC UA. Les *Profils* comportant le terme Facette dans leur titre sont censés être combinés avec d'autres *Profils* pour définir la fonctionnalité complète d'un *Serveur* ou d'un *Client* de l'OPC UA.

La présente norme définit des facettes *Profil* de l'OPC UA pour des *Serveurs* ou des *Clients* lorsqu'ils planifient de prendre en charge l'OPC UA pour les Appareils. Ces facettes sont décrites dans le 10.2 et 10.3.

10.2 Facettes Serveur d'appareils

Les tableaux ci-après spécifient les facettes disponibles pour les *Serveurs* qui mettent en œuvre la norme d'accompagnement *Appareils*. Le Tableau 46 décrit les *Unités de Conformité* incluses dans la facette nécessaire minimale. Cela inclut l'organisation des *Appareils* instanciés dans l'*AddressSpace Serveur*.

Tableau 46 – Définition de BaseDevice_Server_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Information Model (Modèle d'information DI)	Prise en charge d'*Objets* qui se conforment aux types spécifiés dans la norme d'accompagnements "*Appareils*". Cela inclut en particulier les *Objets* de *DeviceType* et *FunctionalGroups*.	M
DI DeviceSet (Ensemble d'appareils DI)	Prise en charge de l'objet **DeviceSet** pour agréger toutes les instances d'*Appareil*.	M
DI DeviceHealth (Santé d'appareil DI)	Prise en charge de la propriété DeviceHealth.	O
DI DeviceSupportInfo (Informations de prise en charge d'appareils)	Le Serveur fournit des données complémentaires pour ses Appareils telles que définies en 5.7.	O
Légende M = Obligatoire O = Facultative		

Le Tableau 47 définit une facette pour le *FunctionalGroup* "Identification" d'*Appareils*. Cela inclut l'option d'identifier le(s) *Protocole*(s).

Tableau 47 – Définition de DeviceIdentification_Server_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Identification	Prise en charge du *FunctionalGroup* **Identification** pour les *Appareils*.	M
DI_Protocol	Prise en charge du *ProtocolType* et de ses instances pour identifier les profils de communication utilisés pour des instances spécifiques.	O
Légende M = Obligatoire		

Le Tableau 48 définit des extensions spécifiquement nécessaires pour les *BlockDevices*.

Tableau 48 – Définition de BlockDevice_Server_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Blocks	Prise en charge du *BlockType* (ou des sous-types) et de l'*Objet Bloc* dans certains des *Appareils* instanciés.	M
Légende M = Obligatoire		

Le Tableau 49 définit une facette pour l'*AddIn* Locking *Capability* (verrouillage). Cela inclut l'option de casser le verrou.

Tableau 49 – Définition de Locking_Server_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Locking	Prise en charge du *LockingService* pour certains *TopologyElements*.	M
DI_BreakLocking	Prise en charge du service *BreakLock* pour casser le verrou détenu par un autre *Client*.	O
Légende M = Obligatoire O = Facultative		

Le Tableau 50 définit une facette pour prendre en charge le modèle de communication d'appareils.

Tableau 50 – Définition de DeviceCommunication_Server_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Network	Prise en charge du *NetworkType* pour instancier des instances de Réseau	M
DI ConnectionPoint	Prise en charge de sous-types du *ConnectionPointType*.	M
DI NetworkSet	Prise en charge de l'objet NetworkSet pour agréger toutes les instances de Réseau.	M
DI ConnectsTo	Prise en charge de la Référence ConnectsTo pour associer des Appareils à un Réseau.	M
Légende M = Obligatoire		

Le Tableau 51 définit une facette pour prendre en charge le modèle d'Hôte d'intégration d'appareils.

Tableau 51 – Définition de DeviceIntegrationHost_Server_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI DeviceTopology	Prise en charge de l'objet DeviceTopology comme nœud de départ pour la topologie de communication des appareils à intégrer.	M
DI Offline	Prise en charge des représentations hors ligne et en ligne des appareils, y compris les méthodes de transfert de données en provenance ou à destination de l'appareil.	M
Légende M = Obligatoire		

10.3 Facette Client d'appareils

Les tableaux ci-après spécifient les facettes disponibles pour les *Clients* qui mettent en œuvre la norme d'accompagnement *Appareils*. Le Tableau 52 décrit les *Unités de Conformité* incluses dans la facette nécessaire minimale.

Tableau 52 – Définition de BaseDevice_Client_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Client Information Model (Modèle d'information client DI)	Consommation d'*Objets* qui se conforment aux types spécifiés dans la norme d'accompagnements "*Appareils*". Cela inclut en particulier les *Objets* de *DeviceType* et *FunctionalGroups*.	M
DI Client DeviceSet	Utilisation de l'*objet* **DeviceSet** pour détecter les *Appareils* disponibles.	M
DI Client DeviceHealth	Utilisation de la propriété DeviceHealth.	O
DI Client DeviceSupportInfo	Utilisation des données complémentaires disponibles pour les Appareils, telles que définies en 5.7.	O
Légende M = Obligatoire O = Facultative		

Le Tableau 53 définit une facette pour le *FunctionalGroup* "identification" d'*Appareils*. Cela inclut l'option d'identifier le(s) *Protocole*(s).

Tableau 53 – Définition de DeviceIdentification_Client_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Client Identification	Consommation du *FunctionalGroup* **Identification** pour les *Appareils*, y compris la référence (facultative) au(x) protocole(s) pris en charge.	M
Légende M = Obligatoire		

Le Tableau 54 définit des extensions spécifiquement nécessaires pour les BlockDevices.

Tableau 54 – Définition de BlockDevice_Client_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Client Blocks	Compréhension et utilisation des BlockDevices et de leurs *Blocs*, y compris les *FunctionalGroups* au niveau tant appareil que bloc	M
Légende M = Obligatoire		

Le Tableau 55 définit une facette pour l'*AddIn Capability* Locking (verrouillage de capacité d'ajout d'option). Cela inclut l'option de casser le verrou.

Tableau 55 – Définition de Locking_Client_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Locking	Utilisation du *LockingService* lorsqu'il est disponible.	M
DI_BreakLocking	Prise en charge de l'utilisation du service *BreakLock* pour casser le verrou détenu par un autre *Client*.	O
Légende M = Obligatoire O = Facultative		

Le Tableau 56 définit une facette pour prendre en charge le modèle de communication d'appareils.

Tableau 56 – Définition de DeviceCommunication_Client_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI Network	Utilisation du *NetworkType* pour instancier des instances de Réseau	M
DI ConnectionPoint	Utilisation de sous-types du *ConnectionPointType*.	M
DI NetworkSet	Utilisation de l'objet NetworkSet pour stocker ou trouver des instances de Réseau.	M
DI ConnectsTo	Utilisation de la Référence ConnectsTo pour associer des Appareils à un Réseau.	M
Légende M = Obligatoire		

Le Tableau 57 définit une facette pour prendre en charge le modèle d'Hôte d'intégration d'appareils.

Tableau 57 – Définition de DeviceIntegrationHost_Client_Facet

Unité de Conformité	Description	Facultative/ Obligatoire
DI DeviceTopology	Utilisation de l'objet DeviceTopology comme nœud de départ pour la topologie de communication des appareils à intégrer.	M
DI Offline	Utilisation des représentations hors ligne et en ligne des appareils, y compris les méthodes de transfert de données en provenance ou à destination de l'appareil.	M
Légende M = Obligatoire		

Annexe A
(normative)

Espace de noms et mappings

La présente définit les identificateurs numériques pour tous les *NodeIds* numériques définis dans la présente norme. Les identificateurs sont spécifiés dans un fichier CSV avec la syntaxe suivante:

```
<SymbolName>, <Identifier>, <NodeClass>, c'est-à-dire
<Nom symbolique>, <Identificateur>, <Classe de nœuds>
```

où *SymbolName* est soit le *BrowseName* d'un *Type Node* (nœud de type), soit le *BrowsePath* (chemin d'exploration) pour un *Instance Node* (Nœud d'instance) qui apparaît dans la norme et l'*Identifier* (identificateur) est la valeur numérique pour le *NodeId* (Identificateur de nœud).

Le *BrowsePath* pour une *Instance Node* est construit en aboutant le *BrowseName* de l'instance *Node* au *BrowseName* pour l'instance ou type conteneur. Un trait de soulignement est utilisé pour séparer chaque *BrowseName* dans le chemin. Soit par exemple, le Nœud de type objet de type appareil (*DeviceType ObjectType Node*) qui a la *SerialNumber Property* (propriété numéro de série). Le **Nom** pour la déclaration d'instance (*InstanceDeclaration*) de *SerialNumber* au sein de la déclaration de *DeviceType* est: *DeviceType_SerialNumber*.

Le *NamespaceUri* (identificateur uniforme de ressource d'espace de noms) pour tous les *NodeIds* définis ici est http://opcfoundation.org/UA/DI/

Le fichier CSV publié avec la présente version de la norme peut être consulté à l'adresse:
http://www.opcfoundation.org/UADevices/1.1/NodeIds.csv

NOTE 1 Le plus récent fichier CSV qui est compatible avec la présente version de la norme peut être consulté à l'adresse:

http://www.opcfoundation.org/UADevices/NodeIds.csv

Une version informatisée du Modèle d'information complet défini dans la présente norme est également fournie. Elle suit la syntaxe de Schéma de Modèle d'information XML défini dans l'IEC 62541-6.

Le Schéma de Modèle d'information publié avec la présente version de la norme peut être consulté à l'adresse:

http://www.opcfoundation.org/UADevices/1.1/Opc.Ua.Di.NodeSet2.xml

NOTE 2 Le plus récent Schéma de Modèle d'information qui est compatible avec la présente version de la norme peut être consulté à l'adresse:

http://www.opcfoundation.org/UADevices/1.1/Opc.Ua.Di.NodeSet2.xml

Bibliographie

IEC 61131 (toutes les parties), *Automates Programmables (AP)*

IEC 61499-1:2012, *Blocs fonctionnels – Partie 1: Architecture*

IEC 61784 (toutes les parties), *Réseaux de communication industriels – Profils*

IEC 62591, *Réseaux de communications industriels – Réseau de communications sans fil et profils de communication – WirelessHART™*

IEC 62769 (toutes les parties), *Intégration des appareils de terrain (FDI)*

UA Companion ADI, *OPC UA Companion Specification for Analyser Devices* (disponible en anglais seulement)

UA Companion PLCopen, *OPC UA Companion Specification for PLCopen* (disponible en anglais seulement)

_____

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

3, rue de Varembé
PO Box 131
CH-1211 Geneva 20
Switzerland

Tel: + 41 22 919 02 11
Fax: + 41 22 919 03 00
info@iec.ch
www.iec.ch